ISBN 978-1-939133-40-3

usenix ASSOCIATION

Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)

Santa Clara, CA, USA July 10–12, 2024

conference

proceedings

18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)

Santa Clara, CA, USA

July 10–12, 2024

Sponsored by

usenix® THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

OSDI ’24 Sponsors

Platinum Sponsor

amazon

Diamond Sponsor

ROBLOX

Gold Sponsors

databricks

FUTUREWEI Technologies

Microsoft

NSF

Silver Sponsors

Bloomberg Engineering

Google

Jane Street

Meta

Bronze Sponsors

Alibaba Cloud

Bai du 百度

IBM

NetApp

TWO SIGMA

VMWARE UNIVERSITY RESEARCH FUND

Open Access Sponsor

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

USENIX Supporters

USENIX Patrons

Futurewei • Google • Meta

USENIX Benefactor

Bloomberg

USENIX Partner

Thinkst Canary

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

18th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’24)

July 10–12, 2024
Santa Clara, CA, USA

ISBN 978-1-939133-40-3

Symposium Organizers

Program Co-Chairs

Ada Gavrilovska, *Georgia Institute of Technology*
Douglas B. Terry, *Amazon Web Services*

Program Committee

Atul Adya, *Databricks*
Nitin Agrawal, *Google*
Marcos K. Aguilera, *VMware Research*
Peter Alvaro, *University of California, Santa Cruz*
Nadav Amit, *Technion—Israel Institute of Technology*
George Amvrosiadis, *Carnegie Mellon University*
Mahesh Balakrishnan, *Confluent*
Adam Belay, *MIT CSAIL*
Daniel S. Berger, *Microsoft Research*
Abhishek Bhattacharjee, *Yale University*
Laurent Bindschaedler, *Max Planck Institute for Software Systems (MPI-SWS)*
Ken Birman, *Cornell University*
Haibo Chen, *Shanghai Jiao Tong University*
Kang Chen, *Tsinghua University*
Vijay Chidambaram, *The University of Texas at Austin*
Byung-Gon Chun, *Seoul National University and FriendliAI*
Tyson Condie, *Databricks*
Landon Cox, *Microsoft*
Natacha Crooks, *University of California, Berkeley*
Heming Cui, *University of Hong Kong*
Angela Demke Brown, *University of Toronto*
Prabal Dutta, *University of California, Berkeley*
Jason Flinn, *Meta*
Pedro Fonseca, *Purdue University*
Aishwarya Ganesan, *University of Illinois at Urbana–Champaign and VMware Research*
Roxana Geambasu, *Columbia University*
Jana Giceva, *Technische Universität Munich*
Moises Goldszmidt, *Apple*
Alexey Gotsman, *IMDEA Software Institute*
Haryadi Gunawi, *University of Chicago*
Andreas Haeberlen, *University of Pennsylvania and Roblox*
Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*
Steven Hand, *Google*
Henry Hoffmann, *University of Chicago*
Jon Howell, *VMware Research*
Ryan Huang, *University of Michigan*
Rebecca Isaacs, *Amazon Web Services*
Junchen Jiang, *University of Chicago*
Sudarsun Kannan, *Rutgers University*
Manos Kapritsos, *University of Michigan*
Sanidhya Kashyap, *EPFL*
Baris Kasikci, *University of Washington*
Anne-Marie Kermarrec, *EPFL*
Samira Khan, *University of Virginia and Google*
Ana Klimovic, *ETH Zurich*
Marios Kogias, *Imperial College London*
Eddie Kohler, *Harvard University*
Dejan Kostic, *KTH Royal Institute of Technology*
Arvind Krishnamurthy, *University of Washington*
Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*
Baptiste Lepers, *Université de Neuchâtel*
Philip Levis, *Stanford University*
Jialin Li, *National University of Singapore*
Ming Liu, *University of Wisconsin—Madison*
Wyatt Lloyd, *Princeton University*
Jay Lorch, *Microsoft Research*
Shan Lu, *Microsoft Research*
Jonathan Mace, *Microsoft Research*
Petros Maniatis, *Google DeepMind*
Z. Morley Mao, *University of Michigan and Google*
Changwoo Min, *Igalia*
Dushyanth Narayanan, *Microsoft Research*
Ravi Netravali, *Princeton University*
Jason Nieh, *Columbia University*
Shadi Noghabi, *Microsoft Research*
Amy Ousterhout, *University of California, San Diego*
Anand Padmanabha Iyer, *Georgia Institute of Technology*
Aurojit Panda, *New York University*
Amar Phanishayee, *Microsoft Research*
Peter Pietzuch, *Imperial College London*
Costin Raiciu, *University Politehnica of Bucharest*
Luis Rodrigues, *INESC-ID and Instituto Superior Técnico, University of Lisbon*
Timothy Roscoe, *ETH Zurich*
Malte Schwarzkopf, *Brown University*
Marc Shapiro, *Sorbonne Université, LIP6, and Inria*
Liuba Shrira, *Brandeis University*
Patrick Stuedi, *Meta*
Michael Stumm, *University of Toronto*
Adriana Szekeres, *VMware Research*
Amy Tai, *Google*
Alexey Tumanov, *Georgia Institute of Technology*
Dmitrii Ustiugov, *Nanyang Technological University*
Geoffrey M. Voelker, *University of California, San Diego*
Andy Warfield, *Amazon*
Hakim Weatherspoon, *Cornell University and Exosteller, Inc.*
Yubin Xia, *Shanghai Jiao Tong University*
Gala Yadgar, *Technion—Israel Institute of Technology*
Neeraja Yadwadkar, *The University of Texas at Austin*
Junfeng Yang, *Columbia University*
Ding Yuan, *University of Toronto and YScope*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Zheng Zhang, *Amazon Web Services*
Yuanyuan Zhou, *University of California, San Diego*

Poster Session Co-Chairs

Aishwarya Ganesan, *University of Illinois at Urbana–Champaign and VMware Research*
Amy Ousterhout, *University of California, San Diego*

Steering Committee

Marcos K. Aguilera, *VMware Research*
Angela Demke Brown, *University of Toronto*
Casey Henderson-Ross, *USENIX Association*
Jon Howell, *VMware Research*
Kimberly Keeton, *Google*
Jay Lorch, *Microsoft Research*
Shan Lu, *University of Chicago*
Timothy Roscoe, *ETH Zurich*
Geoff Voelker, *University of California, San Diego*
Hakim Weatherspoon, *Cornell University and Exosteller, Inc.*

External Reviewers

Chris Branner-Augmon   Nicolaas Kaashoek   Anja Kalaba   Jennifer Lam   Michael Wu

Message from the OSDI ’24 Program Co-Chairs

Dear Colleagues,

Welcome to the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’24).

We are pleased that, once again, OSDI is co-located with the USENIX Annual Technical Conference (USENIX ATC ’24). The two conferences combined are publishing a wide array of exciting papers representing some of the most innovative work in the systems research community. We hope you come away from OSDI and USENIX ATC with new colleagues, new friends, and new ideas.

This year, OSDI received 272 submissions. We accepted 49 submissions, which is an 18% acceptance rate. In addition to the papers accepted this year, 4 additional papers were accepted from the OSDI ’23 Revise and Resubmit process. This brings the total program to 53 papers being published in the OSDI proceedings. We continue to be committed to a single-track conference at OSDI, and we have put together a 3-day program in which all papers are being presented in the same room so attendees can watch every single talk if they so desire.

To deal with the number and technical breadth of the OSDI submissions, we assembled a program committee of 95 members not counting us, the Program Co-Chairs. Reviewing papers is a time-consuming task that requires high judgment, and we are grateful to the program committee for their diligence, professionalism, and cooperation during the review process. We are proud that OSDI is known for high quality reviews that help authors to produce their very best work and present it clearly to our community.

The program committee reviewed submissions in two rounds. Every paper received at least three reviews in the first round. Select papers then received 2 or 3 additional reviews in the second round. The accepted papers were chosen based on an online discussion phase and a two-day PC meeting. The committee completed more than 1,000 reviews and posted hundreds of comments as part of the online discussion process. Each accepted paper was assigned a shepherd to work with the authors to revise the paper in response to reviewer feedback.

OSDI ’24 had an artifact-evaluation committee, shared with USENIX ATC ’24, that organized and evaluated the artifacts submitted by authors. The committee co-chairs this year were Jianyu Jiang, Ji Qi, and Cesar A. Stuardo. The committee made one recommendation for a Distinguished Artifact Award.

OSDI ’24 had a poster submission process that was run by Aishwarya Ganesan and Amy Ousterhout, who graciously took on this task in addition to serving on the PC. They accepted 52 posters for display at the OSDI poster session, representing a mix of posters from papers presented at the conference as well as other work.

Once the accepted papers were decided, we began the process of deciding on the Jay Lepreau Best Paper Awards. A small committee of non-conflicted PC members read all of the top-ranked papers and agreed on the award recommendation.

As PC chairs, we are grateful to so many dedicated volunteers and professional staff whose efforts have made this conference a reality. We thank the authors who submitted such high-quality work. This conference is first and foremost a forum for disseminating, sharing, discussing, and debating world-class systems research. Thank you for your hard work and innovation! We thank the PC members and external reviewers for their significant investment of time, energy, and insight into shaping the program. We thank Vaibhav Bhosale and Vishal Suresh Rao, who helped us during the PC meeting and made sure we ran the technology and not the other way around. We especially thank the USENIX staff who have made chairing a conference like this one a well-oiled machine! Finally, we thank you for coming to this conference to engage with each other and with the authors of the accepted papers.

We are honored to have served as the OSDI ’24 Program Co-Chairs. Thank you for entrusting us with this important role. We hope that you enjoy the conference!

Ada Gavrilovska, *Georgia Institute of Technology*
Doug Terry, *Amazon Web Services*
OSDI ’24 Program Co-Chairs

18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)

July 10–12, 2024
Santa Clara, CA, USA

Wednesday, July 10

Memory Management

Low-Latency LLM Serving

Distributed Systems

Thursday, July 11

Deep Learning

Operating Systems

Cloud Computing

Formal Verification

Friday, July 12

Cloud Security

Data Management

Analysis of Correctness

ML Scheduling

Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs

Nikita Lazarev Varun Gohil James Tsai[†] Andy Anderson[†] Bhushan Chitlur[†]
Zhiru Zhang[§] Christina Delimitrou
MIT, CSAIL [†]*Intel Labs* [§]*Cornell University*

Abstract

MicroVM snapshotting significantly reduces the cold start overheads in serverless applications. Snapshotting enables storing part of the physical memory of a microVM guest into a file, and later restoring from it to avoid long cold start-up times. Prefetching memory pages from snapshots can further improve the effectiveness of snapshotting. However, the efficacy of prefetching depends on the size of the memory that needs to be restored. Lossless page compression is therefore a great way to improve the coverage of the memory footprint that snapshotting with prefetching achieves. Unfortunately, the high overhead and high CPU cost of software-based (de)compression make this impractical.

We introduce Sabre, a novel approach to snapshot page prefetching based on hardware-accelerated (de)compression. Sabre leverages an increasingly pervasive near-memory analytics accelerator available in modern datacenter processors. We show that by appropriately leveraging such accelerators, microVM snapshots of serverless applications can be compressed up to a factor of 4.5×, with nearly negligible decompression costs. We use this insight to build an efficient page prefetching library capable of speeding up memory restoration from snapshots by up to 55%. We integrate the library with the production-grade Firecracker microVMs and evaluate its end-to-end performance on a wide set of serverless applications.

1 Introduction

Serverless is an emerging cloud computing paradigm gaining widespread popularity across applications of different classes, from lightweight interactive services [73] to highly data-parallel applications, such as machine learning and video encoding [21, 24, 40, 47, 67]. Serverless offers a Function-as-a-Service (FaaS) execution model, where applications instantiate short-lived, fine-grained resources on-demand without the overhead of provisioning and deployment typical cloud environments incur. When requests are processed, the resources are terminated, achieving a pay-as-you-go model. This both avoids resource overprovisioning, which has been a long-standing issue with cloud infrastructures [22, 28, 58] and reduces the end-to-end deployment cost [33, 46].

Serverless is based on lightweight virtualization/isolation technologies [19] such as Docker, Google gVisor [3], Kata containers [10], NEC's LightVMs [53], and AWS Firecracker microVMs [17]. These technologies implement sandboxes for executing containerized applications with different levels of isolation. For example, gVisor implements a lightweight user space kernel capable of executing most of the system calls within the sandbox. On the other hand, Firecracker is a full lightweight virtualization technology, based on KVM, which can boot standard Linux kernels in sub-second time [17]. Due to the high isolation guarantees of microVMs, security, and performance, Firecracker is widely used in serverless clouds.

Despite its advantages, serverless and microVMs introduce a few critical overheads to performance. A major overhead is cold starts (or cold boots) – the overhead of the initial boot of container sandboxes upon a function invocation. Both industry and academia have proposed numerous techniques to mitigate cold start overheads [31, 35, 64, 73], with one of the most promising being VM snapshotting [31]. VM snapshots capture the current state of the VM and its physical memory and save them in a file. During the next boot, the guest system is restored from the file instead of booting from scratch. Several different techniques can be used to make snapshots, depending on which parts of the guest's physical memory should be saved. For instance, Firecracker can snapshot the full guest memory or only the dirty pages. Additionally, recent studies have proposed using *working sets* of pages [71] to make serverless VM snapshots smaller, faster to fetch, and overall more efficient [20, 64] in reducing cold starts. However, independent of the underlying techniques used to create snapshots, the overhead of storing and prefetching them is non-negligible. Unfortunately, the latter is on the critical path of VM restoration and therefore directly impacts cold starts.

Reducing the size of snapshots can make them significantly more efficient, for example through lossless memory compression. While memory compression has been used in domains where the application performance is not critical (e.g., *zswap* [39], *zram* [38] in Android OS for mobile devices), in serverless the restoration of a memory snapshot is on the critical path, precluding the use of existing software-based (de)compression algorithms. At the same time, there are numerous hardware implementations of (de)compression [23, 41, 51, 52, 56], however, they had not, until now, been implemented in mainstream datacenter processors [43]. In particular, Intel recently released the In-Memory Analytics Accelerator (IAA) [6] in their 4^{th} Gen Xeon Scalable CPUs, which enables efficient compression for datacenter applications at scale.

We present Sabre, a hardware-accelerated *general-purpose* memory prefetching system, which uses lossless compression mechanisms, such as IAA, to compress and restore microVM snapshots. This paper makes two major contributions. First, we characterize, for the first time, the IAA accelerator on a set of diverse benchmarks, and show its potential for compressing memory pages. We show that IAA can compress pages by $2-4.5\times$, depending on the underlying page selection algorithm. Moreover, we show that decompression can be done up to $10\times$ faster with hardware acceleration, and with careful design, this time can be entirely hidden behind the disk I/O and page fault handling. This results in near-free decompression in terms of the overall memory restoration latency, while reducing the size and the loading time of snapshot pages.

Second, based on this characterization, we build Sabre and integrate it with the Firecracker virtual machine monitor (VMM) in a serverless environment with snapshotting support. Sabre is agnostic to the underlying page snapshotting policy, it operates entirely in the host's user space, and interacts with the IAA accelerator via the Shared Virtual Memory (SVM) mechanism. The latter enables out-of-box and transparent integration of Sabre with existing VMMs at scale.

We evaluate Sabre on its efficiency in restoring microVMs from snapshots across a wide range of end-to-end serverless applications using two methods of creating snapshots: dirty page-based and working set-based. We show that Sabre compresses microVM snapshots up to $4.5\times$ without introducing any decompression overheads. Moreover, we show that Sabre enables up to 55% faster memory restoration, which results in an additional reduction of the end-to-end cold start time by 20% with respect to already optimized state-of-the-art snapshotting baselines.

Sabre is open-source software and it is available at the following link [13].

2 Background

2.1 MicroVMs for Serverless

Serverless is gaining popularity across many application domains by reducing the cloud provisioning overhead and enabling higher elasticity for applications with high parallelism and intermittent activity. Lightweight virtualization technologies (or microVMs) became a popular choice for cloud providers due to the isolation and fast instantiation they provide [17, 69]. Fully virtualized VMs running over Type-1 hypervisors, such as KVM [37] or Hyper-V [70], allow isolating tenants down to hardware and provide the highest security guarantees for applications running in the cloud. On the other hand, microVMs are much faster to boot and have a much smaller memory footprint than traditional Type-1 virtual machines. This means that microVMs achieve the best of both worlds between containers and Type-1 hypervisors. MicroVMs are widely used in serverless, where applications require both strong isolation guarantees and fast start-up.

Modern microVMs, such as AWS Firecracker [17], use several optimizations to boot up in sub-seconds. However, booting the VM itself is only part of the end-to-end application execution latency [31, 64], with a significant component corresponding to the initialization of the software dependencies after the boot. For applications based on complex multi-layer stacks, such as gRPC servers and JavaScript runtimes, bringing up the dependencies might be as high as several seconds [31]. Additionally, the applications themselves can contain long-running initialization routines, which also contribute to end-to-end latency. For example, machine learning (ML) services need to load the models before serving inference queries. Altogether, this makes the end-to-end execution of the first batch of requests running on freshly booted microVMs an order of magnitude slower than subsequent requests. This is known as *cold start*, and all microVMs are prone to it.

Mitigating cold starts is one of the most well-researched aspects of microVMs [65]. Existing solutions range from scheduling techniques optimized for specific applications to runtime and infrastructure optimizations [54, 73]. One solution that is generally agnostic to applications is VM snapshotting [31].

2.2 MicroVM Snapshotting and Prefetching

Snapshotting is a technology that allows storing the VM state and guest OS physical memory in a file in the local or remote filesystem. Snapshots are usually created after the VM and the application logic with all its dependencies are fully initialized and ready to serve requests. Upon the next invocation of the VM the hypervisor restores the VM state and guest memory from the snapshot, instead of booting the VM from scratch. This dramatically reduces cold start overheads.

In the most basic case, snapshots contain the entire guest physical memory. Some hypervisors, such as Firecracker, also allow dirty-memory tracking, which only stores the dirty guest pages as seen by the hypervisor. Snapshots can be organized hierarchically following the software dependencies of applications [31]. However, recovering from snapshots is far from free. In some cases, the size of the snapshots can be as high as the whole guest memory, therefore precluding the possibility of loading pages from snapshots in advance. For this reason, existing commercial microVMs implement memory restoration via *on-demand* paging. Unfortunately, on-demand paging yields a lot of page faults on the critical path of the restoration from snapshots, which slows down request execution.

A way to reduce the overhead of page faults is to enable *prefetching* of pages from snapshots. This can be efficiently done through, for example, *working set* (WS) estimation. This approach has been used to create VM checkpoints [71], and recently – for serverless microVMs [64]. Here, each snapshot is accommodated in a WS file, storing pages that are likely to be accessed during subsequent invocations. There are different ways of constructing WS files [20, 64, 71] according to various working set estimation techniques. For example, in Record-and-Replay (REAP) [64], the authors propose to record all guest pages being accessed during the first invocation of serverless functions and put them into the WS file. Upon the next invocation, the WS file can be prefetched from the disk, and the WS pages can be installed in the guest's memory to speed up the next cold invocations. REAP works well for applications with a similar working set across different invocations of the same function. When this does not hold, REAP can fail to deliver good performance; in this case, prefetching some other subset of dirty pages (or even all dirty pages) can be more beneficial. Snapshots generally consume a lot of disk space and require cloud providers to carefully provision their storage resources [2]. Even working-set-based snapshots can sometimes be as large as a few hundred megabytes [20]. This is non-negligible given that a single server might host hundreds of microVMs.

Independently of the underlying technique to create microVM snapshots and/or WS files, the efficiency of prefetching depends on the memory size that needs to be restored from the disk into the guest memory. A general rule to make prefetching-based techniques more efficient is to reduce the size of the snapshots/WS files. This also reduces the disk space needed to store snapshots. This size reduction can be achieved through memory compression. However, memory restoration happens on the critical path of the VM boot-up, and for compression algorithms with high deflate ratios, the decompression might take a long time. Moreover, such algorithms usually consume a lot of CPU time for compression and therefore VM snapshotting. This makes the use of software memory compression for microVM snapshotting undesirable.

2.3 Hardware-Accelerated (De)Compression

Software-based memory compression has been extensively used in applications where performance is not critical. For example, *zram* [38] is used in Android OS on mobile devices, while *zswap* [39] can improve the efficiency of memory swapping for non-performance critical applications. Unfortunately, this does not apply to microVM memory restoration, where decompression directly impacts the cold start overhead.

There have been many proposals for accelerating memory compression in hardware. For example, Pekhimenko et. al. [56] propose base-delta compression for on-chip caches. Hoyong et. al. [41] derive a novel compression algorithm for GPU memory. Li et al. [52] introduced a hardware accelerator for the compression of genome sequences. All such proposals are based on application-specific, special-purpose compression accelerators and algorithms, and therefore have never been implemented on commodity datacenter processors. Many (de)compression accelerators are based on FPGA cards [25, 34, 50, 57] which are only available in a small set of public clouds. However, the demand for *general-purpose* (de)compression acceleration *at scale* is actively growing.

(De)Compression is known to be one of the major sources of datacenter tax [36,42,61]. A recent study from Google [36] showed that compression accounts for up to 30% of cycles for large-scale database applications, such as BigTable and BigQuery [32,62], and it is also extensively used in many other applications. This motivates cloud providers and chip vendors to build efficient hardware accelerators [43] for *general-purpose* lossless (de)compression. The primary use cases for such accelerators are databases and query-processing engines. In particular, Intel recently introduced the In-Memory Analytic Accelerator (IAA), which is now part of commodity datacenter processors, such as the Xeon 4^{th} Generation CPUs, which are already widely available. This accelerator can perform DEFLATE compression, which is suitable for compressing memory footprints. While other compression algorithms that are optimized for performance (e.g., *Snappy*, *zstd*, *LZ4*) can compress memory faster, they typically result in much lower compression ratios, which is critical for microVM snapshotting. Their software implementations increase the amount of CPU resources required for making snapshots, especially when configured for more aggressive compression [1]. At the same time, hardware-accelerated DEFLATE yields high compression ratios as well as high speed, while requiring no CPU cycles for (de)compression. It should also be noted that hardware accelerators for other compression algorithms are also feasible [16, 26, 60] but not yet implemented in mainstream datacenter processors at scale.

IAA and other similar accelerators are designed for cloud environments. They are typically implemented as on-chip near-memory PCIe components, which allows them to be easily integrated with cloud services. For instance, IAA can run entirely in user space, it operates transparently over the application's virtual memory and can be virtualized through standard technologies, such as S-IOV [8]. All this makes IAA attractive for microVM memory snapshotting and prefetching. In this work, we explore this direction.

We first characterize the capabilities of IAA when it comes to compressing memory pages. Based on this characterization, we design Sabre, a memory prefetching system for microVMs built using IAA. Finally, we show how our memory prefetching unit integrates with serverless microVMs and evaluate its impact on end-to-end serverless benchmarks.

3 In-Memory Analytic Accelerator: Overview, Characterization, Insights

We now present an overview and characterization of the Intel In-Memory Analytic Accelerator (IAA) [6] using a set of diverse benchmarks [15]. This work mainly focuses on the compression/decompression capabilities of the accelerator. However, given that other capabilities share the same IAA frontend pipeline, interfaces, and software semantics, most of our findings also apply to these other domains. To our knowledge, this is the first publicly available characterization of the IAA hardware. The insights from this characterization are used to derive the design of Sabre, described in Section 4.

Note that given the diverse set of execution models, configurations, workloads, and variations of IAA hardware in different SKUs, it may be possible to achieve even higher performance compared to what we showcase in our characterization. Specifically, benchmarks designed specifically to stress test the accelerator may be able to improve IAA's performance further. For the purpose of this paper, we only benchmark the accelerator with a set of scenarios required to give comprehensive insights into using in-memory compression techniques in serverlerss microVMs and to derive the design of Sabre.

3.1 Overview of the IAA

Intel's IAA is a hardware accelerator first introduced in Intel's 4^{th} Gen Xeon Scalable Processors (code-named Sapphire Rapids) [12] to speed up data processing across application classes. It was designed with the primary use case being databases and query processing systems [7]. The accelerator physically resides in the uncore part of the processor's SoC near the memory controller and Last Level Cache. A single CPU can accommodate multiple IAA devices on its SoC.

The IAA accelerators are logically integrated as PCIe devices and exposed to the host as a single root complex integrated endpoint. This is set up to enable transparent integration of the accelerators with software. IAA features scalability, full virtualization support via PCIe S-IOV, Shared Virtual Memory support (SVM or SVA as defined by Linux kernel documentation) [11], and transparent user-space interaction with applications via a new ISA extension, called *ENQCMD*.

Communication and job submission to the accelerator are handled via *Work Queues* (WQs), similar to another emerging hardware – Data Streaming Accelerator (DSA) [48]. For this, software needs to create *descriptors* and *completions* allocated anywhere in the application virtual address space. Descriptors contain information describing the jobs assigned to the accelerator's *Processing Units* (PEs), such as the locations of the source and destination buffers, opcodes, and operational and memory policy flags. Descriptors are submitted to the accelerator via the ENQCMD instruction directly from user space, which writes them into the device's memory-mapped I/O (MMIO) registers. Upon receiving descriptors, the PEs fetch data based on the pointers in the descriptors. This is done through SVM which enables transparent sharing of the application's virtual memory with accelerators. When a PE finishes processing, it writes the corresponding completion record with the status information. The software can poll the completion records to identify the termination of tasks and any error information. If some application memory pages associated with data buffers are not available, the accelerator can request them via either *Page Request Service* (PRS) or through userspace page fault handling. In the latter case, software applications can resolve the page faults in a more suitable for a particular usage scenario way (e.g., by requesting pages from, for example, the network) in the user space.

IAA's job submission mechanism enables asynchronous/non-blocking and out-of-order processing of descriptors. The current hardware permits a large number of in-flight requests, which can be submitted from different threads and processes/tenants. The micro-architectural pipeline of the IAA hardware contains multiple PEs that can execute jobs concurrently. The DSA specification [4] and in-depth characterization [48] contain more details since both accelerators share the same specification for this part.

The IAA hardware contains PEs implementing different processing capabilities. These include encryption, compression, CRC offload, data filtering, scanning, extraction, selection, and expansion [5] (Table 3-1). Due to the scope of this work, we only focus on characterizing the (de)compression capability of IAA. IAA performs DEFLATE [30] compression, as defined in RFC 1951. DEFLATE is based on LZ77 matching and Huffman encoding. LZ77 matching eliminates redundancy by replacing repeated occurrences of substrings with references to a single version of the substring. This is a computationally intensive process making software implementations slow. The Huffman coder further deflates data by re-encoding the most common symbols with fewer bits using statistics of data distribution in the input stream. Internally, the IAA compression unit operates in three modes.

In the first – *Huffman-mode*, IAA performs hardware-accelerated LZ77 dictionary coding, using 4 KB windows, and encodes the results with pre-defined static Huffman tables. The second IAA mode – *Statistics-mode* is designed to sample input streams and construct the statistical data distribution to optimize Huffman tables for a particular input. Compression with input-specific Huffman tables usually allows much higher compression ratios. In Statistics-mode, IAA only constructs the histogram of the distribution of Huffman codes, but it does not write the actual Huffman tables yet. The latter

Style	Description
Fixed Block	Standard static DEFLATE; based on Huffman-mode with standard Huffman tables; enables faster compression, but under general Huffman tables, which usually results in low compression ratios.
Static Block	Similar to Fixed Block, but using user-defined Huffman tables; can result in good compression ratios if the application is able to provide Huffman tables fitting all inputs well.
Dynamic Block	Standard dynamic DEFLATE; two-phase compression with Statistics-mode followed by Huffman-mode; enables optimal Huffman tables per block and a better compression ratio, but requires more time to compress.
Canned	Allows sharing the same Huffman tables between multiple blocks of compressed data; this is important when compressing many small scattered chunks to avoid having to keep/access Huffman tables per block.

Table 1: End-to-end compression styles supported in Intel IAA.

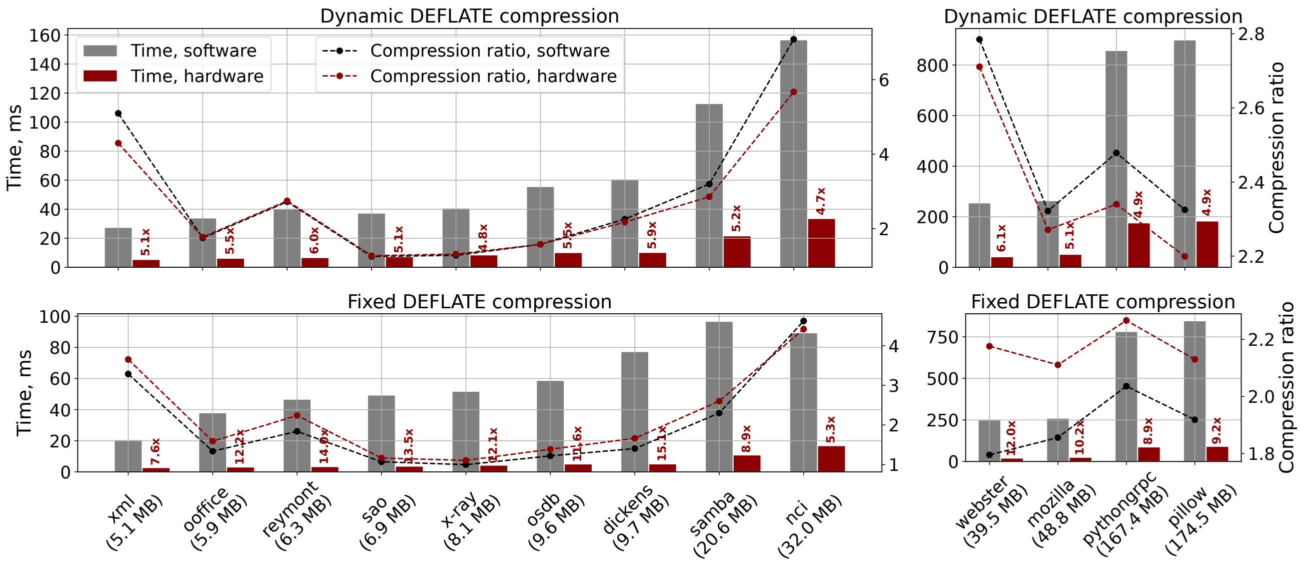


Figure 1: Comparison of software- and hardware-based DEFLATE *compression* for different datasets (sorted by uncompressed size) from Silesia Corpus and serverless VM snapshots (last two); the numbers denote speed-up of the hardware execution; a single IAA device with a single PE (engine) in blocking/synchronous mode is used; the software baseline runs on a single thread.

is done in the third mode – *Huffman-Generation mode*, which is only supported in some IAA implementations.

Based on these modes, IAA defines four main *styles* of compression (Table 1), and it is up to the control plane software to implement them. To make end-to-end (de)compression easier to implement, Intel has recently released the Query Processing Library (QPL) [7], which abstracts away the IAA modes and allows users to express compression in any of the supported modes. We next show the results of microbenchmarking IAA with different modes and implementations.

3.2 Characterizing Compression Using IAA

We characterize IAA (de)compression with a set of benchmarks written using the Intel QPL library v1.3.1. As input data, we use 11 datasets from the standard Silesia Corpus [29]; a common way to evaluate (de)compression. We also add two more datasets specific to our use case representing the dirty memory snapshots of microVMs. The snapshots were obtained during request execution for two serverless applications from vSwarm [66] and FunctionBench [44,45]: a Python gRPC server (*pythongrpc*) and the Pillow image processing library (*pillow*). Both datasets only contain dirty pages of guest memory. Table 2 shows the specification of our testbed. At the time of writing, we had access to two Sapphire Rapids systems (SKUs) with slightly different configurations. We use the most recent production-grade SKU (*Server #2* in Table 2) in all experiments unless otherwise noted.

3.2.1 Benchmarking IAA: Core Compute

We start with the characterization of the in-memory core computing capability of IAA. We assume that data is always available in memory, both for the source and destination buffers. We ensure that memories are initialized and touched to avoid page faults. This is important as page fault handling affects the accelerator's performance, and therefore we evaluate it separately. For all experiments in this section, we use a single

CPU
(Server #1)
	Intel 4^{th} Gen Xeon Scalable Processor;
	2 NUMA nodes, 56 cores/112 threads;
	Core/Uncore frequency (GHz): 1.7/ 1.8;
	LLC capacity (MB): 110
	IAA devices: 8 (4 per NUMA node)

CPU
(Server #2)
	Intel(R) Xeon(R) Gold 6438Y+;
	2 NUMA nodes, 32 cores/64 threads;
	Core/Uncore frequency (GHz): 2.3/ 1.8;
	LLC capacity (MB): 60
	IAA devices: 2 (1 per NUMA node)

IAA	available PEs per device: 8 capabilities (as per *GENCAP* register): - Huffman generation mode: *disabled*; - Page Request Service (PRS): *enabled*; - Block-on-Fault: *enabled*; WQ configuration: shared, 8 per device, size: 32
Memory	Type: DDR5; Capacity (GB): 250
Disk	Intel SSDSC2KG960G8 Sequential O_DIRECT read bandwidth (MB/s): 550

Host OS	Ubuntu 22.04;
	Kernel: 5.15, patched with [9] to enable ENQCMD;
	Kernel boot arguments: $intel\_iommu = on, sm\_on$

Guest OS **(Section 5)**	Rootfs: Debian GNU/Linux 12 (bookworm) Kernel: 4.14.174
IAA stack	Driver: idxd Middleware: Intel QPL v1.3.1

Table 2: Testbed hardware and software configuration.

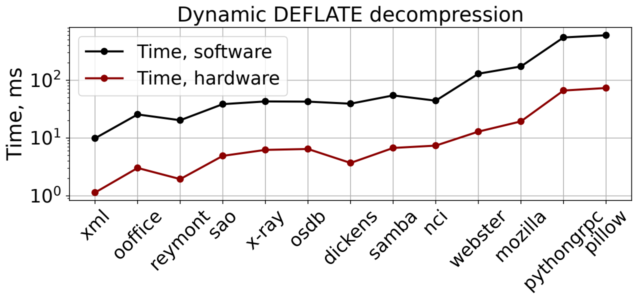


Figure 2: Comparison of software- and hardware-based DEFLATE *decompression* on the same datasets and setup as in Figure 1

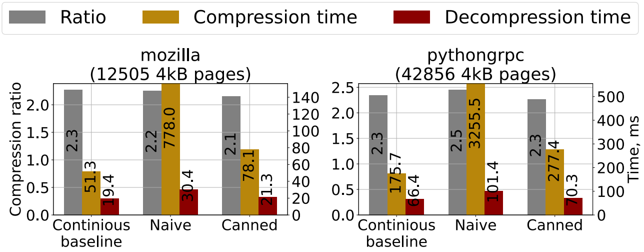


Figure 3: (De)compression of scattered 4kB pages across modes.

IAA device configured with a single PE (engine); we submit the jobs to the accelerator from a single CPU thread in the blocking/synchronous mode and wait till completion by polling associated completion records. The software baseline runs on a single CPU core.

We first compare the performance and compression ratios of IAA-enabled compression and its software implementation in QPL. Since the IAA hardware does not allow explicitly selecting compression levels (compression levels are subjective and vary across implementations), we set the default compression level-1, as defined by QPL, for the corresponding software implementation. Since our version of IAA does not offload Huffman table generation, the hardware implementation of dynamic DEFLATE compression is actually hybrid: statistics collection, LZ77 encoding, and compressed stream generation run in hardware, while Huffman table generation runs in software. The hybrid operations run on a single CPU core. Figure 1 shows the compression results.

The hardware implementation always overperforms software in compression time. The difference reaches 6.1× and 13.5× for dynamic and fixed compression, respectively. In our datasets of dirty memory snapshots, the speedup reaches 9×. The achieved compression ratios for software and hardware executions are similar. Figure 2 shows the performance of decompression. We only show the decompression of *dynamic* streams, as in these datasets, decompression performance does not depend on the compression mode. In all cases, IAA decompresses an order of magnitude faster than software.

Note that in this paper, our experiments only compare IAA to the software implementation of the same DEFLATE algorithm. To compare against many other software compression algorithms (e.g., Snappy, zstd, LZ4, etc.), please refer to the publicly available in-memory benchmarks based on the Silesia Corpus (for example, lzbench [1]). The *synchronous* throughput of IAA's (de)compression can be obtained from Figures 1 and 2 based on the size of the datasets. For example, the fixed-DEFLATE compression on the *nci* dataset reaches 1800 MB/s, and its decompression - 4600 MB/s on a single engine. These numbers are expected to be lower than the *asynchronous* streaming (de)compression (using the non-blocking mode of IAA) which we do not characterize in this paper.

We then profile (Figure 3) IAA's (de)compression for many small 4 KB sized chunks for the same datasets. As previously mentioned, the Canned—which is essentially a Static Block—mode allows sharing Huffman tables between data chunks, therefore reducing both the space and processing time when data is scattered over many small blocks. This is very useful when compressing individual memory pages. In Figure 3, the first group of bars shows the baseline compression using Dynamic Block over a continuous region. We then break it into 4 KB chunks and compress them naïvely, with the Dynamic Block, independently for each chunk. As a result, compression time explodes due to processing tables separately; the decompression time also suffers, as the Huffman tables need to be parsed. The Canned operation reduces the overhead of scattered compression. Most interestingly, it enables fast decompression of scattered data, which is only marginally higher than the continuous baseline.

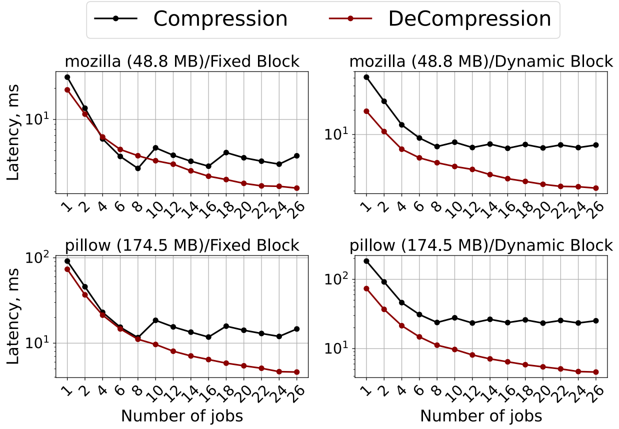


Figure 4: Latency of single-thread *synchronous* (de)compression with parallel *hardware* execution on 4 IAA devices with 8 engines each; only two benchmarks are shown for brevity.

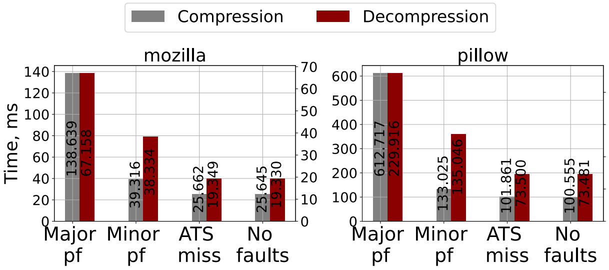


Figure 5: Impact of page faults and translation fetch on IAA performance via *PRS* with *block-on-fault* enabled

.

3.2.2 Benchmarking IAA: System Integration

We now characterize the IAA hardware together with system-level aspects. We use the same datasets and setup as before.

We first evaluate how job parallelization within the IAA PEs may further speed up (de)compression. Given that our earlier SKU has more IAA engines/PEs (refer to Table 2), we use *Server #1* in this experiment. Note that at the time of writing, the QPL library did not allow crossing NUMA boundaries between the data and IAA devices. Therefore, only 4 IAA devices (32 PEs/engines in total) are utilized at most for this experiment.

There are two main ways to leverage parallelism within the IAA: with synchronous and asynchronous job submission. In the first case, a large chunk of data can be split into multiple smaller blocks, and these blocks are then submitted to multiple available PEs. The software then blocks and waits until all PEs finish processing. This allows us to reduce the time/latency of a single (de)compression job. In the asynchronous case, a *stream* of multiple blocks from potentially different dataflows/threads is supplied into the accelerator without waiting for the completion of the previous blocks, therefore utilizing the hardware at maximum capacity. This yields the highest IAA utilization and *throughput*. For fast microVM restoration, the only performance metric that matters is how fast the system can decompress a single snapshot from a single CPU thread into the microVM guest memory. We therefore only benchmark the synchronous job parallelization in this paper.

We implement parallel synchronous processing via the *non-blocking synchronous* descriptor submission. Here multiple descriptors are submitted at the same time from a single CPU thread without waiting for immediate completion of individual descriptors. Figure 4 shows that for a hardware concurrency of up to 8 compression jobs, the latency reduction reaches $4-7\times$ with respect to sequential execution. Dynamic Block compression scales worse due to the software overhead of Huffman table creation. *Decompression* scales up to 26 jobs, reaching 17× latency reduction for the *pillow* dataset. These results demonstrate how multiple IAA engines can be used to achieve even lower (de)compression latency.

Until now, we have only tested the behavior of IAA when processing in-memory data, which was the majority of initial use cases for the accelerator. In-memory operation is achieved when source and destination buffers are present both in the memory and page table of the calling process. This holds when, e.g., streaming over the same buffers. However, in certain cases, data is not entirely present in memory, e.g., when processing inputs from a file or into newly allocated memory. In these cases, the accelerator must resolve page faults.

The fundamental source of page faults in systems such as IAA and DSA is the fact that they operate directly on the application's virtual address spaces via SVM [11]. As a result, similarly to CPU processing, when a page requested by the accelerator is not found in the page table, a major or minor page fault occurs. The result of the page fault handling (e.g., the translation) is then cached in the accelerator's Address Translation Service (ATS). When the translation is available in the CPU/kernel, the page fault does not happen, but the ATS must fetch the translation from the host via a translation fetch request. We now benchmark IAA in the case of page faults and translation fetch requests. We use standard 4 KB pages and an SSD disk with 550 MB/s of provisioned sequential read bandwidth. For compression, we use a single-pass Fixed Block to avoid the side effects of hybrid two-phase operations.

IAA supports two modes to handle page faults: via hardware-initiated on-demand paging via PRS or in user space with custom application-defined page fault handlers. The mode is controlled via the work queue configuration or through the PCIe device configuration if the former is not available. When PRS is enabled, the hardware can request up to *N* (specified in *PRSREQCAP* register) pages from the host concurrently. The actual page fault handling is done by the kernel through IOMMU interrupts. Figure 5 shows the time required to process descriptors in case of different hardware-

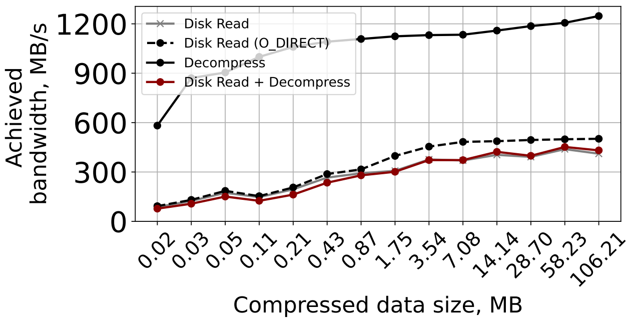


Figure 6: *Single-job, single-engine, synchronous* decompression bandwidth when reading disk inputs via *PRS* with *block-on-fault* enabled.

initiated page faults. We benchmark the worst-case scenario when the entire dataset causes page faults (12492 pages for *mozila* and 44672 pages for *pillow*). As expected, major page faults have a severe impact on the accelerator's performance. The impact of minor page faults is lower, but still $\approx 2\times$ for decompression. The ATS translation fetch has the smallest impact of ≈ 20 *us* across all pages. In general, decompression is more sensitive to page faults than compression; this is because the former is less compute-bounded.

Finally, we characterize the achieved decompression bandwidth when reading input data from an SSD (Figure 6), which is critical for our memory restoration use case. The dashed black and gray lines show achieved sequential disk read bandwidth with and without *direct I/O* (O_DIRECT). Direct I/O allows bypassing the page cache when reading files from the disk. In certain cases, this helps to reach the highest bandwidth of I/O operations. The solid black line shows the achievable bandwidth of a single-job synchronous decompression over data in memory, and the red line when reading input from the disk via hardware on-demand paging (i.e., via PRS). The latter is achieved through shared mapping of the input file into the IAA buffers and enforcing sequential I/O using *posix_fadvise*.

IAA decompression is a *streaming* operation, and it always accesses input buffers sequentially. In an ideal system, the decompression phase will completely overlap with the operation fetching data from the disk. Hence, the achieved end-to-end throughput will be decided by the slower operation - be it decompression or disk I/O. As Figure 6 shows, with default hardware on-demand paging, the achieved end-to-end bandwidth is the same as disk read without direct I/O. This demonstrates that IAA streaming processing can entirely overlap with disk I/O. It is $10-15\%$ lower, however, than the achievable bandwidth of reads with direct I/O, because IAA communicates with the disk via the OS page cache when running over PRS. A way to further improve IAA over data from disk is to replace PRS with application-specific page fault handling. However, in that case, the end-to-end behavior depends on whether IAA is configured with enabled *block-on-fault*, a feature that allows the accelerator to block and wait until data becomes available. Without block-on-fault, IAA terminates with partial completion and cannot continue decompression from the place where it stopped; as a result, the job needs to restart from scratch. Given the streaming nature of IAA, it is possible to entirely close the gap between direct disk I/O and decompression by using block-on-fault in combination with O_DIRECT reads in a custom page service handler, either in a driver or user space. We leave this to future work.

4 Sabre Design

4.1 Memory Prefetching Accelerator

We use the insights from the characterization study of Section 3 to design Sabre. Sabre is a hardware-accelerated memory snapshotting and restoration system for microVMs that is agnostic to the underlying algorithm used to identify dirty pages and create VM snapshots.

Sabre is designed to efficiently compress the guest VM physical pages to create snapshots, such that they can be decompressed (and mapped) quickly when restoring the snapshot upon a function invocation resulting in a cold start. As an input, Sabre accepts a vector of addresses for each of the guest physical memory pages which need to be placed in the snapshot, according to the underlying dirty page selection mechanism. It then compresses pages using IAA and writes them in a file. During the VM restoration process, Sabre uses fast IAA decompression in combination with efficient sequential disk I/O to quickly fetch the pages from the snapshot, decompress, and install them in the target VM's physical memory. The main goal of Sabre is to hide the decompression latency as much as possible behind the disk I/O and page fault handling (when mapping pages) such that the overhead of decompression is minimized. This is possible to achieve given the streaming nature of IAA decompression.

Sabre shows that irrespective of the method used to identify dirty pages and create VM snapshots, hardware-accelerated compression, and restoration can have a significant impact on performance. In the simplest case, dirty pages can be identified by the page tracking mechanism in the VMM (e.g., Firecracker's *Diff* snapshots); in more complicated cases, custom algorithms can be used (e.g., different working set estimation techniques [64, 72]).

Figure 7-A shows an overview of Sabre's snapshot creation pipeline along with two designs for memory prefetching, both of which are used by Sabre under different scenarios.

Snapshot creation: We first describe our snapshot creation process, which is based on two observations. First, creating a snapshot is outside of the VM restoration's critical path, so the objective is selecting the compression algorithm that achieves the highest compression ratio, which as we showed in Section 3 is dynamic DEFLATE. Second, since the VM dirty pages are distributed in a non-contiguous manner across the guest's physical memory space, the (de)compressor should operate over separate (often small) chunks of memory. As

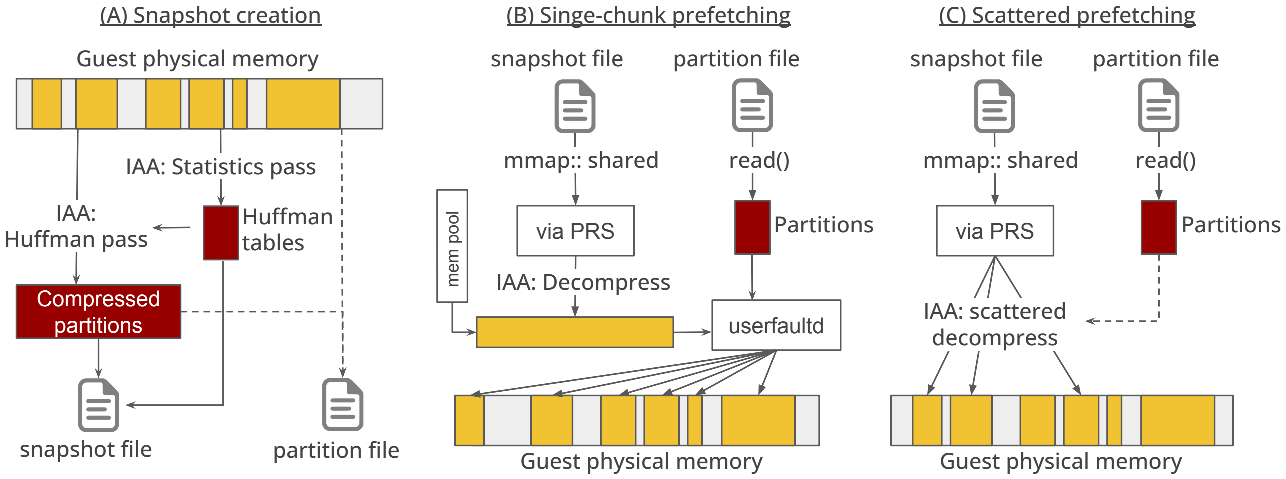


Figure 7: High-level overview of snapshotting with Sabre.

Figure 3 shows, the "Canned" style compression works best in this case: it enables implementing static DEFLATE with pre-computed Huffman tables, which closely resembles the efficiency of Dynamic DEFLATE. Sabre's snapshot creation process first runs IAA in Statistics mode to sample the statistical distribution of data in all dirty pages and create appropriate Huffman tables. It then compresses the scattered regions of dirty pages with these Huffman tables. The resulting compression stream alongside the Huffman tables is written into the snapshot file. For experimentation reasons, we also enable using Dynamic DEFLATE as well. In addition to the snapshot file itself, Sabre also writes the partition file containing the "schema" of the dirty pages, i.e., the offsets and original/compressed sizes of each partition.

Memory prefetching: The memory prefetching process is more complicated to engineer as it is on the critical path of VM restoration, and therefore needs to be carefully optimized. The main trade-off Sabre must navigate is balancing the desire to handle all partitions of dirty pages as a single contiguous memory region and the cost that comes with that.

Handling all partitions of dirty pages as a single continuous memory region is better for the accelerator, as continuous DMA is more efficient than scattered DMA, and it is also better for the PRS and disk I/O, as the underlying PRS-initiated page faults are sequential. The latter works well with sequential disk reads, therefore yielding the best utilization in terms of disk bandwidth. However, continuous decompressed partitions need to be placed by the same addresses in the guest VM physical memory as in the original VM when the snapshot was taken. Sabre implements it using *userfaultfd*, which comes at the cost of memory copy.

We implement this approach in Sabre's *single-chunk memory prefetching* shown in Figure 7-B. To reduce the overhead of allocating the decompression buffers, Sabre can optionally use a pre-allocated memory pool for the buffer for the time of decompression. The size of the pool is bounded by the sum of the sizes of dirty pages of the VMs currently restored simultaneously. This space is reusable across different restoration processes and therefore does not consume much memory. However, users of Sabre can always disable the memory pool (at $\approx 10\%$ cost of memory restoration) if the pool's impact on the memory density is an important concern. Sabre's memory prefetching relies heavily on the PRS hardware mechanism to bring snapshots from the disk. This is achieved by running IAA against a shared not pre-faulted (i.e. the actual file I/O gets initiated by PRS) mapping of the snapshot file. As Figure 6 shows, default PRS is near-optimal at handling IAA inputs from the disk, and it allows hiding the decompression time by overlapping it with the disk I/O. In most cases, the difference with sequential disk read bandwidth is marginal. We confirm that running IAA over pre-faulted or pre-fetched (via *read*) snapshot files is much slower than via PRS.

To address the high cost of partition placement when treating the entire memory region as contiguous, Sabre also implements memory prefetching based on scattered IAA decompression (Figure 7-C). Here, Sabre directly DMAs decompressed partitions into the right locations in the guest's physical memory, while still handling inputs from disk via PRS. This allows the system to bypass page installation, however, it makes the IAA hardware less efficient due to the large number of scattered DMAs and the bookkeeping of the corresponding descriptors (the current implementation of IAA does not allow chaining and batching of descriptors, so each one must be submitted separately by software). In addition, splitting the decompression stream into multiple jobs hurts the efficiency of PRS at reading data from the disk, which further slows down memory prefetching. The latter can be addressed by implementing a custom user-space page fault handler, as discussed in Section 3.

In both designs for memory prefetching, IAA decompression can be done using a single IAA job/engine or parallelized across all available engines. This is implemented via non-blocking job submission with a rotating pool of descriptors. In this mode, Sabre attempts to submit N decompression jobs at the same time, where N is the desired concurrency degree or the number of available free engines (whichever is smaller). Upon asynchronous out-of-order completion, the corresponding descriptors are returned to the pool for later reuse. This enables a streaming operation for Sabre's decompression when multiple engines are used. Note that a single IAA engine in blocking synchronous mode is capable of achieving $\approx 1.2\ GB/s$ at decompressing our snapshot datasets. Since this is higher than our disk read bandwidth, we always use a single IAA job in all remaining experiments, unless otherwise noted.

The exact operation of the memory prefetching unit, such as the choice of the restoration design (between single-chunk and scattered), the compression style (dynamic or static DEFLATE) for snapshot creation, the number of concurrent decompression jobs, etc. are configured by Sabre during runtime. The desired configuration can be selected differently for each microVM's snapshotting/restoration call. Next, we microbenchmark our memory prefetching unit under different configurations and types of snapshots.

4.2 Microbenchmarking Memory Restoration

We now analyze the two design options for memory prefetching shown in Figure 7 using a dataset of microVM dirty memory snapshots with different sparsities. We create synthetic datasets from the *pillow* snapshot (Section 3) that range from *most scattered*, when each page is separated, to a case with few large contiguous regions of dirty pages. In practice, the pattern depends on the underlying mechanism used to identify snapshot pages and the applications running in the VMs.

Figure 8 shows the restoration time with each of the two restoration mechanisms when the restoration is done in hardware and software. The x-axis shows the sparsity of the dataset. In all cases, the total size of the dataset is 174.5 MB. The sparsity index denotes the number of pages in continuous regions; each region is separated by its neighbor via an empty page, which is not included in the snapshot. For instance, in sparsity 1, each individual page is separated. We use a single IAA engine in all experiments. As the top figure shows, the scattered memory prefetching design outperforms single-chunk prefetching for sparsities of more than 4 pages. This is because this design avoids additional page copying during the installation phase via userfaultfd. However, when memory partitions become as sparse as every page or two pages, the overhead of scattered DMA and suboptimal PRS handling make scattered prefetching slower than single-chunk. Given this, Sabre uses different memory prefetching strategies depending on the sparsity of the underlying snapshots.

The dashed line in Figure 8 (Top) denotes the time required to restore memory from uncompressed snapshots. We optimize this path similarly to REAP [64], where the whole snapshot is fetched as a single continuous disk read via O_DIRECT file I/O and installed via $userfaultfd$. Sabre's memory restoration overperforms prefetching of uncompressed snapshots by up to 1.9×. Note that the achieved compression ratio on this dataset is 2.2×, meaning that the theoretical upper bound of memory restoration speed-up with respect to uncompressed baselines is also 2.2×. Sabre's memory prefetching is very close to this because it hides decompression behind disk I/Os. For faster disks, these results would still hold.

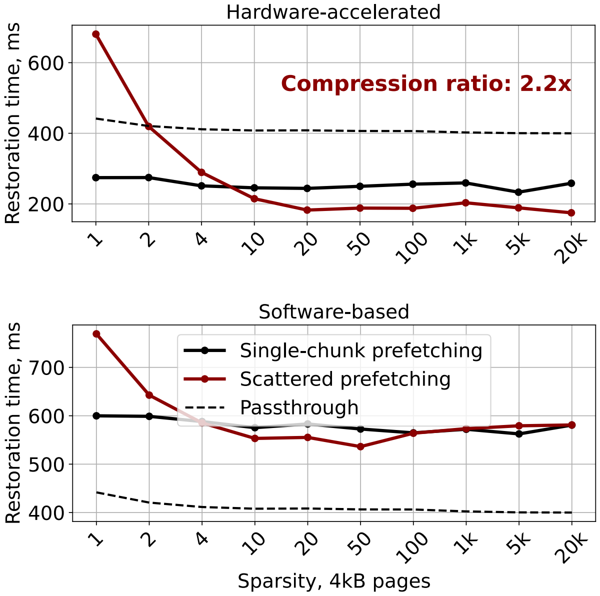


Figure 8: Single-chunk and scattered prefetching across different page sparsities; *passthrough* denotes the time to read uncompressed snapshots; the system runs at 2.3 GHz with a single IAA engine.

Figure 8 (bottom) shows the same results when running Sabre with software-based DEFLATE decompression. Across all different memory sparsities, the overhead of software decompression kills the speed-up of fetching deflated snapshots. This demonstrates that hardware acceleration is required to make snapshot compression practical.

For the sake of completeness, we additionally integrate other *software* compression algorithms optimized for performance in Sabre: *Snappy*, *Zstandard (zstd)*, and *LZ4*. We run them on a dedicated CPU core under the highest possible turbo boost frequency of *4 GHz* and repeat the snapshotting microbenchmark experiment. As Figure 9 shows, across all sparsities, the memory restoration with IAA outperforms these algorithms. The restoration times under *Zstd level-3,10*, and *20* are very close to IAA results and are much lower than prefetching uncompressed snapshots. However, as Figure 9 (Bottom) shows, they require a *significant* amount of CPU resources at the snapshot creation stage (up to several seconds at the highest frequency), and they do not demonstrate performance as high as IAA when running at lower CPU frequencies. We do not show results for *LZ4* as its compression ratios are low.

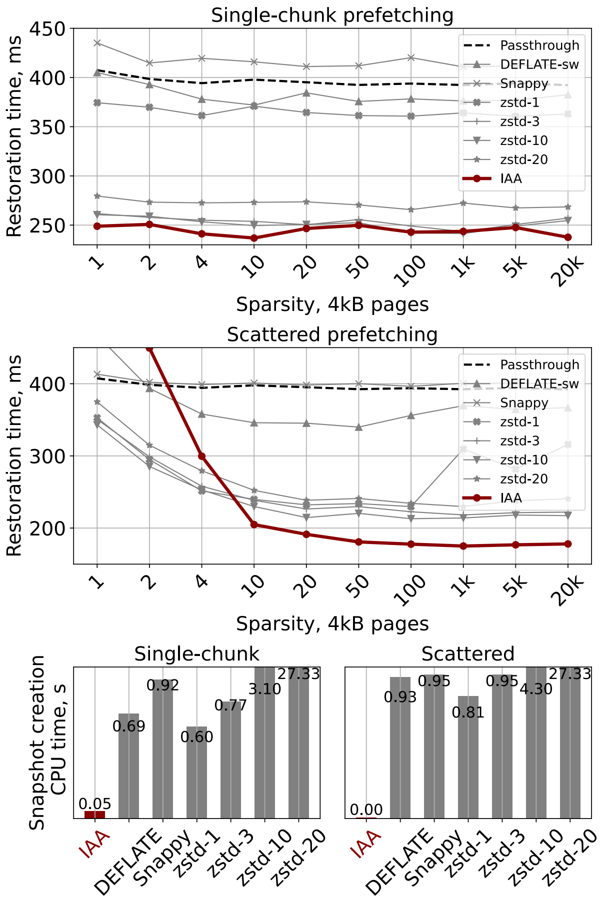


Figure 9: Sabre with different fast *software* (de)compression algorithms running on dedicated CPU cores under the turbo boost frequency of *4 GHz*; the bottom plots show CPU resources (in seconds) consumed to make the snapshots (averaged over sparsities).

4.3 Full System Implementation

We now discuss how we integrate Sabre's memory prefetching unit in an end-to-end serverless framework. We choose Firecracker microVMs [17] as the target serverless sandbox. Firecracker is the current industry-leading VMM, and it already provides good VM snapshotting capabilities. Most recent work in this space, such as REAP [64], is also based on Firecracker. To build the end-to-end serverless pipeline, we partially reuse the infrastructure of vHive – an academic framework for serverless used to showcase the effectiveness of REAP working sets. vHive allows running serverless Docker images inside Firecracker microVMs via *firecracker-containerd*. vHive also extends the native Firecracker Go SDK to support snapshotting and implements a simple orchestrator to simplify managing the serverless environment.

We write Sabre's snapshotting unit in ≈ 3500 LoC in C++17 excluding unit tests and benchmarks, using Intel's QPL library v1.3.1 and build it as a dynamic library. We then integrate it with Firecracker VMM v1.5.0 in only 50 LoC in Rust via FFI. Sabre runs in the default Firecracker's snapshotting/restoration thread, and does not require additional CPU resources. To expose Sabre's snapshotting to the higher layers of serverless frameworks, we additionally extend the Firecracker's Go SDK 1.0.0 with several new APIs. Figure 10 shows the simplified overview of the full system design. Similarly to vHive [64], our infrastructure enables running end-to-end serverless applications in Firecracker microVMs with standard Docker environments.

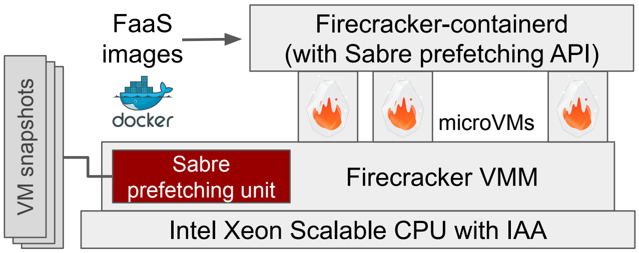


Figure 10: Sabre's integration in end-to-end serverless frameworks.

5 End-to-End Evaluation

Methodology: We evaluate Sabre on a large set of end-to-end serverless benchmarks from vSwarm [66], FunctionBench [44, 45], and SeBS [27]. We modify the benchmarks to run grpc servers and support server reflection so that we can invoke functions using grpcurl. The set includes only one synthetic benchmark – *python-list* implementing traversing a large sparse Python list; we include it to showcase the upper-bound of compression achievable with Sabre. We slightly modify the *dna-visualisation* benchmark from SeBS to make it use different DNA sequence datasets across different invocations (the default benchmark is always based on the same dataset; *dna-visualisation-1*). Similarly, we modify the datasets for the *model training* benchmark to use smaller 2 MB and larger 10 MB images. All other benchmarks are taken from the aforementioned suites. In all experiments, we use a single IAA engine for memory prefetching.

Sabre's memory prefetching unit is agnostic to the underlying mechanism of creating a snapshot. It can be used with dirty page-based snapshots, with working sets, and even when snapshotting the whole guest's VM memory. Since the latter is not practical for realistic serverless workloads, and therefore rarely used in practice, we only evaluate Sabre on the first two options. In all experiments, we use our testbed with the configuration shown in Table 2. We focus our evaluation on two metrics: (1) how well Sabre is able to compress snapshots of different types, and (2) what is the impact of decompression on the end-to-end cold start time.

End-to-end performance impact: Figure 11 shows the result of running the default dirty page-based snapshots of Firecracker (Diff snapshots [2]) with Sabre. Diff snapshotting is enabled via dirty page tracking in the hypervisor. We find Diff snapshots to be relatively large (a few hundred MB)

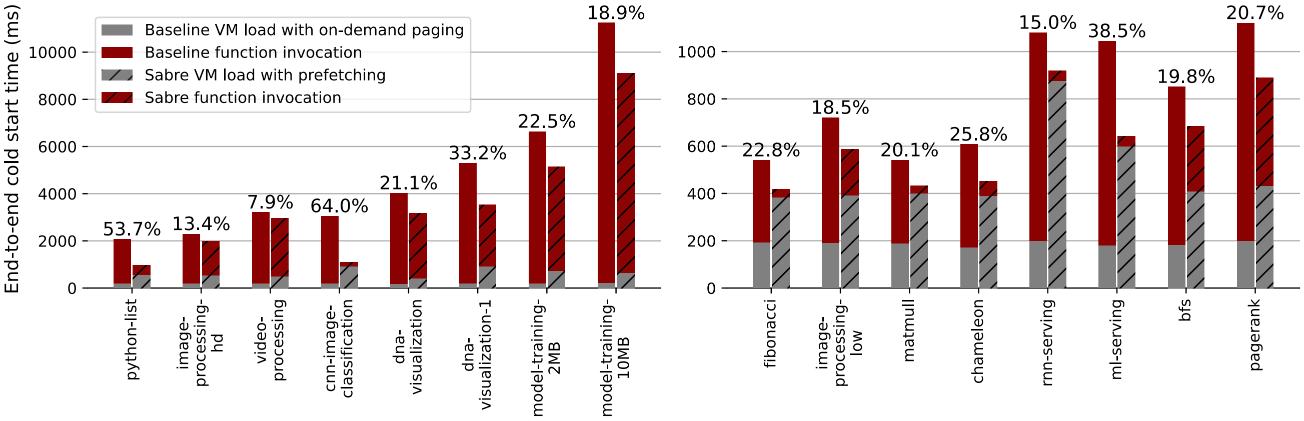


Figure 11: End-to-end evaluation of serverless cold starts with Sabre on Firecracker's default Diff snapshots with prefetching; annotated numbers show speedup of Sabre over the baseline.

and coarse-grained for all applications, which means that the scattered prefetching (Figure 7-C) works best in this case.

Figure 11 compares the end-to-end cold start latency when serving requests from a VM restored with on-demand paging (default mechanism in Firecracker) and via prefetching with Sabre. We find that in all cases, Sabre is able to compress Diff snapshots by up to 4×, and 2.5× on average. This is a significant reduction in storage requirements, given the large original size of Diff snapshots, ranging from hundreds of megabytes to several gigabytes. This is even more significant given that in serverless deployments, a physical node can host hundreds or thousands of snapshotted VMs [17].

Most importantly, in all cases, the hardware-accelerated decompression in Sabre allows restoring a compressed snapshot without any negative impact on the end-to-end latency. Moreover, in some applications, we observe up to 60% lower cold start overhead, enabled by the fast memory prefetching. The speed-up and compression effect are particularly evident for our synthetic *python-list* benchmark, where the dirty memory, i.e., the Python runtime heap storing the list, is well compressible. The same holds for *dna-visualization* as well.

Optimizing the VM snapshotting strategy: While Diff snapshots and dirty page tracking currently represent the industry standard in microVM snapshotting, they are not the most efficient way to restore memory via prefetching. More efficient mechanisms are enabled via working set estimation. We implement working sets in Sabre which are used with our memory prefetching unit for evaluation purposes. Our implementation is based on the record-and-replay technique. Similarly to the original paper [64], Sabre records working sets during the first invocation of serverless functions after the standard restoration from vanilla Firecracker snapshots. We intercept in user-space the guest memory page faults and record all accessed addresses in a vector. The recorder then groups the accessed pages to form continuous chunks, whenever possible, and saves them in a WS file. The REAP restoration uses the *passthrough* functionality of Sabre's memory prefetching unit, which resembles the original REAP specification [64], i.e., direct I/O disk reads combined with page installation in the guest physical memory via *userfaultfd*. After prefetching, the hypervisor continues serving the rest of the pages outside of the working set via standard on-demand paging.

REAP working set files tend to be much more scattered than Firecracker's Diff snapshots. We make a similar observation in our applications as well. Therefore, memory restoration through single-chunk prefetching works best in this case, and we configure Sabre accordingly for REAP snapshots. Table 3 shows the achieved compression ratios of REAP working set files and the corresponding prefetching speedup when using Sabre. Figure 12 shows the end-to-end cold start latencies across the same set of serverless benchmarks as before.

Table 3 shows that working set files are better compressible than dirty pages. The compression ratio reaches up to 4.7×; 3.2× on average. The prefetching time itself with Sabre is accelerated by $25-55\%$. On the synthetic application *python-list*, the speedup reaches 70%. As expected, there is an obvious correlation between the achieved compression ratio and prefetching speedup. End-to-end, working set prefetching speedup translates in up to 20% of improvement in the application's cold start latency. For several of the examined applications, the speedup is diminished due to the working set size being small or their compute time being high. In the latter case, the impact of accelerated restoration gets hidden behind computing. In general, compression of snapshots/working set files is more impactful on applications with larger working sets, especially if they are memory-bounded (as in the case of the *python-list* benchmark). Even when Sabre does not accelerate cold start time significantly, it still greatly reduces the size of the working set files without a negative impact on prefetching latency and/or CPU cycles for restoration.

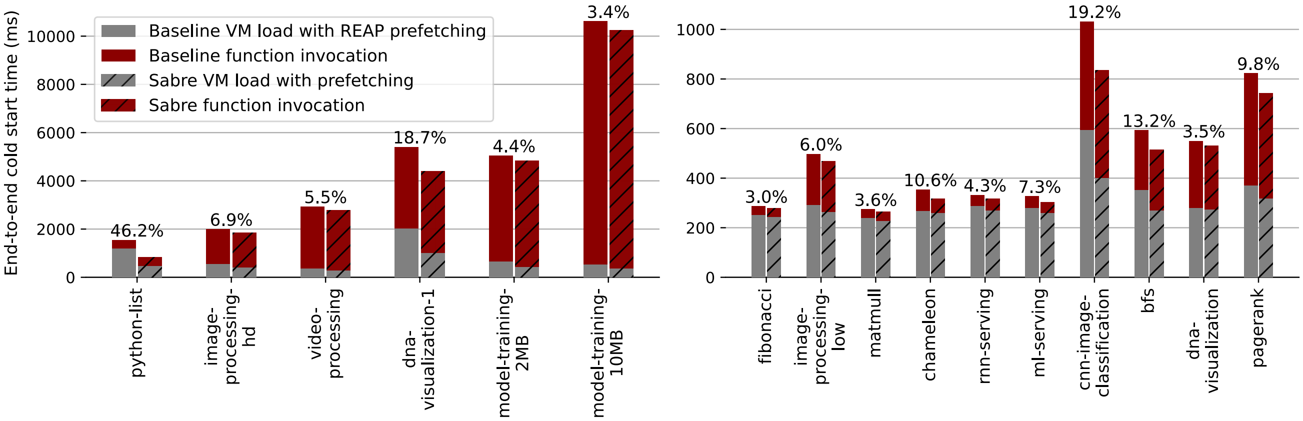


Figure 12: End-to-end evaluation of serverless cold starts with Sabre on REAP snapshots; annotated numbers show the speedup over REAP.

Table 3: Compression ratios and page prefetching speedup of Sabre over REAP working sets.

Application	Size of REAP WS (MB)	Compression ratio	Sabre's prefetching speedup
fibonacci	12.93	2.64×	29.02%
python list	405.23	14.82×	70.81%
image-processing-low	27.67	3.74×	45.17%
image-processing-hd	120.66	2.73×	35.27%
matmull	13.02	2.62×	32.65%
chameleon	18.45	2.90×	36.05%
video-processing	44.91	3.21×	43.30%
rnn-serving	17.64	2.50×	26.59%
ml-serving	22.10	2.67×	35.00%
cnn-image-classification	136.36	3.10×	38.73%
bfs	44.22	4.29×	49.39%
dna-visualization	16.04	2.76×	28.73%
dna-visualization-1	720.95	4.70×	55.01%
pagerank	62.23	2.94×	34.60%
model-training-2MB	171.28	3.60×	43.48%
model-training-10MB	111.71	3.57×	45.54%

6 Discussion and Future Work

6.1 Prefetching on Faster Disks and Networks

While the performance impact of memory prefetching becomes less critical as the speed of disks and NVMe/persistent memory devices increases, Sabre benefits when it comes to storage space reduction remain. This is even more critical, given the higher cost per Byte of new memory technologies. CPU-free memory decompression, especially at zero negative impact on end-to-end latency will always be beneficial, independent of the underlying storage technology. In future work, we plan to evaluate Sabre on CXL-enabled memory devices and explore the potential of serving compressed snapshots from them. The byte-addressable organization of CXL and other similar memory disaggregation devices will make the integration with near-memory accelerators, such as IAA, much more efficient than when using commodity disks, essentially eliminating all overheads of PRS discussed in Section 3.

Additionally, the streaming nature of hardware accelerators, such as IAA, allows combining Sabre with any streaming I/O, including networking. This makes Sabre attractive for *remote snapshotting* – another technology in serverless microVMs where snapshots and/or working set files are served from centralized storage or a remote server. Fast streaming decompression can have a dramatic reduction of network bandwidth consumed for snapshotting, which is critical for highly multi-tenant and geographically distributed datacenters.

6.2 Further Optimizing Sabre

The biggest limitation in our current design of Sabre is its integration with the disk via the standard IAA's PRS mechanism. This disallows bypassing the OS page cache when feeding an input to IAA, and results in lower effective bandwidth utilization than direct I/O. This can be optimized by redesigning the default PRS in one of two ways.

First, PRS can be replaced with user-space page fault handling for input buffers. Disk I/O can be initiated separately in user space using the *read* system call combined with *O_DIRECT* file opening. In this case, the IAA page fault handler only needs to wait until the disk DMA catches up with the sequential data transfer. This requires enabling the *block-on-fault* feature of IAA, otherwise, the input stream would need to be resubmitted from the beginning every time IAA reaches pages that have not been fetched yet. Alternatively, one can directly connect the disk's and IAA's DMA engines, and allow streaming of compressed inputs directly into the accelerator through a small FIFO buffer. However, this can only be done in the host kernel in custom IAA drivers and is challenging to implement in a scalable way.

A second limitation in the current design's single-chunk prefetching (Figure 7-B) is using the COPY-based `userfaultfd` mechanism. The original REAP snapshots [64] suffer from the same inefficiency. Starting with Linux kernel 5.13, `userfaultfd` can be handled via minor page faults and *UFFDIO_CONTINUE* page installation. Instead of copying pages from the continuous buffers, one can install them from the page cache via the underlying page table modification with zero-copy. This will, however, complicate managing the decompression/prefetching buffers to ensure they are never reclaimed, while the corresponding microVM is running.

6.3 Beyond MicroVM Snapshotting

VM memory compression and fast restoration go well beyond microVM snapshotting and serverless. For example, VM live migration [18,59] can benefit from hardware-accelerated compression and on-the-fly decompression of memory pages. The recent work [14] is exploring this opportunity for KVM. This can dramatically accelerate applications heavily relying on VM migration, including VM bin-packing for cloud management [55], low-latency cloud-native applications such as vRAN [49, 68], and fast fault-tolerance solutions [63]. We plan to extend Sabre to benefit these applications as well.

7 Conclusion

MicroVM snapshotting and restoration via page prefetching is the most effective technique for reducing the cold start overhead in serverless. Memory compression is a promising technique to reduce the size of VM snapshots and speed up prefetching during memory restoration, but it is only efficient if decompression is fast. We showed that emerging hardware accelerators for *general-purpose* compression are suitable for microVM memory restoration as well. We first characterized the Intel IAA accelerator and then designed Sabre, a system for fast prefetching of VM memory from compressed snapshots. We showed that Sabre compresses snapshots of real serverless applications up to 4.5×, and speeds up prefetching by up to 55% compared to uncompressed baselines. This results in up to 20% of end-to-end performance improvement for cold function invocations over the most optimized snapshotting technologies.

Acknowledgements

We sincerely thank our shepherd, the Intel IAA team, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported in part by NSF CAREER Award CCF-2326182, a Sloan Research Fellowship, an Intel Research Award, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] In-memory benchmarks for compression alogorithms. https://github.com/inikep/lzbench.

[2] Firecracker snapshotting documentation. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md, 2023.

[3] Google gvisor. https://gvisor.dev/docs/, 2023.

[4] Intel data streaming accelerator (dsa), architecture. https://www.intel.com/content/www/us/en/content-details/759709, 2023.

[5] Intel in-memory analytic accelerator (iaa), architecture specification. https://www.intel.com/content/www/us/en/content-details/721858, 2023.

[6] Intel in-memory analytic accelerator (iaa), use guide. https://www.intel.com/content/www/us/en/content-details/780887, 2023.

[7] Intel query processing library (qpl). https://www.intel.com/content/www/us/en/developer/tools/query-processing-library/overview.html, 2023.

[8] Introducing intel scalable i/o virtualization. https://www.intel.com/content/www/us/en/developer/articles/technical/introducing-intel-scalable-io-virtualization.html, 2023.

[9] Linux kernel patch to enable enqcmd. https://lore.kernel.org/lkml/20210920192349.2602141-1-fenghua.yu@intel.com/T/#rd6d542091da1d1159eda0a44a16e57d0c0dfb209, 2023.

[10] The operstack foundation. kata containers - the speed of containers, the security of vms. https://katacontainers.io/, 2023.

[11] Shared virtual addressing (sva) with enqcmd. https://docs.kernel.org/next/x86/sva.html, 2023.

[12] Technical overview of intel 4th gen xeon scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.0466t8, 2023.

[13] Implementation of sabre. https://github.com/barabanshek/sabre, 2024.

[14] Kvm live migration with iaa in-memory compression. https://lore.kernel.org/all/20240319164527.1873891-1-yuan1.liu@intel.com/T/, 2024.

[15] Sabre iaa benchmarks. https://github.com/barabanshek/IAA_benchmarking.git, 2024.

[16] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. Data compression accelerator on ibm power9 and z15 processors : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2020.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[18] Mahdi Aiash, Glenford Mapp, and Orhan Gemikonakli. Secure live virtual machines migration: Issues and solutions. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 160–165, 2014.

[19] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: A study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Amine Barrak, Fabio Petrillo, and Fehmi Jaafar. Serverless on machine learning: A systematic mapping study. *IEEE Access*, 10:99337–99352, 2022.

[22] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[23] Jacob Breiholz, Farah Yahya, Christopher J. Lukas, Xing Chen, Kevin Leach, David Wentzloff, and Benton H. Calhoun. A 4.4 nw lossless sensor data compression accelerator for 2.9x system power reduction in wireless body sensors. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1041–1044, 2017.

[24] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. Fpga acceleration of zstd compression algorithm. pages 188–191, 06 2021.

[26] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. Fpga acceleration of zstd compression algorithm. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 188–191, 2021.

[27] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[29] Sebastian Deorowicz. Silesia compression corpus. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia, 2023.

[30] P. Deutsch. Rfc1951: Deflate compressed data format specification version 1.3, 1996.

[31] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, IDEAS '15, page 202–203, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[34] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, 2015.

[35] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu scientific & technical journal*, 47, 07 2011.

[38] Junyeong Han, Sungeun Kim, Sungyoung Lee, Jaehwan Lee, and Sung Jo Kim. A hybrid swapping scheme based on per-process reclaim for performance improvement of android smartphones (august 2018). *IEEE Access*, 6:56099–56108, 2018.

[39] Seth Jennings. The zswap compressed swap cache. https://lwn.net/Articles/537422/, 2023.

[40] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Hoyong Jin, Donghun Jeong, Taewon Park, Jong Hwan Ko, and Jungrae Kim. Multi-prediction compression: An efficient and scalable memory compression framework for gp-gpu. *IEEE Computer Architecture Letters*, 21(2):37–40, 2022.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015.

[43] Sagar Karandikar, Aniruddha N. Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. Cdpu: Co-designing compression and decompression processing units for hyperscale systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[45] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[47] Kamil Kojs. A survey of serverless machine learning model inference, 2023.

[48] Reese Kuper, Ipoom Jeong, Yifan Yuan, Jiayu Hu, Ren Wang, Narayan Ranganathan, and Nam Sung Kim. A quantitative analysis and guideline of data streaming accelerator in intel 4th gen xeon scalable processors, 2023.

[49] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient baseband processing in virtualized rans with slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 654–667, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. High-throughput fpga-based hardware accelerators for deflate compression and decompression using high-level synthesis. *IEEE Access*, 8:62207–62217, 2020.

[51] Sang Muk Lee, Jung Hwan Oh, Ji Hoon Jang, Seong Mo Lee, Ji Kwang Kim, and Seung Eun Lee. Live demonstration: An fpga based hardware compression accelerator for hadoop system. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 744–745, 2016.

[52] Weigang Li. Optimize genomics data compression with hardware accelerator. In *2017 Data Compression Conference (DCC)*, pages 446–446, 2017.

[53] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[55] Khine Moe Nwe, Mi Khine Oo, and Maung Maung Htay. Efficient resource management for virtual machine allocation in cloud data centers. In *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pages 419–420, 2018.

[56] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 377–388, New York, NY, USA, 2012. Association for Computing Machinery.

[57] Weikang Qiao, Jieqiong Du, Zhenman Fang, Libo Wang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. pages 291–291, 05 2018.

[58] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012.

[59] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *SIGPLAN Not.*, 53(3):45–56, mar 2018.

[60] Sudhir Satpathy, Vikram Suresh, Raghavan Kumar, Vinodh Gopal, James Guilford, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Ram Krishnamurthy, Vivek De, and Sanu Mathew. A 1.4ghz 20.5gbps gzip decompression accelerator in 14nm cmos featuring dual-path out-of-order speculative huffman decoder and multi-write enabled register file array. In *2019 Symposium on VLSI Circuits*, pages C238–C239, 2019.

[61] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 733–750. Association for Computing Machinery, 2020.

[62] Rajesh Thallam. Bigquery explained: An overview of bigquery's architecture. `https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview`, 2023.

[63] Wen-Hsiu Tsai, Po-Jui Tsao, and Che-Rung Lee. Fvmm: Fast vm migration for virtualization-based fault tolerance using templates. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16, 2022.

[64] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.

[66] vhive-serverless. vswarm: Serverless framework for multi-tenant serverless computing. `https://github.com/vhive-serverless/vSwarm`, 2023. Accessed: Dec 6, 2023.

[67] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[68] Jiarong Xing, Junzhi Gong, Xenofon Foukas, Anuj Kalia, Daehyeok Kim, and Manikanta Kotaru. *Enabling Resilience in Virtualized RANs with Atlas*. Association for Computing Machinery, New York, NY, USA, 2023.

[69] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[70] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2017.

[71] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. *SIGPLAN Not.*, 46(7):87–98, mar 2011.

[72] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, page 87–98, New York, NY, USA, 2011. Association for Computing Machinery.

[73] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.

NOMAD: Non-Exclusive Memory Tiering via Transactional Page Migration

Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan[†], Ren Wang[†]
The University of Texas at Arlington, [†]Intel Labs

Abstract

With the advent of byte-addressable memory devices, such as CXL memory, persistent memory, and storage-class memory, tiered memory systems have become a reality. Page migration is the *de facto* method within operating systems for managing tiered memory. It aims to bring hot data whenever possible into fast memory to optimize the performance of data accesses while using slow memory to accommodate data spilled from fast memory. While the existing research has demonstrated the effectiveness of various optimizations on page migration, it falls short of addressing a fundamental question: Is exclusive memory tiering, in which a page is either present in fast memory or slow memory, but not both simultaneously, the optimal strategy for tiered memory management?

We demonstrate that page migration-based exclusive memory tiering suffers significant performance degradation when fast memory is under pressure. In this paper, we propose *non-exclusive* memory tiering, a page management strategy that retains a copy of pages recently promoted from slow memory to fast memory to mitigate memory thrashing. To enable non-exclusive memory tiering, we develop NOMAD, a new page management mechanism for Linux that features *transactional page migration* and *page shadowing*. NOMAD helps remove page migration off the critical path of program execution and makes migration completely asynchronous. Evaluations with carefully crafted micro-benchmarks and real-world applications show that NOMAD is able to achieve up to 6x performance improvement over the state-of-the-art transparent page placement (TPP) approach in Linux when under memory pressure. We also compare NOMAD with a recently proposed hardware-assisted, access sampling-based page migration approach and demonstrate NOMAD's strengths and potential weaknesses in various scenarios.

1 Introduction

As new memory devices, such as high bandwidth memory (HBM) [4, 30], DRAM, persistent memory [7, 39], Compute Express Link (CXL)-based memory [1, 37, 44], and storage-class memory [53, 55] continue to emerge, future computer systems are anticipated to feature multiple tiers of memory with distinct characteristics, such as speed, size, power, and cost. Tiered memory management aims to leverage the strength of each memory tier to optimize the overall data access latency and bandwidth. Central to tiered memory management is *page management* within operating systems (OS), including page allocation, placement, and migration. Efficient page management in the OS is crucial for optimizing memory utilization and performance while maintaining transparency for user applications.

Traditionally, the memory hierarchy consists of storage media with at least one order of magnitude difference in performance. For example, in the two-level memory hierarchy assumed by commercial operating systems for decades, DRAM and disks differ in latency, bandwidth, and capacity by 2-3 orders of magnitude. Therefore, the sole goal of page management is to keep hot pages in, and maximize the hit rate of the "performance" tier (DRAM), and migrate (evict) cold pages to the "capacity" tier (disk) when needed. As new memory devices emerge, the performance gap in the memory hierarchy narrows. Evaluations on Intel's Optane persistent memory [56] and CXL memory [50] reveal that these new memory technologies can achieve comparable performance to DRAM in both latency and bandwidth, within a range of 2-3x. As a result, the assumption of the performance gap, which has guided the design of OS page management for decades, may not hold. It is no longer beneficial to promote a hot page to the performance tier if the migration cost is too high.

Furthermore, unlike disks which must be accessed through the file system as a block device, new memory devices are byte-addressable and can be directly accessed by the processor via ordinary `load` and `store` instructions. Therefore, for a warm page on the capacity tier, accessing the page directly and avoiding migration to the performance tier could be a better option. Most importantly, while the performance of tiered memory remains hierarchical, the hardware is no longer hierarchical. Both the Optane persistent memory and

CXL memory appear to the processor as a CPUless memory node and thus can be used by the OS as ordinary DRAM.

These unique challenges facing emerging tiered memory systems have inspired research on improving page management in the OS. Much focus has been on expediting page migrations between memory tiers. Nimble [54] improves page migration by utilizing transparent huge pages (THP), multi-threaded migration of a page, and concurrent migration of multiple pages. Transparent page placement (TPP) [44] extends the existing NUMA balancing scheme in Linux to support asynchronous page demotion and synchronous page promotion between fast and slow memory. Memtis [37] and TMTS [24] use hardware performance counters to mitigate the overhead of page access tracking and use background threads to periodically and asynchronously promote pages.

However, these approaches have two fundamental limitations. *First*, the existing page management for tiered memory assumes that memory tiers are *exclusive* to each other – hot pages are allocated or migrated to the performance tier while cold pages are demoted to the capacity tier. Therefore, each page is only present in one tier. As memory tiering seeks to explore the tradeoff between performance and capacity, the working set size of workloads that benefit most from tiered memory systems likely exceeds the capacity of the performance tier. Exclusive memory tiering inevitably leads to excessive hot-cold page swapping or memory thrashing when the performance tier is not large enough to hold hot data.

Second, there is a lack of an efficient page migration mechanism to support tiered memory management. As future memory tiers are expected to be addressable by the CPU, page migrations are similar to serving minor page faults and involve three steps: 1) unmap a page from the page table; 2) copy the page content to a different tier; 3) remap the page on the page table, pointing to the new memory address. Regardless of whether page migration is done synchronously upon accessing a hot page in the slower capacity tier or asynchronously in the background, the 3-step migration process is expensive. During migration, an unmapped page cannot be accessed by user programs. If page migration is done frequently, e.g., due to memory thrashing, user-perceived bandwidth, including accesses to the migrating pages, is significantly lower (up to 95% lower) than the peak memory bandwidth [54].

This paper advocates *non-exclusive* memory tiering that allows a subset of pages on the performance tier to have shadow copies on the capacity tier [1]. Note that non-exclusive tiering is different from *inclusive* tiering which strictly uses the performance tier as a cache of the capacity tier. The most important benefit is that under memory pressure, page demotion is made less expensive by simply remapping a page if it is not dirty and its shadow copy exists on the capacity tier. This allows for smooth performance transition when memory demand exceeds the capacity of the performance tier.

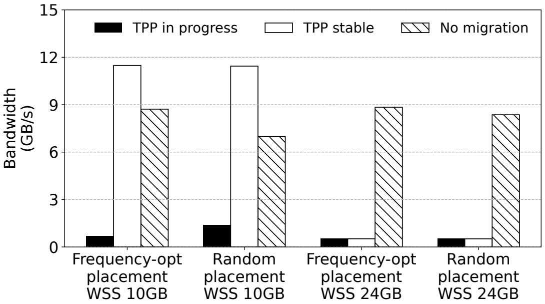


Figure 1: The comparison of achieved memory bandwidth in a micro-benchmark due to different phases in TPP and a baseline approach that disables page migration. Higher is better performance.

To reduce the cost of page migration, especially for promotion, this paper proposes *transactional page migration* (TPM), a novel mechanism to enable page access during migration. Unlike current page migrations, TPM starts page content copy without unmapping the page from the capacity tier so that the migrating page is still accessible by user programs. After page content is copied to a new page on the performance tier, TPM checks whether the page has been dirtied during the migration. If so, the page migration (i.e., the transaction) is invalidated and the copied page is discarded. Failed page migrations will be retried at a later time. If successful, the copied new page is mapped in the page table and the old page is unmapped, becoming a shadow copy of the new page.

We have developed NOMAD, a new page management framework for tired memory that integrates non-exclusive memory tiering and transactional page migration. NOMAD safeguards page allocation to prevent out-of-memory (OOM) errors due to page shadowing. When the capacity tier is under memory pressure, NOMAD prioritizes the reclamation of shadow pages before evicting ordinary pages. We have implemented a prototype of NOMAD in Linux and performed a thorough evaluation on four different platforms, including an FPGA-based CXL prototype, a persistent memory system, and a pre-market, commercial CXL system. Experimental results show that, compared to two representative page management schemes: TPP and Memtis, NOMAD achieves up to 6x performance improvement over TPP during memory thrashing and consistently outperforms Memtis by as much as 130% when the working set size fits into fast memory.

2 Motivation and Related Work

We introduce the background of page management in tiered memory systems and use TPP [44], a state-of-the-art page placement system designed for CXL-enabled tiered memory, as a motivating example to highlight the main limitations of current page management approaches.

[1] We assume that page migrations only occur between two adjacent tiers if there are more than two memory tiers.

2.1 Memory Tiering

Caching and tiering are two traditional software approaches to manage a memory, or storage hierarchy, consisting of various types of storage media (e.g., CPU caches, DRAM, and hard disks) differing in performance, capacity, and cost. Without loss of generality, we consider a two-level memory hierarchy with 1) a *performance* tier (i.e., the fast tier), backed with smaller, faster, but more expensive storage media; and 2) a *capacity* tier (i.e., the slow tier) with larger, slower, and cheaper storage media. For *caching*, data is stored in the capacity tier, and copies of frequently accessed or "hot" data are strategically replicated to the performance tier. For *tiering*, new data is first allocated to the performance tier and remains there if it is frequently accessed, while less accessed data may be relegated to the capacity tier when needed. At any moment, data resides exclusively in one of the tiers but not both. Essentially, caching operates in an *inclusive* page placement mode and retains pages in their original locations, only temporarily storing a copy in the performance tier for fast access. Conversely, tiering operates in an *exclusive mode*, actively relocating pages across various memory/storage mediums.

Diverse memory/storage devices, such as high bandwidth memory (HBM) [4], CXL-based memory [1], persistent memory (PM) [7], and fast, byte-addressable NVMe SSDs [31], have emerged recently. While they still make a tradeoff between speed, size, and cost, the gap between their performance narrows. For example, Intel Optane DC persistent memory (PM), available in a DIMM package on the memory bus enabling programs to directly access data from the CPU using `load` and `store` instructions, provides (almost) an order of magnitude higher capacity than DRAM (e.g., 8x) and offers performance within a range of 2-3x of DRAM, e.g., write latency as low as 80 *ns* and read latency around 170 *ns* [56]. More recently, compute express link (CXL), an open-standard interconnect technology based on PCI Express (PCIe) [1], provides a memory-like, byte-addressable interface (i.e., via the `CXL.mem` protocol) for connecting diverse memory devices (e.g., DRAM, PM, GPUs, and smartNICs). Real-world CXL memory offers comparable memory access latency (<2x) and throughput ($\sim$50%) to ordinary DRAM [50].

From the perspective of OS memory management, CXL memory or PM appears to be a remote, CPUless memory node, similar to a multi-socket non-uniform memory access (NUMA) node. State-of-the-art tiered memory systems, such as TPP [44], Memtis [37], Nimble [54], and AutoTiering [32], all adopt *tiering* to *exclusively* manage data on different memory tiers. Unlike the traditional two-level memory hierarchy involving DRAM and disks, in which DRAM acts as a cache for the much larger storage tier, current CXL memory tiering treats CXL memory as an extension of local DRAM. While exclusive memory tiering avoids data redundancy, it necessitates data movement between memory tiers to optimize the performance of data access, i.e., promoting hot data to the fast tier and demoting cold data to the slow tier. Given that all memory tiers are byte-addressable by the CPU and the performance gap between tiers narrows, it remains to be seen whether exclusive tiering is the optimal strategy considering the cost of data movement.

We evaluate the performance of transparent page placement (TPP) [44], a state-of-the-art and the default tiered memory management in Linux. Figure 1 shows the bandwidth of a micro-benchmark that accesses a configurable working set size (WSS) following a Zipfian distribution in a CXL-based tiered memory system. More details of the benchmark and the hardware configurations can be found in Section 4. We compare the performance of TPP while it actively migrates pages between tiers for promotion and demotion (denoted as *TPP in progress*) and when it has finished page relocation (*TPP stable*) with that of a baseline that disables page migration (*no migration*). The baseline does not optimize page placement and directly accesses hot pages from the slow tier. The tiered memory testbed is configured with 16GB fast memory (local DRAM) and 16GB slow memory (remote CXL memory). We vary the WSS to fit in (e.g., 10GB) and exceed (e.g., 24GB) fast memory capacity. Note that the latter requires continuous page migrations between tiers since hot data spills into slow memory. Additionally, we explore two initial data placement strategies in the benchmark. First, the benchmark pre-allocates 10GB of data in fast memory to emulate the existing memory usage from other applications. *Frequency-opt* is an allocation strategy that places pages according to the descending order of their access frequencies (hotness). Thus, the hottest pages are initially placed in fast memory until the WSS spills into slow memory. In contrast, *Random* employs a random allocation policy and may place cold pages initially in fast memory.

We have important observations from results in Figure 1. First, page migration in TPP incurs significant degradation in application performance. When WSS fits in fast memory, *TPP stable*, which has successfully migrated all hot pages to fast memory, achieves more than an order of magnitude higher bandwidth than *TPP in progress*. Most importantly, *no migration* is consistently and substantially better than *TPP in progress*, suggesting that the overhead of page migration outweighs its benefit until the migration is completed. Second, TPP never reaches a stable state and enters memory thrashing when WSS is larger than the capacity of fast memory. Third, page migration is crucial to achieving optimal performance if it is possible to move all hot data to fast memory and the initial placement is sub-optimal, as evidenced by the wide gap between *TPP stable* and *no migration* in the 10GB WSS and random placement test.

2.2 Page Management

In this section, we delve into the design of page management in Linux and analyze its overhead during page migration. We focus our discussions on 1) how to effectively track memory accesses and identify hot pages, and 2) the mechanism to migrate a page between memory tiers.

Tracking memory access can be conducted by software (via the kernel) and/or with hardware assistance. Specifically, the kernel can keep track of page accesses via page faults [2, 32, 44], scanning page tables [2, 14, 19, 43, 54], or both. Capturing each memory access for precise tracking can be expensive. Page fault-based tracking traps memory accesses to selected pages (i.e., whose page table entry permissions are set to `no access`) via hint (minor) page faults. Thus, it allows the kernel to accurately measure the *recency* and *frequency* of these pages. However, invoking a page fault on every memory access incurs high overhead on the critical path of program execution. On the other hand, page table (PT) scanning periodically checks the `access` bit in all page table entries (PTE) to determine recently accessed pages since the last scanning. Compared to page fault-based tracking, which tracks every access on selected pages, PT scanning has to make a tradeoff between scanning overhead and tracking accuracy by choosing an appropriate scanning interval [37].

Linux adopts a *lazy* PT scanning mechanism to track hot pages, which lays the foundation for its tiered memory management. Linux maintains two LRU lists for a memory node: an *active* list to store hot pages and an *inactive* list for cold pages. By default, all new pages go to the inactive list and will be promoted to the active list according to two flags, `PG_referenced` and `PG_active`, in the per-page struct `page`. `PG_reference` is set when the `access` bit in the corresponding PTE is set upon a PTE check and `PG_active` is set after `PG_reference` is set for two consecutive times. A page is promoted to the active list when its `PG_active` flag is set. For file-backed pages, their accesses are handled by the OS through the file system interface, e.g., `read()` and `write()`. Therefore, their two flags are updated each time they are accessed. For anonymous pages, e.g., application memory allocated through `malloc`, since page accesses are directly handled by the MMU hardware and bypass the OS kernel, the updates to their reference flags and LRU list management are only performed during memory reclamation. Under memory pressure, the swapping daemon `kswapd` scans the inactive list and the corresponding PTEs to update inactive pages' flags, and reclaims/swaps out those with `PG_reference` unset. Additionally, `kswapd` promotes hot pages (i.e., those with `PG_active` set) to the active list. This lazy scanning mechanism delays access tracking until it is necessary to reduce the tracking overhead, but undermines tracking accuracy.

TPP [44] leverages Linux's PT scanning to track hot pages and employs page fault-based tracking to decide whether to promote pages from slow memory. Specifically, TPP sets all pages residing in slow memory (e.g., CXL memory) as `inaccessible`, and any user access to these pages will trigger a minor page fault, during which TPP decides whether to promote the faulting page. If the faulting page is on the active list, it is migrated (promoted) to the fast tier. Page demotion occurs when fast memory is under pressure and `kswapd` migrates pages from the inactive list to slow memory.

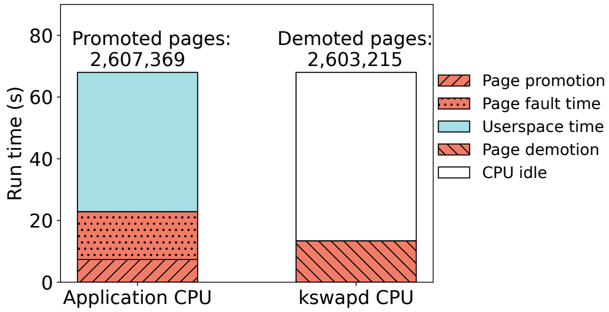


Figure 2: Time breakdown in the execution of *TPP in progress*: Synchronous page migration and page fault handling account for a significant portion of the runtime.

Accurate and lightweight memory access tracking can be achieved with hardware support, e.g., by adding a PTE *count field* in hardware that records the number of memory accesses [45]. However, hardware-based tracking can increase the complexity and require extensive hardware changes in mainstream architectures (e.g., x86). In practice, the hardware-assisted sampling, such as via Processor Event-Based Sampling (PEBS) [24, 37] on Intel platforms, has been employed to record page access (virtual address) information from sampled hardware events (e.g., LLC misses or store instructions). However, PEBS-based profiling also requires a careful balance between the frequency of sampling and the accuracy of profiling. We observed that the PEBS-based approach [37], with a sampling rate optimized for minimizing overhead, remains coarse-grained and fails to capture many hot pages. Further, the sampling-based approach may not accurately measure access recency, thus limiting its ability to make timely migration decisions.

Page migration between memory tiers involves a complex procedure: ① The system must trap to the kernel (e.g., via page faults) to handle migration; ② The PTE of a migrating page must be locked to prevent others from accessing the page during migration and be `ummapped` from the page table; ③ A translation lookaside buffer (TLB) shootdown must be issued to each processor (via inter-processor interrupts (IPIs)) that may have cached copies of the stale PTE; ④ The content of the page is copied between tiers; ⑤ Finally, the PTE must be `remapped` to point to the new location. Page migration can be done *synchronously* or *asynchronously*. Synchronous migration, e.g., page promotion in TPP, is on-demand triggered by user access to a page and on the critical path of program execution. During migration, the user program is blocked until migration is completed. Asynchronous migration, e.g., page demotion in TPP, is handled by a kernel thread (i.e., `kswapd`), oftentimes off programs' critical path, when certain criteria are met. Synchronous migration is costly not only because pages are inaccessible during migration but also may involve a large number of page faults.

Figure 2 shows the run time breakdown of the aforementioned benchmark while TPP is actively relocating pages between the two memory tiers. Since page promotion is synchronous, page fault handling and page content copying (i.e., promotion) are executed on the same CPU as the application thread. Page demotion is done through `kswapd` and uses a different core. As shown in Figure 2, synchronous promotion together with page fault handling incurs significant overhead on the application core. In contrast, the demotion core remains largely idle and does not present a bottleneck. As will be discussed in Section 3.1, userspace run time can also be prolonged due to repeated minor page faults (as many as 15) to successfully promote one page. This overhead analysis explains the poor performance of TPP observed in Figure 1.

2.3 Related Work

A long line of pioneering work has explored a wide range of tiered storage/memory systems, built upon SSDs and HDDs [12, 15, 22, 26, 33, 41, 49, 52, 57], DRAM and disks [21, 25, 29, 46], HBM and DRAMs [23, 45, 48], NUMA memory [2, 3], PM and DRAM [13, 14, 40, 43], local and far memory [19, 27, 34, 35, 47], DRAM and CXL memory [37, 38, 44], and multiple tiers [32, 36, 40, 51, 54]. We focus on tiered memory systems consisting of DRAM and the emerging byte-addressable memory devices, e.g., CXL memory and PM. NOMAD also applies to other tiered memory systems such as HBM/DRAM and DRAM/PM.

Lightweight memory access tracking. To mitigate software overhead associated with memory access tracking, Hotbox [19] employs two separate scanners for fast and slow tiers to scan the slow tier at a fixed rate while the fast tier at an adaptive rate, configurable based on the local memory pressure. Memtis [37] adjusts its PEBS-based sampling rate to ensure its overhead is under control (e.g., $< 3\%$). TMTS [24] also adopts a periodic scanning mechanism to detect frequency along with hardware sampling to more timely detect newly hot pages. While these approaches balance scanning/sampling overhead and tracking accuracy, an "always-on" profiling component does not seem practical, especially for high-pressure workloads. Instead, thermostat [14] samples a small fraction of pages, while DAMON [3] monitors memory access at a coarser-grained granularity (i.e., region). Although both can effectively reduce the scanning overhead, coarse granularity leads to lower accuracy regarding page access patterns. On the other hand, to reduce the overhead associated with frequent hint page faults like AutoNUMA [2], TPP [44] enables the page-fault based detection only for CXL memory (i.e., the slow tier) and tries to promote a page promptly via synchronous migration; prompt page promotion avoids subsequent page faults on the same page.

Inspired by existing lightweight tracking systems, such as Linux's active and inactive lists and hint page faults, NOMAD advances them by incorporating more recency information with *no* additional CPU overhead. Unlike hardware-assisted approaches [24, 37, 42], NOMAD does not require any additional hardware support.

Page migration optimizations. To hide reclamation overhead from applications, TPP [44] decouples page allocation and reclamation; however, page migration remains in the critical path, incurring significant slowdowns. Nimble [54] focuses on mitigating page migration overhead with new migration mechanisms, including transparent huge page migration and concurrent multi-page migration. Memtis [37] further moves page migration out of the critical path using a kernel thread to promote/demote pages in the background. TMTS [24] leverage a user/kernel collaborative approach to control page migration. In contrast, NOMAD aims to achieve prompt, on-demand page migration while moving page migration off the critical path. It is orthogonal to and can benefit from existing page migration optimizations. The most related work is [20], which leverages hardware support to pin data in caches, enabling access to pages during migration. Again, NOMAD does not need additional hardware support.

3 NOMAD Design and Implementation

NOMAD is a new page management mechanism for tiered memory that features *non-exclusive memory tiering* and *transactional page migration*. The goal of NOMAD design is to enable the processor to freely access pages from both fast and slow memory tiers and move the cost of page migration off the critical path of users' data access. Note that NOMAD does not make page migration decisions and relies on the existing memory access tracking in the OS to determine page temperature. Furthermore, NOMAD does not impact the initial memory allocation in the OS and assumes a standard page placement policy. Pages are allocated from the fast tier whenever possible and are placed in the slower tier only when there is an insufficient number of free pages in the fast tier, or attempts to reclaim memory in the fast tier have failed. After the initial page placement, NOMAD gradually migrates hot pages to the fast tier and cold pages to the slow tier. NOMAD seeks to address two key issues: 1) *how to minimize the cost of page migration?* 2) *how to minimize the number of migrations?*

Overview. Inspired by multi-level cache management in modern processors, which do not employ a purely inclusive or exclusive caching policy between tiers [16] to facilitate the sharing of or avoid the eviction of certain cache lines, NOMAD embraces a *non-exclusive* memory tiering policy to prevent memory thrashing when under memory pressure. Unlike the existing page management schemes that move pages between tiers and require that a page is only present in one tier, NOMAD instead copies pages from the slow tier to the fast tier and keeps a shadow copy of the migrated pages at the slow tier. The non-exclusive tiering policy maintains shadow copies

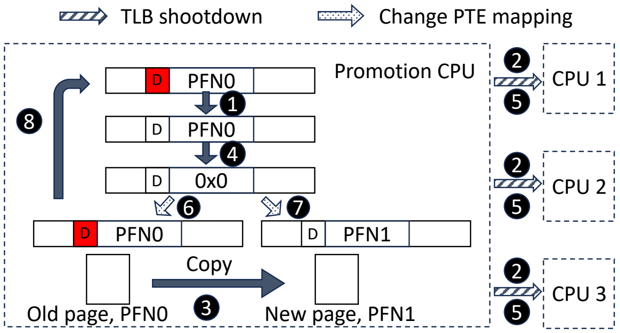


Figure 3: The workflow of transactional page migration. PFN is the page frame number and D is the dirty bit in PTE. The page is only inaccessible by user programs during step 4 when the page is remapped in the page table.

only for pages that have been promoted to the fast tier, thereby not an inclusive policy. The advantage of the non-exclusive policy is that the demotion of clean, cold pages can be simplified to remapping the page table entry (PTE) without the need to copy the cold page to the slower tier.

The building block of NOMAD is a new *transactional page migration* (TPM) mechanism to reduce the cost of page migrations. Unlike the existing unmap-copy-remap 3-step page migration, TPM opportunistically copies a page without unmapping it from the page table. During the page copy, the page is not locked and can be accessed by a user program. After the copy is completed, TPM checks if the page has been dirtied during the copy. If not, TPM locks the page and remaps it in the PTE to the faster tier. Otherwise, the migration is aborted and will be tried at a later time. TPM not only minimizes the duration during which a page is inaccessible but also makes page migration asynchronous, thereby removing it from the critical path of users' data access.

Without loss of generality, we describe NOMAD design in the context of Linux. We start with transactional page migration and then delve into page shadowing – an essential mechanism that enables non-exclusive memory tiering.

3.1 Transactional Page Migration

The motivation to develop TPM is to make page migration entirely asynchronous and decoupled from users' access to the page. As discussed in Section 2.2, the current page migration in Linux is synchronous and on the critical path of users' data access. For example, the default tiered memory management in Linux, TPP, attempts to migrate a page from the slow tier whenever a user program accesses the page. Since the page is in `inaccessible` mode, the access triggers a minor page fault, leading TPP to attempt the migration. The user program is blocked and makes no progress until the minor page fault is handled and the page is remapped to the fast tier, which can be a time-consuming process. Worse, if the migration fails, the OS remains in function `migrate_pages` and retries the aforementioned migration until it is successful or reaching a maximum of 10 attempts.

TPM decouples page migration from the critical path of user programs by making the migrating page accessible during migration. Therefore, users will access the migrating page from the slow tier before the migration is complete. While accessing a hot page from the slow tier may lead to sub-optimal memory performance, it avoids blocking user access due to the migration, thereby leading to superior user-perceived performance. Figure 3 shows the workflow of TPM. Before migration commences, TPM clears the protection bit of the page frame and adds the page to a migration pending queue. Since the page is no longer protected and not yet unmapped from the page table, following accesses to the page will not trigger additional page faults.

TPM starts a migration transaction by clearing the dirty bit of the page (step ❶) and checks the dirty bit after the page is copied to the fast tier to determine whether the transaction was successful. After changing the dirty bit in PTE, TPM issues a TLB shootdown to all cores that ever accessed this page (step ❷). This is to ensure that subsequent writes to the page can be recorded on the PTE. After the TLB shootdown is completed, TPM starts copying the page from the slow tier to the fast tier (step ❸). To commit the transaction, TPM checks the dirty bit by loading the entire PTE using atomic instruction `get_and_clear` (step ❹). Clearing the PTE is equivalent to unmapping the page and thus another TLB shootdown is needed (step ❺). Note that after unmapping the page from PTE, it becomes inaccessible by users. TPM checks whether the page was dirtied during the page copy (step ❻) and either commits the transaction by remapping the page to the fast tier if the page is clean (step ❼) or otherwise aborts the transaction (step ❽). If the migration is aborted, the original PTE is restored and waits for the next time when TPM is rescheduled to retry the migration. The duration in which the page is inaccessible is between ❹ and ❼/ ❽, significantly shorter than that in TPP (possibly multiple attempts between ❶ and ❼).

Page migration is a complex procedure that involves memory tracing and updates to the page table for page remapping. The state-of-the-art page fault-based migration approaches, e.g., TPP in Linux [44], employ synchronous page migration, a mechanism in the Linux kernel for moving pages between NUMA nodes. In addition to the extended migration time affecting the critical path of user programs, this mechanism causes excessive page faults when integrated with the existing LRU-based memory tracing. TPP makes *per-page* migration decisions based on whether the page is on the *active* LRU list. Nevertheless, in Linux, memory tracing adds pages from the inactive to the active LRU list in batches of 15 requests [2], aiming to minimize the queue management overhead. Due to synchronous page migration, TPP may submit multiple requests (up to 15 if the request queue is empty) for a page to

[2] The 15 requests could be repeated requests for promoting the same page to the active LRU list

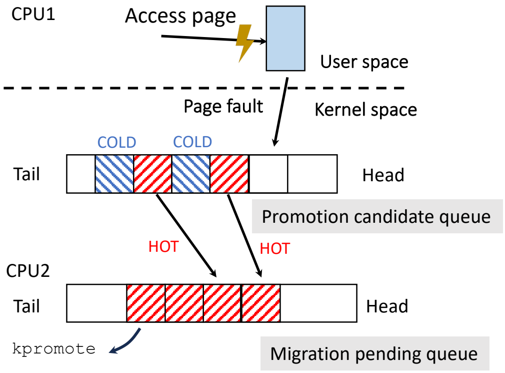


Figure 4: TPM uses a two-queue design to enable asynchronous page migration.

be promoted to the active LRU list to initiate the migration process. In the worst case, migrating one page may generate as many as 15 minor page faults.

TPM provides a mechanism to enable **asynchronous page migration** but requires additional effort to interface with memory tracing in Linux to minimize the number of page faults needed for page migration. As shown in Figure 4, in addition to the inactive and active LRU lists in memory tracing, TPM maintains a separate *promotion candidate* queue (PCQ) for pages that 1) have been tried for migration but 2) not yet promoted to the active LRU list. Upon each time a minor (hint) page fault occurs and the faulting page is added to PCQ, TPM checks if there are any hot pages in PCQ that have both the `active` and `accessed` bits set. These hot pages are then inserted to a *migration pending queue*, from where they will be tried for asynchronous, transactional migration by a background kernel thread `kpromote`. Note that TPM does not change how Linux determines the temperature of a page. For example, in Linux, all pages in the active LRU list, which are eligible for migration, have the two memory tracing bits set. However, not all pages with these bits set are in the active list due to LRU list management. TPM bypasses the LRU list management and provides a more efficient method to initiate page migration. If all transactional migrations were successful, TPM guarantees that only one page fault is needed per migration in the presence of LRU list management.

3.2 Page Shadowing

To enable non-exclusive memory tiering, NOMAD introduces a one-way *page shadowing* mechanism to allow a subset of pages resident in the performance tier to have a shadow copy in the capacity tier. Only pages promoted from the slow tier have shadow copies in the slow tier. Shadow copies are the original pages residing on the slow tier before they are unmapped in the page table and migrated to the fast tier. Shadow pages play a crucial role in minimizing the overhead of page migration during periods of memory pressure. Instead of swapping hot and cold pages between memory tiers, page shadowing enables efficient page demotion through page table remapping. This would eliminate half of the page migration overhead, i.e., page demotion, during memory thrashing.

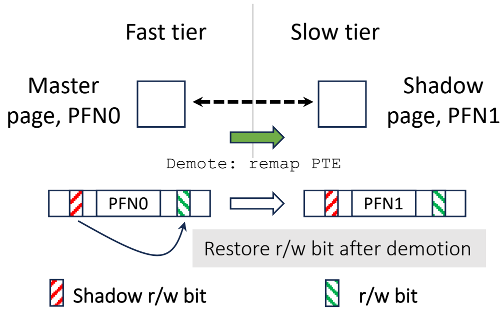


Figure 5: Shadow page management using `shadow r/w` bit.

Indexing shadow pages. Inspired by the indexing of file-based data in the Linux page cache, NOMAD builds an XArray for indexing shadow pages. An XArray is a radix-tree like, cache-efficient data structure that acts as a key-value store, mapping from the physical address of a fast tier page to the physical address of its shadow copy on the slow tier. Upon successfully completing a page migration, NOMAD inserts the addresses of both the new and old pages into the XArray. Additionally, it adds a new `shadow` flag to the `struct page` of the new page, indicating that shadowing is on for this page.

Shadow page management. The purpose of maintaining shadow pages is to assist with page demotion. Fast or efficient page demotion is possible via page remapping if the master page, i.e., the one on the fast tier, is clean and consistent with the shadow copy. Otherwise, the shadow copy should be discarded. To track inconsistency between the master and shadow copies, NOMAD sets the master page as `read-only` and a write to the page causes a page fault. To simplify system design and avoid additional cross-tier traffic, NOMAD discards the shadow page if the master page is dirtied.

However, tracking updates to the master page poses a significant challenge. Page management in Linux relies heavily on the read-write permission to perform various operations on a page, such as copy-on-write (CoW). While setting master pages as `read-only` effectively captures all writes, it may affect how these master pages are managed in the kernel. To address this issue, NOMAD introduces a procedure called *shadow page fault*. It still designates all master pages as `read-only` but preserves the original read-write permission in an unused software bit on the page's PTE (as shown in Figure 5). We refer to this software bit as `shadow r/w`. Upon a write to a master page, a page fault occurs. Unlike an ordinary page fault that handles write violation, the shadow page fault, which is invoked if the page's `shadow` flag is set in its `struct page`, restores the read-write permission of the faulting page according to the `shadow r/w` bit and discards/frees the shadow page. The write may proceed once the shadow page fault returns and reinstates the page to be writable. For `read-only` pages, tracking shadow pages does not impose additional overhead; for writable pages, it requires one additional shadow page fault to restore their write permission.

Reclaiming shadow pages. Non-exclusive memory tiering introduces space overhead due to the storage of shadow pages. If shadow pages are not timely reclaimed when the system is under memory pressure, applications may encounter out-of-memory (OOM) errors, which would not occur under exclusive memory tiering. There are two scenarios in which shadow pages should be reclaimed. First, the Linux kernel periodically checks the availability of free pages and if free memory falls below `low_water_mark`, kernel daemon `kswapd` is invoked to reclaim memory. NOMAD instructs `kswapd` to prioritize the reclamation of shadow pages. Second, upon a memory allocation failure, NOMAD also tries to free shadow pages. To avoid OOM errors, the number of freed shadow pages should exceed the number of requested pages. However, frequent memory allocation failures could negatively affect system performance. NOMAD employs a straightforward heuristic to reclaim shadow pages, targeting 10 times the number of requested pages or until all shadow pages are freed. While excessive reclamation may have a negative impact on NOMAD's performance, it is crucial to prevent Out-of-Memory (OOM) errors. Experiments in Section 4 demonstrate the robustness of NOMAD even under extreme circumstances.

3.3 Limitations

NOMAD relies on two rounds of TLB shootdown to effectively track updates to a migrating page during transactional page migration. When a page is used by multiple processes or mapped by multiple page tables, its migration involves multiple TLB shootdowns, per each mapping, that need to happen simultaneously. The overhead of handling multiple IPIs could outweigh the benefit of asynchronous page copy. Hence, NOMAD deactivates transactional page migration for multi-mapped pages and resorts to the default synchronous page migration mechanism in Linux. As high-latency TLB shootdowns based on IPIs continue to be a performance concern, modern processors, such as ARM, future AMD, and Intel x86 processors, are equipped with ISA extensions for faster broadcast-based [17, 18] or micro-coded RPC-like [28] TLB shootdowns. These emerging lightweight TLB shootdown methods will greatly reduce the overhead of TLB coherence in tiered memory systems with expanded memory capacity. NOMAD will also benefit from the emerging hardware and can be extended to scenarios where more intensive TLB shootdowns are necessary.

4 Evaluation

This section presents a thorough evaluation of NOMAD, focusing on its performance, overhead, and robustness. Our primary goal is to understand tiered memory management by comparing NOMAD with existing representative approaches to reveal the benefits and potential limitations of current page management approaches for emerging tiered memory.

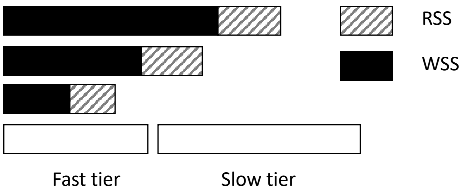


Figure 6: The three memory provisioning schemes used in the evaluation. From bottom to top concerning fast memory: over-provisioning, approaching capacity, and under-provisioning.

We analyze two types of memory footprints: 1) resident set size (RSS) – the total size of memory occupied by a program, and 2) working set size (WSS) – the amount of memory a program actively uses during execution. RSS determines the initial page placement, while WSS dictates the number of pages that should be migrated to the fast tier. Since we focus on in-memory computing, WSS is typically smaller than RSS. Figure 6 illustrates the three scenarios we study with the WSS size smaller than, close to, and larger than fast memory size.

Testbeds. We conducted experiments on *four* platforms with different configurations in CPU, local DRAM, CXL memory, and persistent memory, as detailed in Table 1.

- *Platform A* was built with commercial off-the-shelf (COTS) Intel Sapphire Rapids processors and a 16 GB Agilex-7 FPGA-based CXL memory device [6].
- *Platform B* featured an engineering sample of the Intel Sapphire Rapids processors with the same FPGA-based CXL memory device. The prototype processors have engineering tweaks that have the potential to enhance the performance of CXL memory, which were not available on platform A.
- *Platform C* included an Intel Cascade Lake processor and six 256 GB 100 series Intel Optane Persistent Memory. This platform enabled the full capability of PEBS-based memory tracking and allowed for a comprehensive comparison between page fault- and sampling-based page migration.
- *Platform D* had an AMD Genoa 9634 processor and four 256 GB Micron's (pre-market) CXL memory modules. This platform allowed us to evaluate NOMAD with more realistic CXL memory configurations.

Since the FPGA-based CXL memory device had only 16 GB of memory, we configured local DRAM to 16 GB for all platforms [3]. Note that platform C was equipped with DDR4 DRAM as fast memory while the other platforms used DDR5

[3] Although platform C and D have larger PM or CXL memory sizes, we configured them with 16 GB slow memory consistent with platform A and B for a fair comparison in micro-benchmarks.This limit was lifted when testing real applications.

	Platform A	Platform B (engineering sample)	Platform C	Platform D
CPU	4th Gen Xeon Gold 2.1GHz	4th Gen Xeon Platinum 3.5GHz	2nd Gen Xeon Gold 3.9GHz	AMD Genoa 3.7GHz
Performance tier (DRAM)	16 GB DDR5	16 GB DDR5	16 GB DDR4	16GB DDR5
Capacity tier (CXL or PM Memory)	Agilex 7 16 GB DDR4	Agilex 7 16 GB DDR4	Optane 100 256 GB DDR-T ×6	Micron CXL memory 256GB ×4
Performance tier read latency	316 cycles	226 cycles	249 cycles	391 cycles
Capacity tier read latency	854 cycles	737 cycles	1077 cycles	712 cycles
Performance tier bandwidth (GB/s) Single Thread / Peak performance	Read: 12/31.45 Write: 20.8/28.5	Read: 12/31.2 Write: 22.3/23.67	Read: 12.57/116 Write: 8.67/85	Read: 37.8/270 Write: 89.8/272
Capacity tier bandwidth (GB/s) Single Thread/Peak performance	Read: 4.5/21.7 Write: 20.7/21.3	Read: 4.45/22.3 Write: 22.3/22.4	Read: 4/40.1 Write: 8.1/13.6	Read: 20.25/83.2 Write: 57.7/84.3

Table 1: The configurations of four testbeds and performance characteristics of various memory devices.

Workload Type	In progress Promotion	In progress Demotion	Steady Promotion	Steady demotion
Small WSS	(1.2M|1M)/(15.9K|134K)/ (1.16M|781K)	(2.4M|2.2M)/(15.9K|140K)/ (2.7M|1.5M)	(0|3.3K)/(7.7K|104K)/ (82|74)	(424K|56K)/(0|104K)/ (48K|0)
Medium WSS	(4M|6M)/(0|0)/ (1.6M|5M)	(4.7M|6M)/(2|512)/ (2.5M|4.8M)	(1.8M|3.2M)/(17.4K|0)/ (417K|1.6M)	(1.9M|3.2M)/(16.9K|0)/ (293K|1.4M)
Large WSS	(7M|5.9M)/(0|0)/ (4.5M|7M)	(7.2M|6.5M)/(0|15)/ (4.1M|7.2M)	(7.1M|5.2M)/(0|143K)/ (6.8M|8.8M)	(7.1M|5.3M)/(0|143K)/ (6.8M|8.9M)

Table 2: The number of page promotions/demotions for read|write during the *migration in progress* and the *stable* phases for TPP/Memtis-Default/NOMAD. The data corresponds to Figure 7 for platform A.

DRAM. We evaluated both CXL memory and persistent memory (PM) as slow memory. Table 1 lists the performance characteristics of the four platforms for single-threaded and peak (multi-threaded) performance. While CXL memory and PM have distinct characteristics, including persistence, concurrent performance, and read/write asymmetry, they achieve comparable performance within 2-3x of DRAM and provide a similar programming interface as a CPUless memory node. To ensure a fair comparison, we only enabled one socket on each of the four platforms. Intel platforms were configured with 32 cores while the AMD platform had 84 cores.

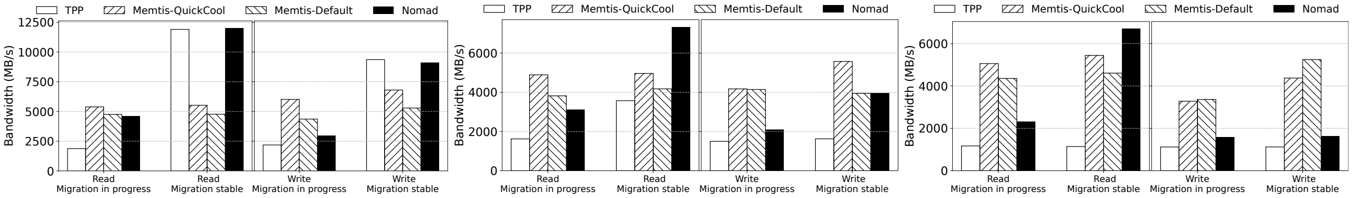
Baselines for comparison. We compared NOMAD with two state-of-the-art tired memory systems: TPP [44] and Memtis [37]. We evaluated both TPP and Nomad on Linux kernel v5.13-rc6 and ran Memtis on kernel v5.15.19, the kernel version upon which Memtis was built and released. We tested two versions of Memtis – Memtis-Default and Memtis-QuickCool – with different data cooling speeds (i.e., the number of samples collected before halving a page's access count). Specifically, Memtis-Default used the default cooling period of 2,000k samples, while Memtis-QuickCool used a period of 2k samples. A shorter cooling period encourages more frequent page migration between the memory tiers.

Memtis relies on Intel's Processor Event-Based Sampling (PEBS) to track memory access patterns. It samples various hardware events, including LLC misses, TLB misses, and retired store instructions, to infer accessed page addresses and build frequency-based histograms to aid in making migration decisions. Memtis currently only supports Intel-based systems, though it can be ported to AMD processors with Instruction-based Sampling (IBS). Thus, Memtis was not evaluated on platform D. Memtis works slightly differently on CXL-memory systems (platforms A and B) and the PM system (platform C). LLC misses to CXL memory are regarded as *uncore* events on Intel platforms and thus cannot be captured by PEBS. Therefore, Memtis relies solely on TLB misses and retired store instructions to infer page temperature on platforms A and B.

4.1 Micro-benchmarks

To evaluate the performance of NOMAD's transactional page migration and shadowing mechanisms, we developed a micro-benchmark to precisely assess NOMAD in a controlled manner. This micro-benchmark involves 1) allocating data to specific segments of the tiered memory; 2) running tests with various working set sizes (WSS) and resident set sizes (RSS); and 3) generating memory accesses to the WSS data that mimic real-world memory access patterns with a Zipfian distribution. We created three scenarios representing small, medium, and large WSS, as illustrated in Figure 6, to evaluate tiered memory management under different memory pressures. As platform B behaved similarly to platform A in micro-benchmarks, it is excluded from the discussion.

Small WSS. We began with a scenario with a small WSS of 10 GB and a total RSS of 20 GB. Initially, we filled the first 10 GB of local DRAM with the first half of the RSS data. Subsequently, we allocated 10 GB of WSS data as the



(a) Small working set size

(b) Medium working set size

(c) Large working set size

Figure 7: Performance comparison between TPP, Memtis-Default, Memtis-QuickCool, and NOMAD on platform A.

second half of the RSS – 6 GB on the local DRAM and 4 GB on CXL memory (platforms A and D) and the PM (platform C). The micro-benchmark continuously performed memory reads or writes (following a Zipfian distribution) to this 10 GB WSS data, spread across both the local DRAM and CXL memory or PM. The frequently accessed, or "hot" data, was uniformly distributed along the 10 GB WSS. In TPP and NOMAD, accessing data on CXL memory or PM triggered page migration to the local DRAM, with TPP performing this migration synchronously and NOMAD asynchronously. In contrast, Memtis used a background thread to migrate hot data from CXL memory or PM to the local DRAM. Due to page migration, the 4 GB WSS data, initially allocated to CXL memory or PM, was gradually moved to the local DRAM. Since the WSS was small (i.e., 10 GB), it could be completely stored in the fast tier (i.e., local DRAM) after the micro-benchmark reached a *stable* state.

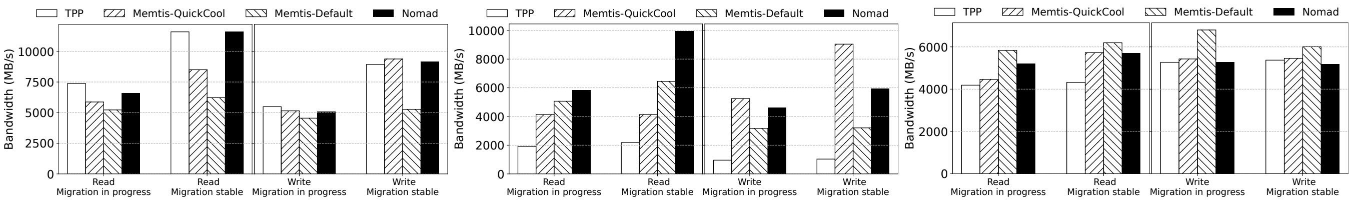
Figures 7 (a), 8 (a), and 9 (a) show that in the *transient* phase, during which page migration was conducted intensively (i.e., *migration in progress*), both NOMAD and Memtis demonstrated similar performance regarding memory bandwidth for reads. Although page-fault-based page migration in NOMAD could incur more overhead than the PEBS-based approach in Memtis, when the WSS can fit in fast memory and no memory thrashing occurs, the benefit of migration outweighs its overhead. For writes, e.g., on platform A, NOMAD incurred noticeable performance degradation compared to Memtis due to possibly aborted migrations and the maintenance of shadow pages. Note that NOMAD's overhead varies across platforms depending on the performance difference between fast and slow memory. In contrast, Nomad consistently outperformed TPP for both read and write, except for the slightly worse performance on platform C, highlighting the advantage of asynchronous page migration in NOMAD.

In the *stable* phase (i.e., *migration stable*), when most of the WSS data had been migrated from CXL memory or PM to the local DRAM, both NOMAD and TPP achieved similar read/write bandwidth. This was because memory accesses were primarily served by the local DRAM with few page migrations, as shown in Table 2. Memtis performed the worst, achieving as low as 40% of the performance of the other two approaches. We make two observations regarding Memtis's weaknesses. First, its stable phase performance is not drastically different from the transient phase. The migration statistics in Table 2 show that Memtis performed significantly fewer page migrations. This explains its sub-optimal performance in the stable phase as most memory accesses were still served from slow memory. Second, a shorter cooling period in Memtis, which incentivizes more frequent migrations, led to better performance. This also suggests that sampling-based memory access tracking may not accurately identify and timely migrate hot pages to fast memory.

Medium WSS. We increased the size of WSS and RSS to 13.5 GB and 27 GB, respectively. Similarly, we placed the first half of the RSS (13.5 GB) at the start of the local DRAM, followed by 2.5 GB of the WSS on the local DRAM, with the remaining 11 GB residing on CXL memory or PM. However, as the system (e.g., the OS kernel) required approximately 3-4 GB of memory, the WSS could barely fit in the fast tier, resulting in occasional and substantial migrations even during the stable phase. Accurately identifying hot pages and avoiding thrashing is crucial to achieving high performance for this medium-sized benchmark.

Unlike the small WSS case, Figures 7 (b), 8 (b), and 9 (b) show that during the transient phase, NOMAD and TPP generally achieved lower performance for both read and write compared to Memtis. This is because, under the medium WSS, the system experienced higher memory pressure than in the small WSS case, causing NOMAD and TPP to conduct more page migrations (2x - 6x) and incur higher overhead than Memtis, as shown in Table 2. Many of such migrations were futile during thrashing. Conversely, Memtis performed significantly fewer page migrations and avoided the waste. However, there was no evidence that Memtis effectively detected thrashing and throttled migration. The coarse-grained sampling was unable to accurately determine page temperature in a volatile situation and inadvertently sustained high performance under high memory pressure.

In the stable phase, NOMAD significantly outperformed TPP in all cases, especially on platform D. These results show the benefit of NOMAD's transactional page migration and non-exclusive memory tiering compared to TPP's synchronous page migration and exclusive tiering. On platform D, which was equipped with an application-specific integrated circuit (ASIC)-based CXL memory implementation, the performance gap between fast and slow memory narrows. Thus, the software overhead associated with synchronous page migration was exacerbated and NOMAD offered more pronounced per-

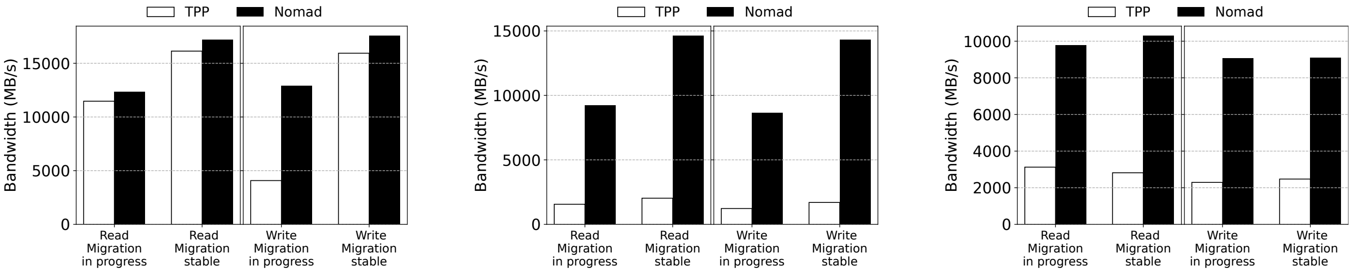


(a) Small working set size

(b) Medium working set size.

(c) Large working set size

Figure 8: Performance comparison between TPP, Memtis-Default, Memtis-QuickCool, and NOMAD on platform C.



(a) Small working set size

(b) Medium working set size

(c) Large working set size

Figure 9: Performance comparison between TPP and NOMAD on platform D with AMD Genoa processor. Memtis does not support AMD's instruction-based sampling (IBS) and thus was not tested.

formance gains. Similarly, NOMAD achieved substantially higher performance in reads than Memtis and comparable performance in writes. Unlike in the small WSS case, in which asynchronous and transactional page migration in NOMAD contributed most to its performance benefit, the advantage of page shadowing played a critical role in alleviating thrashing in the medium WSS case. Under memory thrashing, most demoted pages, which were recently promoted from the slow tier, can be simply discarded without migration. However, for write-intensive workloads, page shadowing requires one additional page fault for each write to restore a page's original read-write permission. This explains NOMAD's inferior write performance in the stable phase compared to Memtis.

Large WSS. We scaled up the WSS and RSS both to 27 GB and fully populated local DRAM with the first 16 GB of the WSS. The remaining WSS spilled onto CXL memory or PM. Unlike the medium WSS that incurred intermittent memory thrashing, this workload caused continuous and severe thrashing as the size of hot data greatly exceeded the capacity of fast memory. Figures 7 (c), 8 (c), and 9 (c) present the performance results in both the *transient* phase and the *stable* phase. Compared to the tests with the medium-sized workload in which NOMAD could outperform Memtis for read-only benchmarks, especially in the stable phase, both NOMAD and TPP performed worse than Memtis in almost all scenarios. It suggests that page fault-based tiered memory management, which makes per-page migration decisions upon access to a page, inevitably incurs high overhead during severe memory thrashing. Nevertheless, NOMAD consistently and significantly outperformed TPP thanks to asynchronous, transactional page migration and page shadowing.

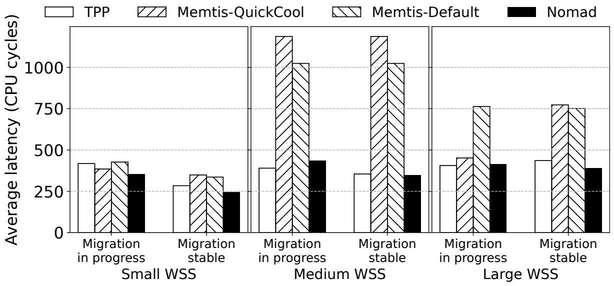


Figure 10: The average cache line access latency on platform C. The benchmark is a point-chasing workload optimized for PEBS-based memory access tracking.

Limitations of PEBS-based approaches. Our evaluation revealed several issues with PEBS-based memory access tracking. While Memtis prevented excessive migrations during thrashing, it achieved sub-optimal performance and failed to migrate all hot data to fast memory even when the WSS could fit in the fast tier. Due to the lack of hardware support for hot page tracking, PEBS-based approaches employ indirect metrics, such as LLC and TLB misses to sample recently accessed addresses to infer page temperature. Sampling-based memory tracking has two fundamental limitations. First, there is a difficult tradeoff between sampling rate and tracking accuracy. Second and most importantly, cache misses may not effectively capture hot pages. For most frequently accessed pages that always hit the caches, Memtis fails to collect enough (cache miss) samples to build the histogram. If such pages are evicted from the caches, e.g., due to conflict or coherence misses, they will be falsely regarded as "cold" pages.

To demonstrate these limitations, we created a favorable

RSS	23GB	25GB	27GB	29GB
Total shadow page size	3.93GB	2.68GB	2.2GB	0.58GB

Table 3: Shadow memory size as RSS changes on platform B. The size of tiered memory (DRAM+CXL) is 30.7 GB.

scenario where Memtis can capture every page access. We used a pointer-chasing benchmark that repeatedly accesses multiple fixed-sized (1 GB) memory blocks. Within each 1 GB block, the benchmark randomly accesses all cache lines belonging to a block while accesses across blocks follow a Zipfian distribution. The number of blocks determines the WSS. Since the block size exceeds the LLC size in our testbeds, every access generates an LLC miss that can be captured by Memtis. Effective memory access tracking should identify hot blocks and place them in fast memory.

Figure 10 shows the average latency to access a cache line in this benchmark on platform C. Note that platform C with PM was the only testbed on which Memtis has full tracking capability and can capture all the PEBS events. According to Table 1, a latency closer to DRAM performance (∼250 cycles) indicates more effective page placement. As shown in Figure 10, when the WSS exceeds fast tier capacity, Memtis achieved latency close to slow memory performance, suggesting that most hot pages still resided in the slow tier. In comparison, page fault-based approaches, e.g., NOMAD and TPP, can timely migrate hot pages and achieve low latency.

Robustness. Page shadowing can potentially increase memory usage and in the worst case can cause OOM errors if shadow pages are not timely reclaimed. In this test, we evaluated NOMAD's shadow page reclamation. We measured the total memory usage and the size of shadow memory using a micro-benchmark that sequentially scans a predefined RSS area. Table 3 shows the change of shadow pages as we varied the RSS. The results suggest that NOMAD effectively reclaimed shadow pages to reduce shadow memory usage as RSS increased and approached memory capacity.

4.2 Real-world Applications

We continued the evaluation of NOMAD using three representative real-world applications with unique memory access patterns: Redis [10], PageRank [9], and Liblinear [5]. We ran these three applications on four platforms (as shown in Table 1) with two configurations: 1) a small RSS (under 32 GB) working with all platforms and 2) a large RSS (over 32 GB) only on platform C and D with large PM or CXL memory. In addition, we include results from a "no migration" baseline which disables page migrations to show whether tiered memory management is necessary.

Key-value store. We first conducted experiments on a *latency-sensitive* key-value database, Redis [10]. The workload was generated from YCSB [11], using its *update-heavy* workload A, with a 50/50 distribution of read and write operations. We crafted three cases with different RSS and total operations.

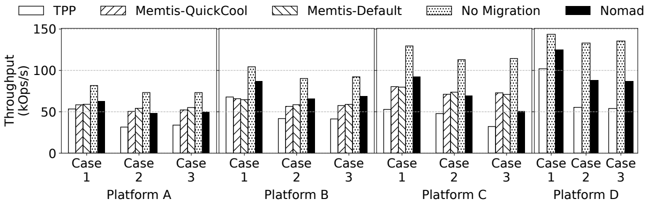


Figure 11: Performance comparisons using Redis and YCSB between TPP, Memtis-Default, Memtis-QuickCool, NOMAD, and "no migration".

Note that the parameters of YCSB were set as default unless otherwise specified. Case 1: We set *recordcount* to 6 million and *operationcount* to 8 million. After pre-loading the dataset, we used a customized tool to demote all memory pages to the slow tier before starting the experiment. The RSS of this case was 13GB. Case 2: We increased the RSS by setting *recordcount* to 10 million and *operationcount* to 12 million. We demoted all the memory pages to the slow tier in the same way. The RSS of this case was 24GB. Case 3: We kept the same total operations and RSS as Case 2. However, after preloading the dataset, we did *not* demote any memory pages.

Consistent with the micro-benchmarking results, Figure 11 shows that NOMAD delivered superior performance (in terms of operations per second) compared to TPP across all platforms in all cases. In addition, NOMAD outperformed Memtis when the WSS was small (i.e., in case 1), but suffered more performance degradation as the WSS increased (i.e., in case 2 and 3) due to an increased number of page migrations and additional overhead. Finally, all the page migration approaches underperformed compared to the "no migration" baseline. It is because the memory accesses generated by the YCSB workload were mostly "random", rendering migrating pages to the fast tier less effective, as those pages were unlikely to be accessed again. It indicates once again that page migration could incur nontrivial overhead, and a strategy to dynamically switch it on/off is needed.

We further increased the RSS of the database and operations of YCSB by setting the *recordcount* to 20 million and *operationcount* to 30 million. The RSS for this case was 36.5GB, exceeding the total size of the tiered memory on platforms A and B. Thus, the large RSS test was only performed on platforms C and D. We tested two initial memory placement strategies for the database – 1) *thrashing* that allocated all pages first to the slow tier and immediately invoked intensive page migrations, and 2) *normal* that prioritized page allocation to fast memory and triggered page migration only under memory pressure. As shown in Figure 14, NOMAD outperformed TPP due to its graceful performance degradation during thrashing but fell short of matching Memtis's performance. The initial placement strategy did not substantially affect the results and performance under different placements eventually converged.

Graph-based computation. We used PageRank [9], an ap-

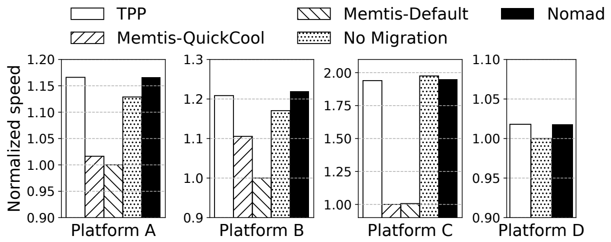


Figure 12: Performance comparisons of PageRank between non-migration, TPP, Memtis, and NOMAD. Performance is normalized to the approach with the lowest speed.

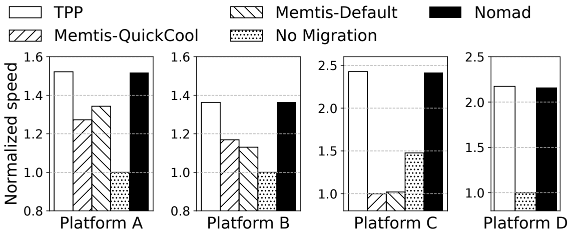


Figure 13: Performance comparisons of Liblinear between non-migration, TPP, Memtis, and NOMAD. Performance is normalized to the approach with the lowest speed.

plication used to rank web pages. It involves iterative computations to determine the rank of each page, based on the link structures of the entire web. As the size of the dataset increases, the computational complexity also increases, making it both memory-intensive and compute-intensive. We used a benchmark suite [8] to generate a synthetic uniform-random graph comprising 2^{26} vertices, each with an average of 20 edges. The RSS in this experiment was 22 GB, indicating that the memory pages were distributed at both the local DRAM and remote CXL memory or PM.

Figure 12 illustrates that there was negligible variance in performance between scenarios with page migrations (using NOMAD and TPP) and without page migrations (no migration). The results suggest that: 1) For *non-latency-sensitive* applications, such as PageRank, using CXL memory can significantly expand the local DRAM capacity without adversely impacting application-level performance. 2) In such scenarios, page migration appears to be unnecessary. These findings also reveal that the overhead associated with NOMAD's page migration minimally influences PageRank's performance. Additionally, it was observed that among all evaluated scenarios, Memtis exhibited the least efficient performance.

Figure 15 shows the case when we scaled the RSS to a very large scale on platforms C & D. When the PageRank program started, it first used up to 100GB memory, then its RSS size dropped to 45GB to 50GB. NOMAD achieved 2x the performance of TPP (both platforms) and slightly better than Memtis (platform C), due to more frequent page migrations – the local DRAM (16 GB) was not large enough to accommodate the WSS in this case.

Workload type	Success : Aborted
Liblinear (large RSS) on platform C	1:1.9
Liblinear (large RSS) on platform D	2.6:1
Redis (large RSS) on platform C	153:1
Redis (large RSS) on platform D	278.2:1

Table 4: The Success rate of transactional migration.

Machine learning. Our final evaluation of NOMAD involved using the machine learning library Liblinear [5], known for its large-scale linear classification capabilities. We executed Liblinear with an L1 regularized logistic regression workload with an RSS of 10 GB. Prior to each execution, we used our tool to demote all memory pages associated with the Liblinear workload to the slower memory tier.

Figure 13 demonstrates that both NOMAD and TPP significantly outperformed "no migration" and Memtis across all platforms, with performance improvement ranging from 20% to 150%. This result further illustrates that when the WSS is smaller than the local DRAM, NOMAD and TPP can substantially enhance application performance by timely migrating hot pages to the faster memory tier. Figure 16 shows that with a much larger model and RSS when running Liblinear, NOMAD consistently achieved high performance across all cases. In contrast, TPP's performance significantly declined, likely due to inefficiency issues, as frequent, high bursts in kernel CPU time were observed during TPP execution.

Migration success rate. As stated in Section 3.1, NOMAD's transactional page migration may be aborted due to updates to the migrating page, resulting in wasted memory bandwidth and CPU cycles. Subsequent retries could also fail. A low success rate could negatively affect application performance. Table 4 shows NOMAD's migration success rate for Liblinear and Redis on platforms C and D. We chose a large RSS for both applications and ensured there were sufficient cross-tier migrations. We observed a low success rate for Liblinear while Redis had a high success rate. Interestingly, this contrasted with NOMAD's performance – it was excellent with Liblinear but poor with Redis with large RSS. This suggests that a high success rate in page migrations does not necessarily lead to high performance. A low success rate indicates that the pages being migrated by NOMAD are also being modified by other processes, implying their "hotness". Timely migration of such pages can benefit ongoing and future accesses.

Summary. The results from micro-benchmarks and applications indicate that when the WSS was smaller than the performance tier, NOMAD enabled workloads to maintain higher performance than Memtis through asynchronous, transactional page migrations. However, when the WSS was comparable to or exceeded the performance tier capacity, leading to memory thrashing, the page-fault-based migration in NOMAD became detrimental to workload performance, underperforming Memtis in write operations. Notably, NOMAD's page shadowing feature preserved the efficiency of read opera-

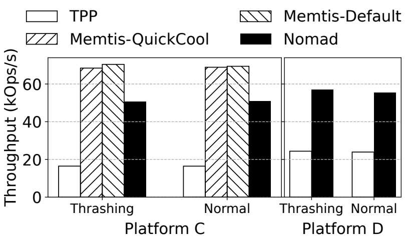


Figure 14: Redis (large RSS).

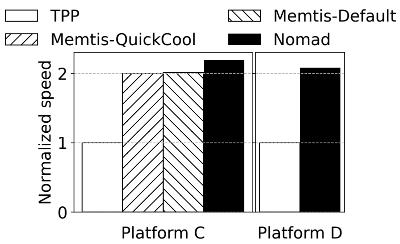


Figure 15: Page ranking (large RSS).

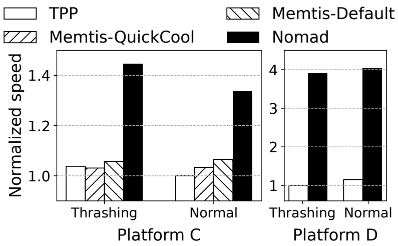


Figure 16: Liblinear (large RSS).

tions even under severe memory thrashing, often maintaining comparable or superior performance to Memtis. In all test scenarios, NOMAD significantly outperformed the state-of-the-art page-fault-based migration approach, TPP.

The evaluation results across four different platforms reveal the following observations: First, NOMAD generally performed better on platform D, which was equipped with faster and larger CXL memory, compared to other platforms. Additionally, the reduced performance gap between fast and slow memory on platform D allowed NOMAD to achieve greater performance gains than TPP, as the performance overhead from TPP's synchronous page migration was more pronounced. Second, while platforms A and B showed similar behavior in micro-benchmarks, their application-level performance varied (slightly) across different applications, suggesting that specific CPU features (differing between the off-the-shelf Intel Sapphire Rapids CPU for platform A and the engineering sample for platform B) may affect the performance of page migration under more realistic workloads.

5 Discussions and Future Work

The key insight from NOMAD's evaluation is that page migration, especially under memory pressure, has a detrimental impact on overall application performance. While NOMAD achieved graceful performance degradation and much higher performance than TPP, an approach based on synchronous page migration, its performance is sub-optimal compared to that without page migration. When the program's working set exceeds the capacity of the fast tier, the most effective strategy is to access pages directly from their initial placement, completely disabling page migration. It is straightforward to detect memory thrashing, e.g., frequent and equal number of page demotions and promotions, and disable page migrations. However, estimating the working set size to resume page migration becomes challenging, as the working set now spans multiple tiers. It requires global memory tracking, which could be prohibitively expensive, to identify the hot data set that can potentially be migrated to the fast tier. We plan to extend NOMAD to unilaterally throttle page promotions and monitor page demotions to effectively manage memory pressure on the fast tier. Note that this would require the development of a new page migration policy, which is orthogonal to the NOMAD page migration mechanisms proposed in this work.

Impact of Platform Characteristics: There exist difficult tradeoffs between page fault-based access tracking, such as TPP and NOMAD, and hardware performance counter-based memory access sampling like Memtis. While page fault-based tracking effectively captures access recency, it can be potentially expensive and on the critical path of program execution. In comparison, hardware-based access sampling is off the critical path and captures access frequency. However, it is not responsive to workload changes and its accuracy relies on the sampling rate. One advantage of NOMAD is that it is a page fault-based migration approach that is asynchronous and off the critical path. A potential future work is integrating NOMAD with hardware-based, access frequency tracking, such as Memtis, to enhance the current migration policy.

6 Conclusion

This paper introduces non-exclusive memory tiering as an alternative to the common exclusive memory tiering strategy, where each page is confined to either fast or slow memory. The proposed approach, implemented in NOMAD, leverages transactional page migration and page shadowing to enhance page management in Linux. Unlike traditional page migration, NOMAD ensures asynchronous migration and retains shadow copies of recently promoted pages. Through comprehensive evaluations, NOMAD demonstrates up to 6x performance improvement over existing methods, addressing critical performance degradation issues in exclusive memory tiering, especially under memory pressure. The paper calls for further research in tiered memory-aware memory allocation.

7 Acknowledgments

We thank our shepherd, Sudarsun Kannan, and the anonymous reviewers for their constructive feedback. This work was supported in part by NSF grants CCF-1845706, CNS-2415774, CCF-2415473, and the gift and equipment from Intel Labs and Micron Technology.

References

[1] https://www.computeexpresslink.org/.

[2] Autonuma: the other approach to numa scheduling. https://lwn.net/Articles/488709/.

[3] Damon-based reclamation. https://docs.kernel.org/admin-guide/mm/damon/reclaim.html#:~:text=DAMON%2Dbased%20Reclamation%20(DAMON_RECLAIM),of%20memory%20pressure%20and%20requirements.

[4] https://blocksandfiles.com/2023/11/20/accelerating-high-bandwidth-memory-to-light-speed/. https://blocksandfiles.com/2023/11/20/accelerating-high-bandwidth-memory-to-light-speed/.

[5] https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/multicore-liblinear/.

[6] Intel agilex® 7 fpga and soc fpga. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7.html.

[7] Intel optane dimm. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[8] Pagerank. https://github.com/sbeamer/gapbs.

[9] Pagerank wiki. https://en.wikipedia.org/wiki/PageRank.

[10] Redis. https://redis.io/.

[11] Ycsb. https://github.com/brianfrankcooper/YCSB.

[12] Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)* (San Jose, CA, Feb. 2012), USENIX Association.

[13] Abulila, A., Mailthody, V. S., Qureshi, Z., Huang, J., Kim, N. S., Xiong, J., and Hwu, W.-m. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 971–985.

[14] Agarwal, N., and Wenisch, T. F. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN Not. 52*, 4 (apr 2017), 631–644.

[15] Ahmadian, S., Salkhordeh, R., and Asadi, H. Lbica: A load balancer for i/o cache architectures. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (2019), pp. 1196–1201.

[16] Alian, M., Shin, J., Kang, K.-D., Wang, R., Daglis, A., Kim, D., and Kim, N. S. Idio: Orchestrating inbound network data on server processors. *IEEE Computer Architecture Letters 20*, 1 (2021), 30–33.

[17] AMD. Zynq ultrascale+ device technical reference manual (ug1085), 2023. https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/TLB-Maintenance-Operations.

[18] ARM. Learn the architecture - aarch64 memory management guide, 2024. https://developer.arm.com/documentation/101811/0103/Translation-Lookaside-Buffer-maintenance/Format-of-a-TLB-operation.

[19] Bergman, S., Faldu, P., Grot, B., Vilanova, L., and Silberstein, M. Reconsidering os memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2022), ISMM 2022, Association for Computing Machinery, p. 1–14.

[20] Bock, S., Childers, B. R., Melhem, R., and Mossé, D. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (New York, NY, USA, 2014), CF '14, Association for Computing Machinery.

[21] Burnett, N. C., Bent, J., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Exploiting Gray-Box knowledge of Buffer-Cache management. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)* (Monterey, CA, June 2002), USENIX Association.

[22] Chen, F., Koufaty, D. A., and Zhang, X. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing* (2011), pp. 22–32.

[23] Chou, C., Jaleel, A., and Qureshi, M. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2017), MEMSYS '17, Association for Computing Machinery, p. 268–280.

[24] Duraisamy, P., Xu, W., Hare, S., Rajwar, R., Culler, D., Xu, Z., Fan, J., Kennelly, C., McCloskey, B., Mijailovic, D., Morris, B., Mukherjee, C., Ren, J., Thelen, G., Turner, P., Villavieja, C., Ranganathan, P., and Vahdat, A. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 727–741.

[25] Forney, B. C., and Arpaci-Dusseau, A. C. Storage-Aware caching: Revisiting caching for heterogeneous storage systems. In *Conference on File and Storage Technologies (FAST 02)* (Monterey, CA, Jan. 2002), USENIX Association.

[26] Guerra, J., Pucha, H., Glider, J., Belluomini, W., and Rangaswami, R. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)* (San Jose, CA, Feb. 2011), USENIX Association.

[27] Holland, D. A., Angelino, E., Wald, G., and Seltzer, M. I. Flash caching on the storage client. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 127–138.

[28] Intel®. Remote action request –white paper. *revision 1.0* (2021). https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf.

[29] Jiang, S., Ding, X., Chen, F., Tan, E., and Zhang, X. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *4th USENIX Conference on File and Storage Technologies (FAST 05)* (San Francisco, CA, Dec. 2005), USENIX Association.

[30] Jun, H., Cho, J., Lee, K., Son, H.-Y., Kim, K., Jin, H., and Kim, K. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)* (2017), IEEE, pp. 1–4.

[31] Jung, M. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). HotStorage '22, Association for Computing Machinery, p. 45–51.

[32] Kim, J., Choe, W., and Ahn, J. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 715–728.

[33] Kim, Y., Gupta, A., Urgaonkar, B., Berman, P., and Sivasubramaniam, A. Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (2011), pp. 227–236.

[34] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. Write policies for host-side flash caches. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, Feb. 2013), USENIX Association, pp. 45–58.

[35] KOLLER, R., MASHTIZADEH, A. J., AND RANGASWAMI, R. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing* (2015), pp. 51–60.

[36] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 460–477.

[37] LEE, T., MONGA, S. K., MIN, C., AND EOM, Y. I. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 17–34.

[38] LI, H., BERGER, D. S., HSU, L., ERNST, D., ZARDOSHTI, P., NOVAKOVIC, S., SHAH, M., RAJADNYA, S., LEE, S., AGARWAL, I., ET AL. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), pp. 574–587.

[39] LIN, Z., XIANG, L., RAO, J., AND LU, H. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 801–815.

[40] LIN, Z., XIANG, L., RAO, J., AND LU, H. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 801–815.

[41] LIU, K., ZHANG, X., DAVIS, K., AND JIANG, S. Synergistic coupling of ssd and hard disk for qos-aware virtual memory. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013), pp. 24–33.

[42] LOH, G. H., JAYASENA, N., CHUNG, J., REINHARDT, S. K., O'CONNOR, J. M., AND MCGRATH, K. J. Challenges in heterogeneous die-stacked and off-chip memory systems.

[43] MARUF, A., GHOSH, A., BHIMANI, J., CAMPELLO, D., RUDOFF, A., AND RANGASWAMI, R. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Los Alamitos, CA, USA, apr 2022), IEEE Computer Society, pp. 925–937.

[44] MARUF, H. A., WANG, H., DHANOTIA, A., WEINER, J., AGARWAL, N., BHATTACHARYA, P., PETERSEN, C., CHOWDHURY, M., KANAUJIA, S., AND CHAUHAN, P. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 742–755.

[45] MESWANI, M. R., BLAGODUROV, S., ROBERTS, D., SLICE, J., IGNATOWSKI, M., AND LOH, G. H. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (2015), pp. 126–136.

[46] PAPAGIANNIS, A., XANTHAKIS, G., SALOUSTROS, G., MARAZAKIS, M., AND BILAS, A. Optimizing memory-mapped {I/O} for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 813–827.

[47] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 315–332.

[48] SIM, J., LOH, G. H., KIM, H., OCONNOR, M., AND THOTTETHODI, M. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), pp. 247–257.

[49] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (USA, 2010), FAST'10, USENIX Association, p. 8.

[50] SUN, Y., YUAN, Y., YU, Z., KUPER, R., JEONG, I., WANG, R., AND KIM, N. S. Demystifying cxl memory with genuine cxl-ready systems and devices. *ArXiv abs/2303.15375* (2023).

[51] WU, K., GUO, Z., HU, G., TU, K., ALAGAPPAN, R., SEN, R., PARK, K., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 307–323.

[52] WU, X., AND REDDY, A. N. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010), pp. 14–23.

[53] XIANG, L., ZHAO, X., RAO, J., JIANG, S., AND JIANG, H. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 488–505.

[54] YAN, Z., LUSTIG, D., NELLANS, D., AND BHATTACHARJEE, A. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 331–345.

[55] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (USA, 2020), FAST'20, USENIX Association, p. 169–182.

[56] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[57] YANG, Q., AND REN, J. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), pp. 278–289.

A Artifact Appendix

Abstract

The artifact contains the source code of NOMAD, TPP, and Memtis for reproducing the results and graphs presented in the paper. The code works on platforms with Persistent Memory, Intel Agilex CXL memory, and/or Micron CXL memory. To facilitate the reproduction, we have provided a collection of scripts for compiling and installing these approaches, executing the experiments, collecting logs, and creating graphs. More details are available in the "README.md" file.

Scope

This artifact demonstrates NOMAD's strengths and weaknesses over TPP and Memtis across various scenarios and platforms, as elaborated in the Evaluation section.

It is open-source and can be used for further research, development, or other purposes by the community.

Contents

NOMAD, TPP and Memtis implementation. We provide two separate patches to enable NOMAD and TPP to work with the upstream kernel version v5.13-rc6. In particular, the TPP patch comes from the Linux community email discussions. Memtis, on the other hand, is directly incorporated from its original artifact, with a few minor bugs fixed.

Documentation The "Reproducing Paper Results" section of "README.md" provides a step-by-step guide for reproducing the results in the paper. This guide includes instructions for compiling the three implementations (i.e., NOMAD, TPP, and Memtis), running the experiments, and generating the graphs as presented in the paper.

Hosting

Artifact link: https://github.com/lingfenghsiang/Nomad
Artifact license: GNU GPL V3.0
Artifact version tag: v0.0

Requirements

To reproduce the results in the paper, the system under test requires one NUMA node with a CPU and another CPU-less NUMA node. If the system has more NUMA nodes, the operating system might encounter unexpected errors. Additionally, Memtis is only fully functional on platforms with Optane Persistent Memory. More details are included in the "Prerequisites" section of "README.md".

Managing Memory Tiers with CXL in Virtualized Environments

Yuhong Zhong ♔ ⊞ Daniel S. Berger ⊞ W Carl Waldspurger* Ryan Wee ♔
Ishwar Agarwal i Rajat Agarwal i Frank Hady i Karthik Kumar i Mark D. Hill W
Mosharaf Chowdhury M Asaf Cidon ♔

♔ *Columbia University* ⊞ *Microsoft Azure* W *University of Washington* **Carl Waldspurger Consulting*
i *Intel* W *University of Wisconsin–Madison* M *University of Michigan*

Abstract

Cloud providers seek to deploy CXL-based memory to increase aggregate memory capacity, reduce costs, and lower carbon emissions. However, CXL accesses incur higher latency than local DRAM. Existing systems use software to manage data placement across memory tiers at page granularity. Cloud providers are reluctant to deploy software-based tiering due to high overheads in virtualized environments. Hardware-based memory tiering could place data at cacheline granularity, mitigating these drawbacks. However, hardware is oblivious to application-level performance.

We propose combining hardware-managed tiering with software-managed performance isolation to overcome the pitfalls of either approach. We introduce *Intel® Flat Memory Mode*, the first hardware-managed tiering system for CXL. Our evaluation on a full-system prototype demonstrates that it provides performance close to regular DRAM, with no more than 5% degradation for more than 82% of workloads. Despite such small slowdowns, we identify two challenges that can still degrade performance by up to 34% for "outlier" workloads: (1) memory contention across tenants, and (2) intra-tenant contention due to conflicting access patterns.

To address these challenges, we introduce *Memstrata*, a lightweight multi-tenant memory allocator. Memstrata employs page coloring to eliminate inter-VM contention. It improves performance for VMs with access patterns that are sensitive to hardware tiering by allocating them more local DRAM using an online slowdown estimator. In multi-VM experiments on prototype hardware, Memstrata is able to identify performance outliers and reduce their degradation from above 30% to below 6%, providing consistent performance across a wide range of workloads.

1 Introduction

Memory tiering is a promising approach to scale memory capacity and reduce the total cost of ownership (TCO) in datacenters. In public clouds, virtual machine (VM) memory sizes are increasing, with typical configurations of 4–32GB per virtual CPU [6, 7, 12]. However, the DRAM capacity accessible via DDR channels is lagging the rapid growth in available cores, due to physical limitations associated with scaling the capacity of DDR DIMMs [73, 91, 92]. To this end, cloud providers are increasingly adding a *capacity memory tier* to augment regular locally-accessed DRAM, which we refer to as the *performance tier* [61, 72, 78, 84, 99].

The recent Compute Express Link (CXL) standard [8, 91] offers a new mechanism to access DRAM or non-volatile memory (NVM) over the PCIe bus, potentially expanding memory capacity significantly. In addition, CXL can reduce TCO and carbon emissions [83, 98] by provisioning it with decommissioned DRAM or NVM. This has led to broad investment in CXL memory by dozens of vendors [9, 10, 15, 25, 27, 28, 37]. The CXL standard envisions a variety of configurations. In this paper, we focus on the basic use case where a CXL memory device is locally attached and dedicated to a single host [91, 98]. This use case is deployable today, and extends to future memory pools [46, 77, 78].

Most prior work on memory tiering assumes software (e.g., the hypervisor or the OS) has full control over data placement, i.e., whether a particular page resides in the capacity tier or the performance tier [45, 61, 70, 74, 78, 84, 89, 90, 99, 100]. We term this *software-managed memory tiering*. Software-managed tiering needs to track memory accesses to identify frequently-accessed data to place in the performance tier. Since the hypervisor/OS is not involved in most memory accesses, it must rely on page table operations management (e.g., scanning access bits [61, 84, 100] or PTE poisoning [45, 70, 84]) or instruction sampling (e.g., Intel PEBS sampling [61, 74, 89] and AMD IBS [4]) to track memory accesses.

However, in our experience at Microsoft Azure, these approaches face severe limitations in virtualized environments (§2). For example, instruction sampling is not supported for VMs and has privacy implications. Fine-grained page table operations consume excessive host CPU cycles [44, 79]. In addition, with software-managed tiering, the hypervisor/OS can only manage memory at page granularity. This leads to suboptimal decisions [76] for the common case where a mix of hot and cold data resides on the same page. This is particularly problematic for hypervisors that use larger page sizes (e.g., 2 MB and 1 GB) to reduce overheads. All of these drawbacks make deploying software-based memory tiering techniques unattractive in general-purpose cloud environments.

This paper addresses these issues by introducing a hardware-managed memory tiering solution for CXL and a system that combines hardware-managed tiering with software-managed multi-tenant isolation. We introduce *Intel® Flat Memory Mode* as the first cache-line granular, hardware-managed memory tiering solution for CXL. Intel® Flat Memory Mode transparently manages data placement between the two tiers at cache-line granularity within the processor memory controller (MC). It exposes the aggregate capacity of both local DRAM and CXL memory to software by placing data *exclusively* at either of the tiers. The hardware promotes the most recently accessed lines to local DRAM by "swapping" them with the lines that used to occupy local DRAM.

To reduce the performance degradation of CXL memory, Intel® Flat Memory Mode supports a *mixed mode* which reserves a certain number of *dedicated* pages that are guaranteed to reside in local memory, while cache lines associated with the remaining pages may be placed in either local or CXL memory, based on whether they were recently accessed.

Intel® Flat Memory Mode should not be confused with the hardware-managed memory tiering solution for Intel® Optane™ NVDIMMs, known as *2LM* or *memory mode* [18, 65]. Such systems employ DRAM as an *inclusive* cache for non-volatile memory, which means the performance tier does not add capacity visible to software. The inclusive cache design makes them less useful for expanding memory capacity and reducing TCO. Additionally, 2LM only supports non-volatile memory, not CXL.

We describe Intel® Flat Memory Mode's design and evaluate it on a real CXL hardware prototype with a set of 115 workloads, comparing it to running fully on local DRAM. We find that 82% of workloads experience small (no more than 5%) slowdown in mixed mode. The remaining "outlier" workloads experience slowdowns up to 34%. We also observe that when VMs are co-located naïvely on the same server, they may interfere by "stealing" local DRAM from each other.

To address these challenges, we implement *Memstrata*, the first multi-tenant memory management software stack for hardware-managed tiered memory. Memstrata prevents inter-VM interference by identifying pages with conflicting cache lines, allocating them to the same VM using *page coloring*. In addition, Memstrata leverages a lightweight online slowdown estimator to assess the overhead incurred by tiered memory misses for each VM. It dynamically allocates dedicated local memory pages across VMs to improve the performance of those that are most sensitive to memory latency. Intel® Flat Memory Mode will be available in the Intel® Xeon® 6 Processor. We open source Memstrata at https://bitbucket.org/yuhong_zhong/memstrata.

We implement a full system prototype on a preproduction Intel® Xeon® 6 Processor that supports Intel® Flat Memory Mode. Memstrata is implemented within the Linux/KVM hypervisor and a new user-space management process. Our evaluation covers common workload and VM mixes observed in production at Azure. We find that Memstrata effectively prevents cross-VM interference and mitigates the tail in all scenarios. Specifically, the worst-case performance slowdown is reduced from 35% to less than 6% in realistic multi-VM experiments. Across all experiments, the maximum CPU overhead of Memstrata is 4% of a single core, which is less than 1% of a single core per VM.

We make the following contributions:

1. We introduce Intel® Flat Memory Mode, the first hardware-managed memory tiering mechanism for CXL. We evaluate it on a real CXL system, and show that for most applications it exhibits small slowdowns.
2. We design Memstrata, the first software multi-tenant management system for hardware-managed CXL that ensures performance isolation and minimizes VM slowdowns.
3. We study a wide range of workloads, and demonstrate that Intel® Flat Memory Mode combined with Memstrata eliminates almost all performance outliers, exhibiting minimal performance degradation compared to regular DRAM.

2 Background and Motivation

This section motivates hardware-managed memory tiering for CXL in virtualized environments.

2.1 Memory Tiering in Public Clouds

Current compute servers, which host customer VMs, use locally-attached DDR5 memory. With CPU core counts of 60-96 [66, 94] and Simultaneous Multithreading (SMT), achieving at least 4-8GB per virtual core requires 8-12 expensive dual-rank DIMMs (e.g., 64GB or 96GB). These DIMMs are the single biggest contributor to server cost [78, 99]. For large-memory VM sizes [6] or 128-288 core-count-CPUs [2, 14, 57], cloud providers need to use DIMMs with 3D stacking, which adds a multiplicative factor to per-GB memory cost [32]. Additionally, DIMMs make up 41% of a server's embodied carbon at Azure [49, 63, 83, 87, 98].

A second tier of memory can effectively reduce this cost. In modern servers, this second tier will use CXL [8, 91] to expand server memory capacity and bandwidth. This saves cost because cloud providers can use multiple smaller and cheaper DIMMs. Cost can be further reduced by *reusing memory from decomissioned servers*. Without CXL, DDR4 memory would be incompatible with modern servers. Instead of discarding DDR4 DIMMs, they can be repurposed for CXL memory. DDR4 reuse is supported today and has significant industry momentum [3, 26, 49, 83, 98]. This pattern can continue in future generations of DRAM, e.g., when DDR6 will be deployed, DDR5 can be reused with CXL. A third option is denser memory media [17, 38]. Both reusing old memory and using denser memory significantly cuts costs and carbon emissions. For example, attaching 40% of memory by reusing DDR4 can save over 20% of server embodied carbon emissions [83]. In this paper, we focus on this use case.

The downside of CXL memory is its latency overhead. CXL.mem customizes the PCIe link and transaction layers for low latency [91]. CPUs can natively access CXL memory via cacheable loads and stores, without involving page faults or DMAs. While an order of magnitude faster than RDMA, CXL is still slower than local DRAM as it essentially converts a parallel bus into a serial one. Depending on the specific memory controller, we measure that CXL memory has 2.02× the load-to-use latency of local DDR5 on the 5th Gen Intel® Xeon® Processor. A bidirectional ×8-CXL port at a typical 2:1 read:write-ratio matches a DDR5-4800 channel. In practice we use at least four ×8 ports.

2.2 Cloud Workload and Design Goals

Azure and other large cloud providers virtualize all workloads. VMs are generally small. For example, in a typical compute cluster at Azure, 40% of VMs use no more than two cores and 86% of VMs use no more than eight cores. Most modern hosts thus run dozens of VMs at any given time. Production cluster schedulers [47, 64, 97] increase utilization by mixing different workloads with no (or few) co-location constraints. Some constraints force similar workloads to be run across many hosts and racks, e.g., for fault tolerance. This leads to typical hosts running heterogeneous sets of workloads representing many different workload behaviors.

We derive the following four first-order design goals:

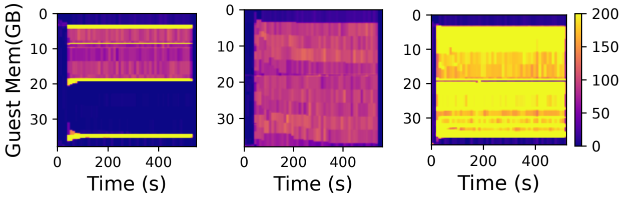
1. Compatibility with unmodified virtual machines. Do not assume guest cooperation.
2. Low host resource overheads. Cloud providers seek to sell almost all cores [44, 79]. Hosts typically use large 2 MB or 1 GB page sizes to reduce overhead.
3. No additional sources of cross-VM interference compared to running entirely on local memory.
4. Performance close to local memory for all workloads. Limit slowdown to about 5%, similar to prior work [78].

As observed in prior work, CXL slowdowns can be high for many workloads [78, 84]. This motivates managing data placement in tiered memory, either in software or in hardware.

2.3 Software-Managed Tiering

Software-managed tiering usually represents tiers as NUMA nodes [61, 78, 84]. Software explicitly allocates memory from a NUMA node and migrates pages between nodes. The hypervisor/OS typically tracks memory hotness to promote hot capacity-tier pages to the performance tier and demote cold pages to the capacity tier [61, 84, 89, 100]. Hotness tracking often relies on page table operations such as scanning PTE access bits [61, 84, 100] or temporarily unmapping entries to trigger minor page faults when they are accessed [45, 70, 84]. Other software tiering systems use instruction sampling (e.g., Intel PEBS [61, 74, 89] or AMD IBS [4]) to sample memory requests along with their associated memory addresses.

Problem 1: High host CPU cost. Tracking hotness at fine granularity is challenging in a cloud environment. Instruction sampling is typically unfeasible due to security and privacy concerns. Thus, cloud platforms need to rely on page-table-based approaches. We find that they can consume excessive host CPU cycles, which runs counter to design goal #2.



(a) Fresh VM (b) Randomized (c) Warm VM

Figure 1: Memory access distribution of `bc-web` measured using DAMON in (a) a fresh VM, (b) a VM that enables free page randomization, and (c) a warm VM that has already run the workload 50 times. The right-hand y-axis represents the number of accesses captured by DAMON.

We measure the CPU overhead of TPP [84], a state-of-the-art software tiering system for guest kernels. We start a VM with 7.5 GB of local DRAM and 2.5 GB of second-tier memory. The VM runs YCSB A on FASTER [54], a production in-memory key-value store. FASTER consumes 8.3 GB memory in total, which means its memory cannot fit entirely in local DRAM. TPP devotes nearly an entire core to track memory accesses and migrate pages. This is caused by the frequent scanning of access bits in `kswapd`, which is used by TPP to demote cold pages. Without frequent access-bit scanning, TPP is unable to leave enough free space in local DRAM to promote hot pages. Scaling to larger systems and multiple VMs requires proportionally more CPU cycles.

The CPU overhead of page-table-based approaches can be reduced by exploiting spatial locality [11]. Unfortunately, spatial locality is limited in virtualized systems, which employ an additional layer of page table indirection. Additionally, a guest's free pages may be randomized for security [21]. We run the `bc-web` workload from the GAP benchmark suite [48] in a fresh VM, a VM with free page randomization enabled, and a warm VM that has already run the same workload 50 times. Figure 1 shows the memory access distribution measured using DAMON in the three VMs. While there is spatial locality in a fresh VM, locality disappears in both the VM with free page randomization and the warm VM. Approaches that scan guest page tables [90] may overcome fragmentation but run counter to goals #1 and #2.

Problem 2: Coarse-grained data placement. Software tiering moves entire pages, making a strong assumption about access locality. Many applications have spatially-sparse access patterns and thus perform poorly on software-managed tiering systems [53, 76]. Commonly, only a fraction of each page's cachelines are hot; moving such pages to the performance tier would be wasteful. This problem is exacerbated as cloud platforms use larger 2 MB and 1 GB page sizes to reduce page table depth and TLB misses [1, 34, 43, 45, 50].

To study how page size affects application performance, we

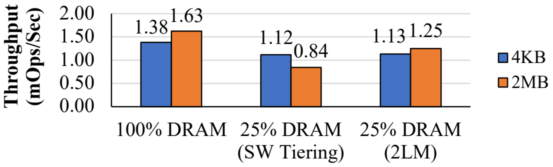


Figure 2: Throughput of FASTER on YCSB A with a varying DRAM ratio and different page sizes. The performance tier is DRAM and the second tier is Intel® Optane™ NVM.

run a VM with FASTER using the YCSB A workload. Since FASTER has a simple and predictable memory access pattern when running YCSB A, we analytically compute the popularity of each of its pages and always place the most popular pages in DRAM. Figure 2 shows that when FASTER's resident set can fit into DRAM, using 2 MB instead of 4 KB as the page size improves the throughput of FASTER by 18% thanks to the reduced page table depth and TLB misses. However, when only 25% of the resident set can fit into DRAM, a 2 MB page size degrades throughput by 25% due to the coarser data placement by software tiering. Google also reports that huge pages make cold page identification and demotion more challenging in their tiered memory production clusters [61].

2.4 Hardware-Managed Tiering

There are multiple variants of hardware memory tiering [68, 75, 85, 103]. They are typically implemented within the MC on the CPU SoC and behave similar to on-die CPU caches. Different memory tiers are typically invisible to software (no NUMA node) and fine-grained cache operations are visible only to the MC.

A well-known implementation of hardware tiering is "2LM" or "memory mode" for Intel® Optane™ NVDIMMs in the 2nd and the 3rd Gen Intel® Xeon® Scalable Processors [18, 65]. 2LM configures DRAM as a direct-mapped cache at cacheline granularity. It thus has no hotness tracking overhead and excels at managing workloads with limited locality [76], regardless of the page size (Figure 2). A major downside of using 2LM in the context of CXL is that the second tier is inclusive of the performance tier. This is wasteful, especially for the case of cloud providers who seek high performance and thus provision a large first memory tier. For example, provisioning 600 GB of DDR5 and 1000 GB of CXL memory means that only 1000 GB of overall memory is available, wasting 60% of CXL capacity.

3 Intel® Flat Memory Mode

In this section, we describe the hardware design of Intel® Flat Memory Mode and present a performance study on a wide range of applications.

3.1 Hardware Design

Intel® Flat Memory Mode overcomes the drawbacks of software-managed memory tiering by implementing the data placement within the MC. This allows it to manage data placement at cacheline granularity without involving host CPU. This design is especially useful in virtualized environments because the data placement is independent of the page size, and almost all the host CPU cores can be used to run VMs.

CXL memory ratio. To ensure minimal slowdown compared to local memory[1] (design goal #4), we assume a 1:1 ratio between the local memory and the CXL memory capacities. Other tiered memory deployments in industry also use small capacity-tier ratios to minimize slowdown: 33% at Meta [84] and 25% at Google [61]. With a 1:1 ratio, we can reduce the amount of local memory provisioned by 50%, which already significantly reduces memory cost. A higher CXL memory percentage may lead to higher slowdowns [78].

Exclusive placement. The amount of physical memory exposed to software is the aggregate capacity of both local DRAM and CXL memory. This is in contrast to 2LM, where the physical memory capacity is only as large as the size of the capacity tier (i.e., non-volatile memory). This design fully utilizes the capacity of both local DRAM and CXL memory by placing data *exclusively* at either of them, but not both. For example, once a cacheline is moved to local DRAM, it will no longer occupy any space in CXL memory.

Associativity. The associativity between physical memory and local memory is direct-mapped, which means each line in the physical memory address space can only be cached at one location in local memory. While direct-mapped associativity may lead to more conflict misses, this effect happens only after all the processor set-associative caches have missed. In addition, a straightforward implementation of set associativity would read multiple local DRAM lines to serve one main memory access, causing substantial bandwidth amplification.

Mixed mode. To further reduce local memory misses and improve the performance of workloads with cache-unfriendly memory access patterns, Intel® Flat Memory Mode supports adding dedicated local DRAM that is not hardware-tiered as a separate range in the physical memory address space. This dedicated local memory is exposed as a second NUMA node alongside the first NUMA node which contains the hardware-tiered memory. This dedicated NUMA node can be used for workloads suffering from severe local memory misses, as implemented in Memstrata (§4). We denote configurations where both hardware-tiered and dedicated NUMA nodes are present as *mixed mode*.

Mapping physical and local memory. In the hardware-tiered NUMA node, the ratio between local DRAM and the total physical memory capacity is 1:2. Thus, each line in local DRAM has 2 physical memory lines that map to it. This means that each 64 B line in the physical address space may be at one of two locations: either in local memory or in CXL memory. We use a modulo operation as the mapping function between the physical memory and local DRAM with the size of local DRAM (L GB) as the modulus. Figure 3 shows

[1] We use "local memory" and "local DRAM" interchangeably.

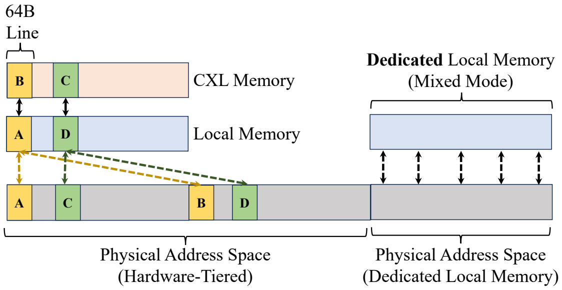
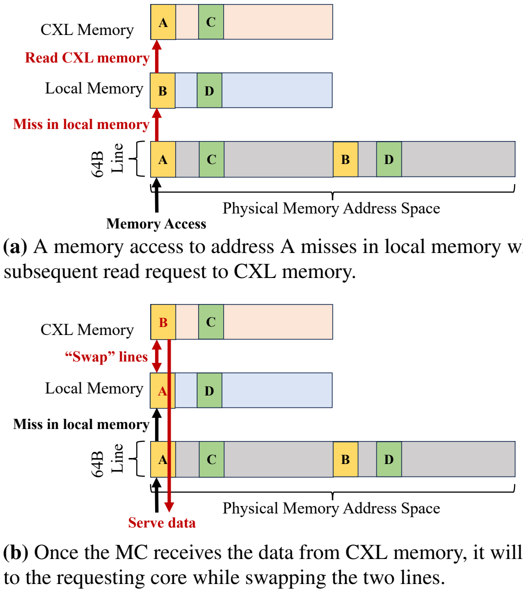


Figure 3: Intel® Flat Memory Mode is a hardware tiering solution at cache line granularity between local and CXL memory. It supports a mixed mode option where part of the address space is not hardware managed and entirely backed by local memory for better performance. In the actively-managed region, cache lines A and B are mapped to the same local memory line, as are C and D. Only the most recently-accessed lines remain in local memory.



(a) A memory access to address A misses in local memory which leads to a subsequent read request to CXL memory.

(b) Once the MC receives the data from CXL memory, it will serve the data to the requesting core while swapping the two lines.

Figure 4: Main memory requests may miss in local memory. This triggers a cache line swap. Subsequent accesses to the same cache line hit local memory.

the mapping between the physical address space and local memory. This mapping halves the hardware-tiered physical memory, where the top half conflicts with the bottom half. For example, to hold physical memory addresses $[0\ GB, 1\ GB]$ in local DRAM, the hardware needs to evict $[L\ GB, L+1\ GB]$ to CXL memory.

Read and write operations. A memory access (i.e., Last Level Cache (LLC) miss) for a cache line in the hardware-tiered NUMA node may result in a "hit" or a "miss". The MC first reads local memory and determines if the requested line is in local memory. As only two physical memory lines can be cached at a given local memory line, the hardware only needs to maintain a single-bit tag to distinguish between them. If the tag matches the read request, the data is sent to the core that requested the line.

Otherwise, the requested line was a miss in local memory. Figure 4 shows how the MC handles a miss. The MC first fetches the data from CXL memory, and then sends the data to the core that requested the line. Meanwhile, the MC swaps the cache lines. Specifically, the MC writes the other line that used to occupy local memory to CXL memory. The MC writes the newly requested line to local memory.

When the MC receives a write request, just like the read flow, it needs to first locate and read the line into the processor caches. When the write is evicted from the processor caches (since writes are posted), the data is written to local memory.

Request interleaving. To achieve maximum bandwidth, we interleave local memory requests across memory channels and interleave CXL memory requests across CXL devices at cache line granularity within the same NUMA node.

3.2 Application Performance

Adding CXL memory capacity can provide clear performance benefits to memory-hungry applications, due to reduced paging to disk, and higher page-cache hit rates. However, these benefits depends heavily on the specific workloads and the total amount of memory available to them. To conservatively evaluate Intel® Flat Memory Mode, we compare it with X local memory and Y CXL memory to a baseline configured with $X+Y$ local memory. We evaluate the performance using a wide range of applications on a prototype CPU that supports Intel® Flat Memory Mode. The detailed hardware setup is described in §6. We use 115 workloads in total, including:

- **Web**: DaCapo [51], Renaissance [88], Ruby YJIT [39], and DeathStarBench benchmarks [62]
- **Database**: TPC-C [41] on Silo [96] and TPC-H [42] on PostgreSQL [33]
- **Machine learning (ML)**: DLRM benchmark [67, 86]
- **Key-value (KV) store**: YCSB [58] on FASTER [54], Redis [36], and memcached [23]
- **Big data**: HiBench [13] on Spark [104]
- **Graph processing**: GAP benchmark [48]
- **Scientific computing**: SPEC CPU 2017 [40]

We measure the performance of each workload running inside a VM on Linux/KVM. The VM memory size is chosen by rounding up the workload's peak resident set size to the nearest common VM memory size on public cloud platforms (2 GB, 4 GB, 8 GB, 16 GB, 32 GB, and 64 GB) [6, 7, 12]. We run each workload in four different settings: (1) local DRAM only, (2) CXL memory only, (3) hardware-tiered memory only, and (4) a mixed mode with 33% dedicated local DRAM and 67% hardware-tiered memory.

When allocating hardware-tiered memory pages to a VM, we allocate pairs of conflicting pages so that half of the allocation can be cached in local DRAM. As a result, our mixed mode configuration consists of 67% local DRAM (33% dedicated + half of 67% hardware-tiered) and 33% CXL memory. We conservatively choose 67% as the percentage of hardware-tiered memory as this configuration can already reduce the

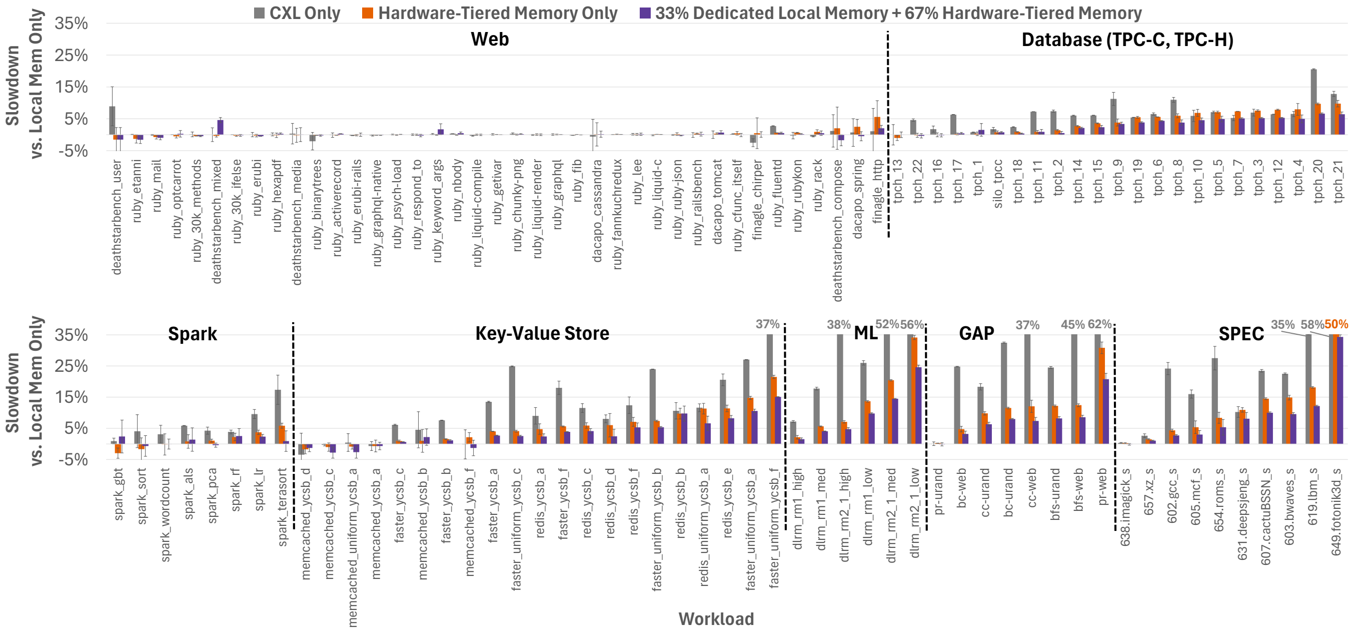


Figure 5: Slowdowns of 115 workloads when using only CXL memory, 100% hardware-tiered memory, or a mixed mode with 33% dedicated memory and 67% hardware-tiered memory. The error bars represent the standard deviations of slowdowns across three runs.

provisioned local memory by 33%. We randomly assign the dedicated local pages across a VM's address space. Each VM sees a uniform address space, as this is the default configuration on cloud platforms.

Figure 5 presents the results, categorizing workloads by their types, and ordering them by their relative slowdowns of hardware tiering. We refer to applications that experience slowdowns above 5% (see goal #4 in §2.2) as *outliers*. Web workloads experience negligible slowdowns even with CXL memory only, indicating their insensitivity to main memory latency. In contrast, database and Spark workloads have some outliers with slowdowns of up to 20% when using only CXL memory. Hardware tiering reduces the slowdowns for most outliers to close to or lower than 5%, although a few outliers still exhibit around 10% degradation. Other categories have more outliers with CXL memory only, with slowdowns of up to 58%. Hardware tiering significantly alleviates the performance degradation for these outliers. For example, the slowdown of FASTER with uniform YCSB C is reduced from 25% to 4%. However, even with reduced degradation, some outliers still experience slowdowns of up to 50% with hardware-tiered memory due to cache-unfriendly memory access patterns and the associated high local memory miss ratios. The most severe outlier, `649.fotonik3d_s`, suffers from a 41% miss ratio due to its large working set and scan-like memory access pattern.

The mixed mode with 33% dedicated local memory improves the performance of these outliers thanks to the reduction in the number of pages that conflict on local DRAM. Overall, with only hardware-tiered memory, 73% of the workloads experience no more than 5% slowdown, and 86% experience no more than 10% slowdown. In the mixed mode, the percentage of workloads with no more than 5% slowdown increases to 82%, and 95% of the workloads experience no more than 10% slowdown. These results are encouraging: despite the non-negligible slowdown of CXL compared to local DRAM, most applications have small slowdowns in the mixed mode. However, even in the mixed mode, some applications experience non-trivial degradation of up to 34%. This observation motivates the use of software to dynamically allocate dedicated memory pages across VMs to consistently achieve minimal slowdown, because the hardware is oblivious to which VMs suffer from local memory misses.

3.3 Noisy Neighbors

In Intel® Flat Memory Mode, two conflicting physical memory lines compete for the same local DRAM line, and only the most recently accessed one can be cached in local DRAM. Therefore, when conflicting pages are allocated to different VMs, they may contend for local memory, resulting in performance interference.

We study this inter-VM interference due to local DRAM conflicts by running two VMs: a normal VM and a *noisy neighbor* VM. In the normal VM, we run one of the workloads from our workload set, while in the noisy-neighbor VM, we always run a 6-thread Intel® Memory Latency Checker (MLC) [19], which scans its memory in a busy loop. We always scale MLC to have the same memory size as the normal VM. We configure MLC to use only 6 threads so that neither the local DRAM bandwidth nor the CXL memory bandwidth is saturated. Running MLC as the workload in the noisy neighbor VM allows us to estimate the worst-case interference, as MLC is optimized to be memory intensive.

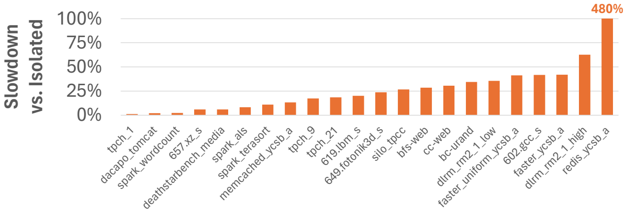


Figure 6: Slowdown caused by the noisy neighbor due to local DRAM conflicts.

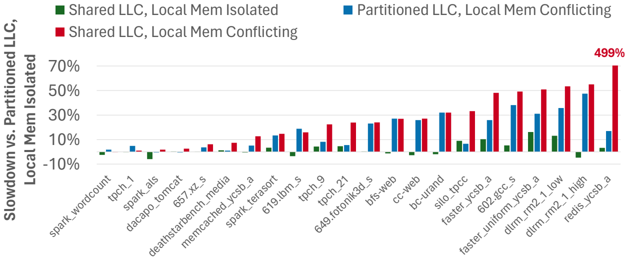


Figure 7: Inter-VM interference caused by LLC contention and local DRAM contention.

The experiments are conducted in two settings:

1. **Isolated.** Allocate conflicting pages to the same VM to ensure each VM only conflicts with itself.
2. **Conflicting.** Allocate conflicting pages to different VMs. This setting measures the worst-case interference since the noisy neighbor VM might monopolize local DRAM.

Figure 6 shows the slowdown of conflicting compared to isolated with a sampled set of representative workloads from each category. 73% of the workloads experience more than 10% slowdown because of local DRAM conflicts. The massive slowdown of Redis is because we use p95 latency as its performance metric. Redis always has some requests with extreme latency (4× higher than the median), and the contention causes the percentage of these requests to exceed 5%, which translates to a 480% slowdown in p95 latency. The results indicate that without any software management to isolate local DRAM conflicts, VMs running on the same host could cause significant performance interference to each other.

Besides local DRAM contention, other sources of interference in multi-tenant environments include contention in the LLC and power. We also study how LLC contention compares to local memory contention in Intel® Flat Memory Mode. We again use a normal VM and a noisy neighbor VM to measure interference. Besides configuring how the two VMs conflict with each other in local memory, we configure the LLC in two settings: (1) sharing LLC across two VMs, or (2) partitioning LLC evenly between two VMs. We use Intel's Cache Allocation Technology [20] to partition the LLC.

When the LLC is partitioned and local DRAM conflicts are isolated, there will be no interference caused by either LLC or local memory contention. When the LLC is shared but local DRAM conflicts are still isolated, we will only observe LLC interference. Similarly, we can only observe local memory interference if the LLC is partitioned and the two VMs are conflicting in local DRAM. Finally, to measure both LLC and local memory interference, we can share the LLC and also let the two VMs are conflicting in local memory.

Figure 7 shows the slowdowns caused by either LLC or local memory interference, as well as the slowdowns when both types of interference exist. Compared to LLC interference, local memory interference is typically larger. In addition, the workloads that suffer from LLC interference also suffer from local memory interference. When both LLC and local memory interference exist, those workloads experience higher slowdowns than when there is a single source of interference. These results again indicate that we should isolate local DRAM conflicts to achieve design goal #3.

4 Memstrata

Memstrata leaves the heavy lifting of fine-grained memory management to the hardware-managed tiering layer at the MC. It provides consistent performance by integrating a lightweight software stack with the virtualization host. This achieves the first two design goals (§2.2).

To provide performance isolation (design goal #3), Memstrata adapts page coloring [69, 105], a classic technique for partitioning CPU caches, to the CXL setting. Memstrata identifies all conflicting pages and allocates conflicting pairs to the same VM, ensuring no inter-VM conflicts (§4.1).

To improve the performance of outliers (design goal #4), Memstrata dynamically allocates dedicated local memory pages across VMs to reduce the outliers' local memory miss rates. Our key insight is that many workloads exhibit low slowdowns even without any dedicated local memory. Therefore, if the hypervisor can identify outlier VMs and move dedicated local memory pages to them, it can limit their slowdowns.

However, the hypervisor has limited visibility into the workloads running inside VMs, making it challenging to identify outliers. Although monitoring local DRAM miss rates seems attractive for detecting outliers, we cannot directly measure per-core or per-VM miss rates because data placement is implemented in the MC, so hardware performance counters can report only the system-wide local memory miss rate.

To tackle these challenges, we analyze numerous performance events measured during our application performance study (§3.2) and propose a proxy to estimate per-VM miss rates. By combining the estimated miss rate with other performance metrics, we can accurately predict the slowdown of a VM using a simple online ML model (§4.2). This model is used by a dynamic page allocator to migrate dedicated local memory pages across VMs, minimizing slowdowns across all workloads with negligible CPU overhead (§4.3). Figure 8 shows an overview and the workflow of Memstrata.

4.1 Page Coloring

Page coloring is a software technique that has been widely used to partition shared processor caches (e.g., the LLC) in a modern CPU [80, 93, 95, 102, 105]. CPU caches are commonly

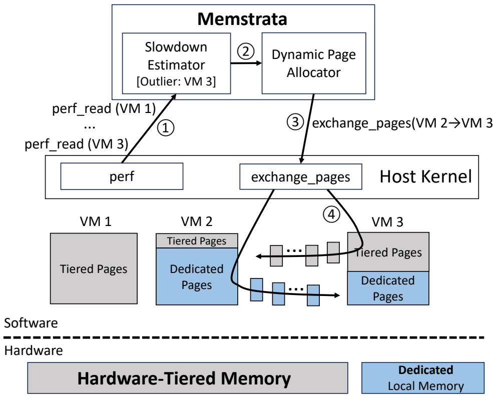


Figure 8: Overview of Memstrata. ①: Memstrata reads the performance events of each VM and runs the slowdown estimator. ②: Slowdown estimations are used to decide the allocation of dedicated pages. ③: The dynamic page allocator uses the `exchange_pages` syscall to migrate dedicated pages to an outlier. ④: The hardware-tiered pages of the outlier are exchanged with the dedicated pages of a non-outlier VM.

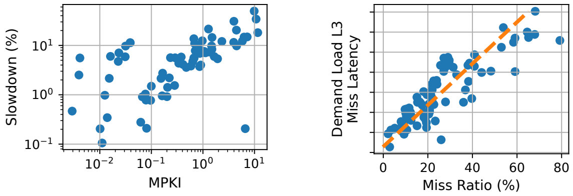
organized in a set-associative (or direct-mapped) manner, in which each physical memory address is mapped to the index of a single set in the cache. If the indexing function is known, then software can determine the subset of cache indices associated with a given memory page, referred to as its *page color*. Since main memory is much larger than the cache, many memory pages have the same color, which means that they compete for the same limited cache space.

System software can control the amount of cache space that may be used by different applications by allocating pages to them with particular colors. For example, a hypervisor can allocate host-physical pages so that each VM uses distinct colors that are disjoint from other VMs.

Similar to shared processor caches, in Intel® Flat Memory Mode, local memory is shared among all VMs within the same NUMA node.[2] Physical memory lines that are mapped to the same local memory line compete for the same local memory space, which will contain the one accessed most recently. Therefore, we can adopt page coloring to partition local memory pages across different VMs to avoid inter-VM local memory conflicts.

We implement page coloring in the context of a virtualized system configured with Intel® Flat Memory Mode. The implementation consists of changing the free-page management logic and the page allocator in the host Linux kernel. We modify Linux's free-page management to group the physical pages that map to the same local DRAM page.

To avoid performance interference due to inter-VM conflicts, the page allocator always allocates the physical pages that map to the same local memory page to the same VM. This isolates each VM, ensuring that it can conflict only with itself, eliminating the possibility of conflicts with other VMs. Isolating local memory conflicts within VMs also prevents their use as inter-VM side channels, similar to those exploited in other caching systems to leak memory access patterns and to exfiltrate data across trust boundaries [71, 81, 82].



(a) MPKI vs. slowdown **(b)** Miss ratio vs. L3 miss latency

Figure 9: Correlation between performance metrics.

While one might expect this to cause poor performance, there are, in fact, many cold or unused cache lines. This means that we actually observe relatively low local memory miss rates (§3.2).

4.2 Identifying Outliers

To improve the performance of outlier workloads, we first need to determine which VMs suffer from high slowdowns because of local DRAM misses. Since the workloads running inside VMs are opaque to the hypervisor, we cannot rely on application-level performance metrics for outlier detection.

Fortunately, modern processors provide performance counters that can be used to infer VM performance characteristics with low overhead [5, 16, 31, 35, 101]. Our prototype CPU also supports various performance events. In our large-scale performance study of Intel® Flat Memory Mode using 115 workloads (§3.2), we configured the CPU to count all performance events for each workload by time-multiplexing them, yielding 151 performance metrics based on raw event counts.

MPKI and its proxy. Among all the performance metrics, local memory miss rate seems promising for detecting outliers, since the local memory is treated as a cache. Specifically, we examine MPKI, which measures the number of local memory misses per thousand instructions. Figure 9a shows that MPKI is correlated ($r^2 = 0.73$) with application slowdown.

Estimating slowdown for a single VM requires per-VM miss rates. As each VM is pinned to a set of exclusive CPU cores, this suggests aggregating the miss rates across its associated cores to compute the VM-level miss rate. Unfortunately, since cacheline promotions and demotions are handled in the MC, it is not easy to track per-core misses, and the system-wide miss rate is insufficient for outlier detection. Although future hardware may support measuring per-core miss rates, this is not implemented in the current prototype.

To work around this limitation, we analyze other performance metrics that can be tracked with per-core granularity to find a proxy for the per-core miss rate. As shown in Figure 9b, we find that the L3 miss latency[3] of demand loads event has a strong linear correlation ($r^2 = 0.87$) with the local

[2] For clarity, we only discuss the hardware-tiered memory *without* dedicated local memory in this subsection. Dedicated local memory is contained in a distinct NUMA node from the local memory associated with the hardware-tiered memory.

[3] Figure 9b omits the y-axis latency scale for confidentiality.

memory miss ratio (not MPKI), defined as the percentage of main memory requests that miss in local memory. This is not surprising, as Intel® Flat Memory Mode exhibits stable hit and miss latencies unless the memory bandwidth is saturated. We leverage this observation by fitting a linear model to estimate the local memory miss ratio from demand load L3 miss latency. The estimated miss ratio is translated to MPKI by multiplying it with the main memory request count and then dividing by the instruction count. We use this estimated MPKI as a proxy for the actual MPKI.

A limitation of this approach is that demand loads represent only a portion of main memory requests. Other sources include read-for-ownership (RFO) requests, non-temporal stores, CPU cache writebacks, and CPU cache prefetches. However, we find that the estimated MPKI works well in practice. When combined with other metrics, it can be used to predict VM slowdown accurately.

Using a model to detect outliers. Although MPKI strongly correlates with application slowdown, we find that MPKI alone is not sufficient to identify outliers because applications can have different sensitivities to memory latency. Therefore we use an online random forest binary classifier [52] to determine whether a VM will experience more than 5% slowdown. We choose a random forest classifier because it performs well with low-level performance metrics [78], is lightweight, and does not require a GPU. The input to the classifier includes the estimated MPKI along with four additional per-VM metrics that also exhibit useful correlations, selected by computing the relative importance of features during classifier training: (a) L3 miss latency of demand loads, (b) L2 miss latency of demand loads, (c) data TLB load miss latency, and (d) L2 MPKI of demand loads. We evenly split the workloads into training and validation sets, and configure the random forest with 100 decision-tree estimators. The classifier achieves 100% accuracy on the training set and 88% accuracy on the validation set, demonstrating the ability to detect outliers across a diverse set of workloads. In contrast, using MPKI as the only feature achieves only 63% accuracy on the validation set.

4.3 Dynamic Page Allocator

The Memstrata dynamic page allocator manages how dedicated local memory pages are allocated across VMs. It uses the ML model to detect outlier VMs, and migrates dedicated local memory pages accordingly to achieve minimum slowdown across all workloads. Within each VM, the page allocator assigns dedicated local memory pages to guest physical pages randomly. We also implemented an alternative hotness-based approach that prioritizes popular guest physical pages, but found that its overhead typically exceeds its benefit.

Inter-VM page allocation. The dynamic page allocator allocates dedicated pages to each VM based on its performance events and slowdown predictions from the ML model. It starts by measuring the events needed by the ML model for a given time interval (10 seconds, by default), and runs the model to predict if the slowdown for each VM is greater than 5%. The 10-second interval enables the page allocator to react quickly to changes, while averaging out noise associated with low-level event counts. To reduce the effect of short-term variations, we employ an exponentially weighted moving average (EWMA) to smooth the performance metrics derived from the event counts (EWMA constant $\alpha = 0.2$, by default).

```
def comparator(vm1, vm2):
  if vm1.isOutlier != vm2.isOutlier:
    return vm2.isOutlier
  return vm1.avgMissCount < vm2.avgMissCount

def migrate(vms, timeInterval, ewma, stepRatio):
  while systemIsRunning():
    sleep(timeInterval)
    updatePerfMetrics(vms, ewma)
    predictSlowdown(vms)
    sort(vms, comparator)
    donor = 0
    for borrower in range(len(vms) - 1, 0, -1):
      if not vms[borrower].isOutlier:
        break
      toBorrow = vms[borrower].pages * stepRatio
      while donor < borrower and toBorrow > 0:
        toDonate = (vms[donor].pages * stepRatio
                    - vms[donor].donated)
        toMigrate = min(toBorrow, toDonate)
        doMigrate(borrower, donor, toMigrate)
        toBorrow -= toMigrate
        vms[donor].donated += toMigrate
        if toDonate == toMigrate:
          donor += 1
```

Listing 1: Inter-VM page migration algorithm.

Once the page allocator obtains the performance metrics and slowdown prediction for each VM, it decides how dedicated local memory pages should be migrated across VMs. Listing 1 presents the page migration algorithm. The page allocator first ranks the VMs based on their predicted slowdowns and the average number of local DRAM misses per allocated hardware-tiered page. The average miss count is computed by multiplying the estimated miss ratio with the main memory request count, and dividing the result by the number of hardware-tiered pages assigned to the VM. The VMs predicted to have less than 5% slowdown receive lower ranks than the outlier VMs. VMs that have the same slowdown prediction are ordered based on their average miss count. The intuition is that prioritizing VMs with higher average miss counts minimizes system-wide local memory misses, since they benefit more from a fixed amount of dedicated local memory pages compared to others [56].

After the VMs are sorted according to their ranks, the allocator repeatedly migrates dedicated pages from the VM with the lowest rank to the VM with the highest rank. To prevent large performance fluctuations, it never migrates more than a fraction `stepRatio` of each VM's pages (10%, by default) during each step. Only VMs predicted to be outliers can receive dedicated local memory pages from others.

To migrate dedicated local memory pages from VM 1 to

VM 2, the page allocator first selects a given number of dedicated pages from VM 1 and the same amount of hardware-tiered pages from VM 2. It then exchanges the selected pages between the two VMs. To avoid introducing inter-VM local DRAM conflicts, conflicting pages are always migrated together. After one round of page migrations, the page allocator stops migrating and measures performance events over the next `timeInterval` before the next round of migration.

Launching and terminating VMs. In cloud environments, running VMs may be terminated and new VMs may be launched at any time. To avoid disruptions, the page allocator first removes a terminating VM from the list of active VMs that participate in page migration. The terminating VM can then be shut down, causing its pages to be returned to the free page pool maintained by the host kernel.

When a new VM is launched, its initial allocation consists of existing free pages from the host kernel, which could be any mix of hardware-tiered pages and dedicated local memory pages. Since the dynamic page allocator does not have any prior information about the new VM, once it is added to the active VM list, the allocator migrates pages so that the new VM contains the same percentage of dedicated pages as the entire system. For example, if the overall system has a total of 33% dedicated local memory and 67% hardware-tiered memory, the new VM will also have 33% dedicated pages. This initial migration is performed by taking (or giving) dedicated pages to (or from) other existing VMs, each of which contributes (or receives) the same number of dedicated pages.

Assigning dedicated memory. By default, the dynamic page allocator assigns dedicated local memory pages to guest physical pages randomly within each VM. We also experimented with an alternative hotness-based page allocation option, which prioritizes popular guest physical pages when allocating dedicated pages. To identify popular guest physical pages, we employ DAMON [11], a low-overhead memory access tracking subsystem integrated into the mainstream Linux kernel. We use DAMON's default settings, but configure its aggregation period to match the `timeInterval` used by the allocator. After finishing inter-VM page migration, the allocator checks the per-region access counts reported by DAMON. Using simple thresholds (cold = 0, hot $\geq$ 20, by default), it exchanges any hardware-tiered pages in hot regions with dedicated pages in cold regions. To avoid large performance fluctuations, such intra-VM migrations are limited to a small fraction (2%, by default) of the VM memory size.

However, we found that the overhead of this hotness-based approach exceeds it benefit (§6.2). Similar to software-managed tiering, it consumes significant CPU cycles to track memory accesses (§2.3). Therefore, by default, Memstrata simply assigns dedicated local pages randomly.

5 Memstrata Implementation

Memstrata's implementation consists of implementing page coloring and the page-exchange system call in the host Linux kernel (v5.19, 2729 LOC), modifying QEMU (v6.2, 60 LOC) to preallocate guest memory for VMs, and building the main functionality of Memstrata as a privileged userspace process that runs on the host (2190 LOC, C++). Like QEMU, Memstrata uses 2 MB as the page size at the host level.

The page-exchange system call `exchange_pages(pid_1, pid_2, page_arr_1, page_arr_2, num_pages)` accepts the PIDs of two processes, an array of linear addresses for each process, and the number of pages to exchange. One can exchange pages within a single process by specifying the same value for `pid_1` and `pid_2`. The syscall is implemented in the host kernel based on the `migrate_pages()` function. To exchange two physical pages, the kernel initially moves the first page to a temporary physical page, then transfers the second page to occupy the first page's original location, and finally relocates the temporary page to the second page's initial position. It uses a Linux MMU notifier [24] to synchronize the secondary page table used by the VM and the QEMU host-level page table.

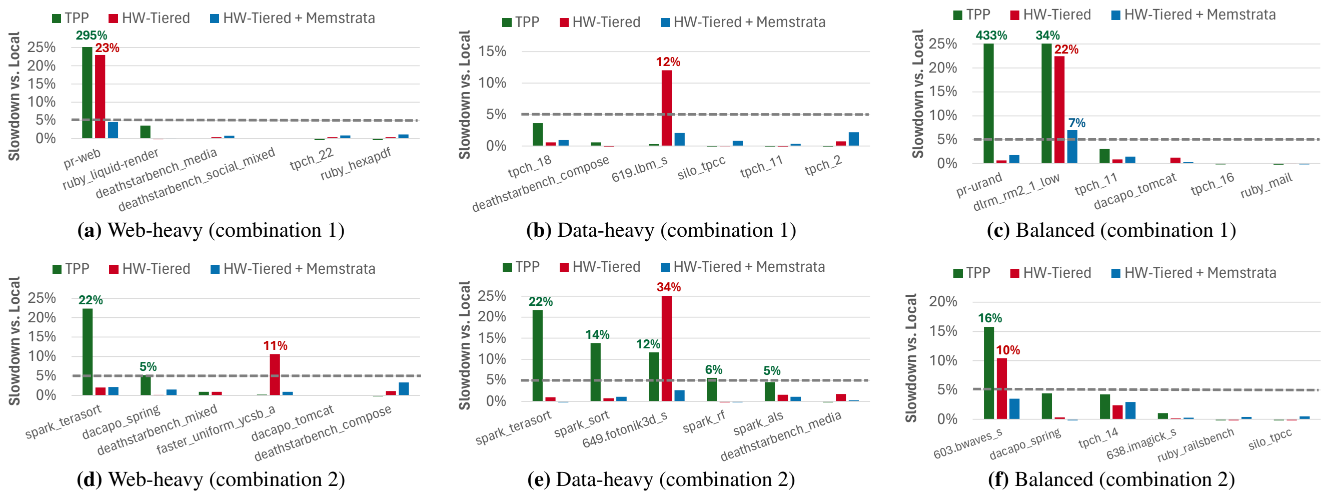
We implement only the necessary mechanisms (i.e., page coloring and page exchange) in the host kernel and run Memstrata as a privileged userspace process, facilitating debugging and extensions. The userspace process configures and reads performance events via the `perf` interface exposed by the host Linux kernel. It uses the custom page-exchange syscall to exchange pages between two VMs, or within a single VM. We use ONNX [60] to run the ML model in the userspace process. To synchronize Memstrata with VM launching and termination, we use POSIX message queues to let the VM scheduler communicate with Memstrata.

6 Evaluation

In this section we seek to answer the following questions:

1. How does Intel® Flat Memory Mode compare to software-managed tiering? (§6.1)
2. Can Memstrata improve the performance of outliers without impacting other applications? (§6.1)
3. How does dedicated memory page allocation affect application performance? (§6.2)
4. Is Memstrata sensitive to its parameters? (§6.3)

Experimental setup. We use a pre-production Intel® Xeon® 6 Processor that implements Intel® Flat Memory Mode. Our test server contains a single socket with 128GB DDR5 local memory and 128GB DDR5 CXL memory. The CXL memory is attached via three CXL cards, which each hold two DDR5-4800 DIMMs and offer an x16 PCIe5 CXL connection. We use a preproduction Astera Labs Leo CXL Smart Memory Controller [22]. The idle latency of the CXL memory is roughly 200-220% the latency of the local memory, and the max bandwidth per CXL card is around 50 GB/s [91]. Although we use DDR5-4800 DIMMs in the CXL cards, we believe the results are transferable to DDR4 DIMMs because the actual CXL bandwidth usage is always below the limit



(a) Web-heavy (combination 1)

(b) Data-heavy (combination 1)

(c) Balanced (combination 1)

(d) Web-heavy (combination 2)

(e) Data-heavy (combination 2)

(f) Balanced (combination 2)

Figure 10: Application slowdown of TPP and hardware tiering with and without Memstrata using different workload combinations.

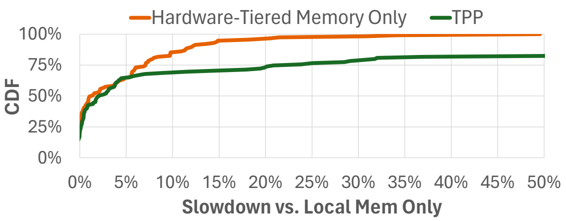


Figure 11: Slowdown distribution of hardware-tiered memory and TPP. TPP is configured with 50% local memory to match the local memory ratio of hardware-tiered memory.

of DDR4 DIMMs. Since the CPU is pre-production, the core count, frequency, and CPU cache sizes may not reflect those of the final product. For confidentiality, we cannot share the detailed technical specification. Similarly, these CXL cards are pre-production and future versions may be faster.

We focus on end-to-end application performance to demonstrate the performance benefits of Memstrata. Our software stack comprises Ubuntu 22.04, modified Linux 5.19, and QEMU/KVM 6.2. Hyperthreading and CPU frequency scaling are disabled. We pin each VM's virtual cores to physical cores in a 1:1 manner. We set the VM memory size of each workload by rounding up its peak resident set size to the next largest VM memory size offered on public cloud platforms.

We do not use public or Azure VM traces since they do not label workloads for VMs. This is because public cloud providers are not generally aware of workloads running inside VMs. Therefore, we rely on analyses of the composition of 188 internal workloads over 100,000 VMs at Azure [98], which reveal that web (31%), big data (32%), and ML (11%) workloads constitute most of the VMs. The remaining workload categories include DevOps and real-time communication workloads, which are challenging to run and have few open-source representatives. Therefore, we focus on the web, big data, and ML workload categories and reuse the set of workloads from §3.2 to match this composition. We exclude the workloads that have unstable performance. With the prototype system only offering 128 GB local memory, we also exclude workloads that require more than 32 GB of memory, so that we can measure a multi-VM local-only baseline.

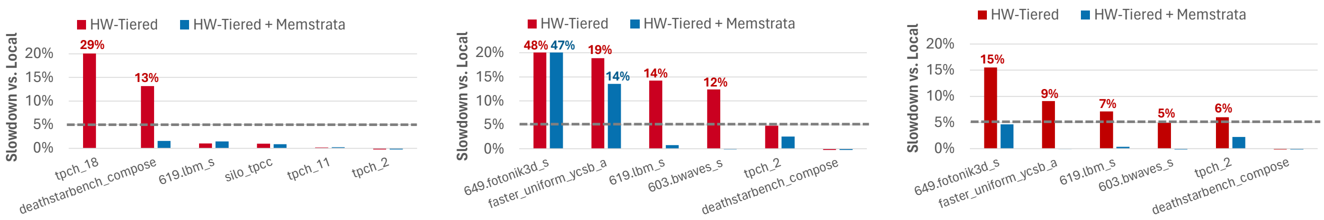
We compare Intel® Flat Memory Mode without and with Memstrata (referred to in the figures as "HW-Tiered" and "HW-Tiered + Memstrata", respectively). To emulate a setting without Memstrata, we use a static allocation scheme in which the percentage of dedicated pages in each VM remains constant over time. In contrast, Memstrata dynamically migrates dedicated pages across VMs to minimize slowdown. All settings use page coloring to avoid inter-VM conflicts.

We also compare hardware tiering without Memstrata to TPP [84], a state-of-the-art software-tiering approach. Since TPP does not support virtualization, we run it within each one of the isolated VMs, with a 2:1 ratio of local DRAM to CXL memory, matching the default setting of hardware tiering. Similar to hardware tiering without Memstrata, TPP does not move memory across VMs. The open-sourced TPP has an issue that wastes some local memory because it allocates local memory only from the `NORMAL` memory zone [30]. We have fixed this issue to enable TPP to perform better.

6.1 Performance Benefits

We assume a workload mix with about $\frac{1}{6}$ of workloads being outliers with hardware tiering, as our results show that 20% of the web, big data, and ML workloads experience more than 5% slowdown with hardware tiering (§3.2). As our prototype offers only 128 GB local memory, we must scale down the set of workloads typical for a large server. We scale to six VMs, typically with a single outlier workload[4]. We also consider the less likely scenarios of 2/6 and 4/6 outliers, as well as

[4] If compute servers indeed were to only host six VMs, scenarios with multiple outliers would be common. However, we seek to represent a scaled-down typical compute server with large VM counts (§2.2). Due to large-number effects most servers will thus have a $\frac{1}{6}$ ratio of outlier workloads. One can also integrate our slowdown estimator (§4.2) into the VM scheduler to explicitly prevent colocating many outliers (§7).



(a) 2 outliers with 33% dedicated local memory **(b)** 4 outliers with 33% dedicated local memory **(c)** 4 outliers with 67% dedicated local memory

Figure 12: Application slowdown when the workload combination contains 2 or 4 outliers.

dynamic VM arrivals. We start each VM with 33% dedicated local memory pages and 67% hardware-tiered pages.

Common cases. As web and data workloads are the dominant workload categories at Azure [98], we choose the following workload mixes to evaluate Memstrata:

1. **Web-heavy**: 4 web, 1 data, and 1 outlier.
2. **Data-heavy**: 1 web, 4 data, and 1 outlier.
3. **Balanced**: 2 web, 2 data, 1 others, and 1 outlier.

For each workload mix, we generate two workload combinations from our workload set. Each combination starts the six workloads simultaneously at the beginning.

Figure 10 shows the slowdown across the entire run. Under TPP, there are significantly more outliers than with hardware tiering, despite TPP having an unfair advantage: TPP has visibility inside the VM, and knows which pages are being used at the 4 KB granularity. Consequently, TPP can place the entire working set into local DRAM if its size does not exceed the local DRAM size. Such visibility assumes guest cooperation and is not compatible with design goal #1 (§2.2).

Interestingly, TPP and hardware tiering sometimes have different outliers. We observe that TPP achieves minimal slowdown whenever the working set can fit into local memory (e.g., SPEC's `619.lbm_s` and FASTER with uniform YCSB A), which is the target use case of TPP [84]. However, if the working set is too large, TPP experiences severe thrashing, causing massive TLB invalidations and page faults due to frequent page migration. For example, in an extreme case where TPP causes `pr-web` to have a 295% slowdown (Figure 10a), TPP migrates memory at 22 GB/s in a 32 GB VM.

This thrashing issue arises because TPP uses NUMA balancing hints [29] to choose promotion candidates and is not aware of the global memory access distribution. We repeat the single-application performance study in §3.2 with TPP. The results show that 17% of the workloads experience more than 50% slowdown with TPP because of thrashing (Figure 11). Although software tiering can measure the global access distribution and only migrate pages when the distribution is skewed to avoid thrashing, the CPU overhead of such global telemetry is prohibitive without guest cooperation, due to the lack of spatial locality in the guest physical memory address space (§2.3). In addition, even with global telemetry, the larger page sizes used with virtualization still make software tiering less effective, as they may average out the skewness in memory access distribution (§2.3).

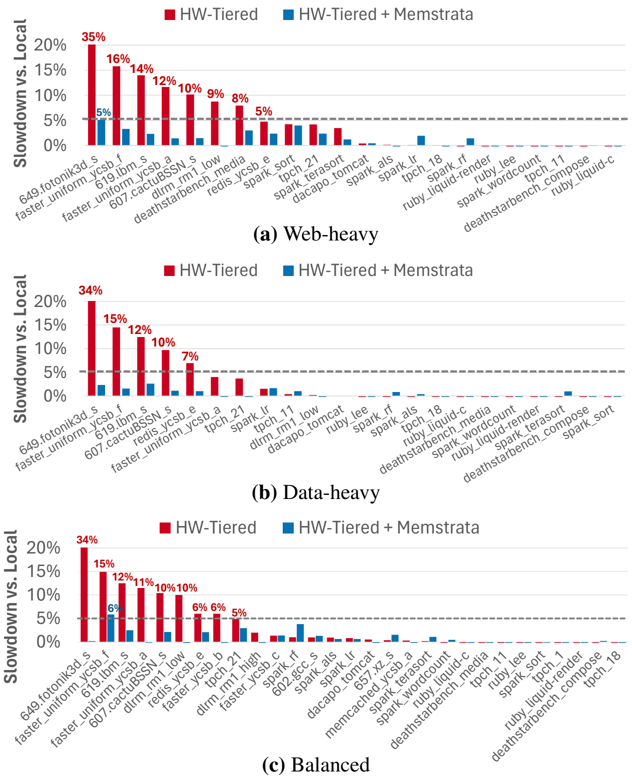
In summary, the comparison with TPP matches the results of recent work [76] indicating that hardware-based memory tiering's low overhead and cacheline-level granularity typically provide superior performance to software-based tiering. Therefore, in the rest of our experiments we focus on comparing Intel® Flat Memory Mode with and without Memstrata.

For all six experiments in Figure 10 Memstrata is able to significantly reduce the slowdown experienced by the outlier application to near 5% or less, with minimal impact to the other non-memory-sensitive applications. The max CPU overhead of Memstrata across all workload combinations is 4% of a single core, including running the ML model. The max memory overhead of Memstrata is 110 MB. The results show that Memstrata can accurately identify the outlier VM and migrate dedicated local memory pages to reduce its slowdown, without affecting the performance of other VMs.

Higher outlier ratio. To understand the limits of Memstrata, we consider a server that hosts a disproportionate ratio of outlier workloads. We consider two combinations: one with two outliers (Figure 12a), and another with four outliers (Figure 12b). In both experiments, Memstrata significantly improves outlier performance. However, is not able to reduce the slowdown for all outliers to below 5% when four outliers exist. This is because 33% of dedicated local DRAM (i.e., 26.4 GB) is insufficient to accommodate the memory needs of four outlier VMs (56 GB). We verify this by repeating the experiment with 67% dedicated local DRAM. In this configuration Memstrata removes all the outliers (Figure 12c).

Dynamic VM arrivals. We evaluate Memstrata in a more complex setting where VMs are continuously launched and terminated. We again use the three workload combinations described above. Whenever a workload of one type finishes, we start a new VM with a workload selected from the same type. The experiment keeps running until all the workloads have been run at least once. We measure the application performance with and without Memstrata. For the workloads that have finished multiple times, we report its average performance across all the completed runs.

Figure 13 shows that Memstrata can significantly reduce the slowdown of the outliers in such dynamic environments under all three workload mixes. The results demonstrate that Memstrata's online outlier detection can identify the outliers on-the-fly and dynamically migrate dedicated local memory pages to reduce their slowdown. We conclude that a combi-



(a) Web-heavy

(b) Data-heavy

(c) Balanced

Figure 13: Application slowdown in realistic environments with three different workload mixes.

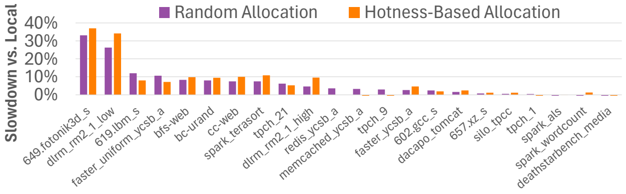
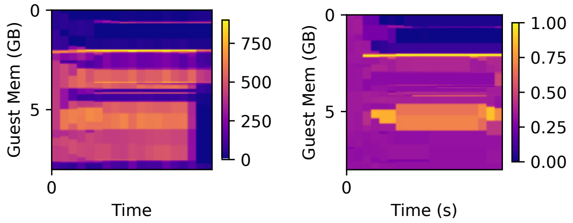


Figure 14: Slowdown of random and hotness-based page allocation.



(a) Memory accesses **(b)** Dedicated pages

Figure 15: Distributions of (a) memory accesses and (b) dedicated pages of `619.lbm_s` in the guest physical address space with hotness-based dedicated page allocation. In (a), the color represents #accesses captured by DAMON; in (b), the density of dedicated pages.

nation of Intel® Flat Memory Mode and Memstrata enables server memory capacity to be expanded by 1.5× using CXL at a minimal performance impact to applications.

6.2 Dedicated Memory Page Allocation

To understand how the allocation of dedicated local memory pages within a VM affects performance, we compare random page allocation (the default) with a hotness-based approach. We study a representative set of 22 workloads from different categories. With random allocation, the 33% of dedicated pages are randomly allocated to the VM when it is launched. With the hotness-based approach, the dedicated pages are also allocated randomly at launch, but Memstrata's dynamic allocator migrates them to hot guest physical regions based on the information provided by DAMON [11]. To measure the best-case performance of the hotness-based approach, we preserve the guest memory's spatial locality by always starting in a fresh VM.

Figure 14 shows the slowdown of both random and hotness-based page allocation. The hotness-based approach provides only marginal benefits and even causes worse slowdowns for some workloads because of its overhead. Figure 15 presents the hotness information recorded by DAMON and how the page allocator moves dedicated pages within the guest physical address space for `619.lbm_s`. Although it migrates dedicated pages to the hot regions identified by DAMON, the improvement is still limited. This is because to track memory accesses, DAMON must clear PTE access bits, resulting in expensive TLB shootdowns that offset its benefits.

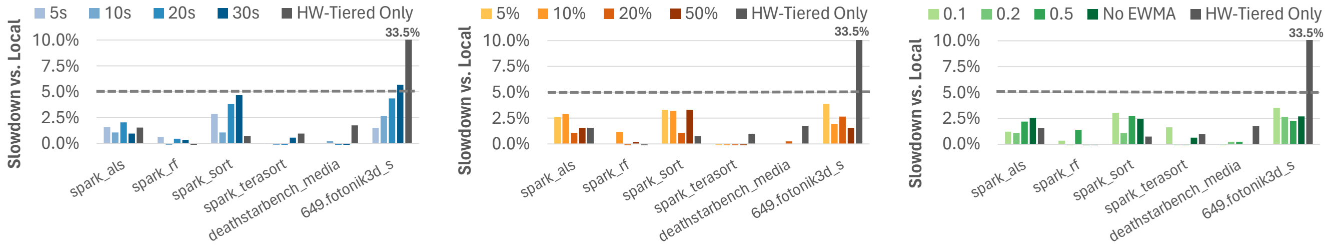
6.3 Sensitivity Analyses

Memstrata has three parameters: `timeInterval` and `stepRatio`, which control the aggressiveness of page migration, and the EWMA constant, which smoothes short-term variations of performance metrics. Figure 16 plots Memstrata's sensitivity to the three parameters using the same workloads as Figure 10e. With a lower `timeInterval` or a higher `stepRatio`, Memstrata migrates dedicated local memory pages to the outlier VM more quickly and achieves lower slowdowns (Figure 16a and Figure 16b). The default parameters (i.e., 10 s `timeInterval` and 10% `stepRatio`) have performance similar to the optimal one in this experiment, but are less aggressive and can avoid large performance fluctuations. Memstrata is not sensitive to the EWMA constant (Figure 16c).

7 Discussion

Non-virtualized environments. Most of Memstrata's components can be readily applied to non-virtualized environments. For example, per-process performance event tracking is already supported by Linux, and page migration mechanisms for both VMs and normal processes are also supported.

However, the page coloring implementation needs to be modified for non-virtualized settings. Unlike VMs, whose memory sizes do not typically change during their lifetimes, processes commonly have dynamic memory footprints. Memstrata statically allocates a fixed number of colors during VM creation, which is insufficient for processes with dynamic memory demands. Therefore, we need to augment the page coloring mechanism to support on-demand color allocation. In addition, as a process continuously allocates and frees memory, allocated colors may have numerous unused pages (e.g.,



(a) `timeInterval` (s) **(b)** `stepRatio` (%) **(c)** EWMA constant (α)

Figure 16: Sensitivity analyses of Memstrata under a data-heavy workload combination.

if a process allocates a large amount of memory but then frees half of it later). Such fragmentation can lead to insufficient free colors for new memory allocations. We therefore need to implement a compaction mechanism to reclaim colors from processes exhibiting significant fragmentation.

Intel® Flat Memory Mode with other memory ratios. In principle, Intel® Flat Memory Mode could support other ratios between local memory and CXL memory such as 1:2 or 1:4. We are not prepared to discuss whether such ratios will be available in a future version. There are also associated costs such as requiring more tag bits for bookkeeping.

Detecting outliers at the VM scheduler. Similar to Pond [78], the VM scheduler can correlate historical performance event measurements with a new VM allocation request by matching the customer ID, VM type, and location. Based on the historical information, the scheduler can perform an initial outlier detection to decide the memory type for the new VM. Additionally, the VM scheduler can also run online outlier detection after the VM is allocated and can live migrate the VM if the initial outlier detection proves to be inaccurate.

Adapting to other slowdown thresholds. Adapting to slowdown thresholds other than 5% requires retraining the random forest model using the performance events and the corresponding slowdowns of various workloads. Since the random forest model is lightweight and does not require any GPU, retraining the model incurs only minimal overhead and can be completed within a few seconds.

8 Related Work

Most prior work relies on software to place data across memory tiers [45, 70, 99, 100, 100], whereas Memstrata combines hardware tiering with a lightweight software layer. HeMem [89] and MEMTIS [74] are recent systems that use Intel PEBS to track memory accesses. Unfortunately, PEBS is not compatible with virtualized environments. In addition, unlike MEMTIS, which balances the TLB benefits of huge pages with the granularity of data placement by dynamically splitting huge pages, Memstrata achieves both low TLB cost and fine-grained data placement without sacrificing either. TPP [84] relies on LRU and NUMA balancing hints [29] to track memory accesses, but incurs high slowdowns (§6.1) and significant CPU overhead (§2.3).

Three prior systems explore software-managed tiered memory in multi-tenant environments. Unfortunately, they are not available for comparison. TMTS [61] is a memory tiering system deployed in Google's datacenters. We believe TMTS is overly conservative and requires large amounts of local memory. Pond [78] statically places VMs into a CXL-based memory pool based on predictions of slowdowns. Pond uses VM live migration to mitigate outliers, which impacts VM performance and thus must be applied conservatively. vTMM [90] is a dynamic software tiering memory management system for VMs. We believe vTMM suffers from overheads similar to other software-based systems (§2.3). Memstrata differs from all three systems due to its unique combination of hardware and software tiering in the same system.

2LM is a hardware-managed tiered memory system for Intel® Optane™ NVM, using DRAM as an inclusive direct-mapped cache of NVM. In contrast, Intel® Flat Memory Mode uses exclusive caching, and 2LM lacks the cross-VM isolation provided by Memstrata. Other hardware approaches have been proposed for DRAM and high-bandwidth memory [68, 75, 85, 103]; some Intel processors support a hardware-managed "cache mode" that uses HBM as a cache for DRAM [55, 59].

9 Conclusions

We presented a new hardware-based CXL tiering system, Intel® Flat Memory Mode, combined with a software stack, Memstrata. The combination provides performance similar to local DRAM across a wide range of workloads. Consequently, they enable expanding server memory capacity by 1.5× with minimal impact to performance. We believe there remain many open research challenges in deploying CXL in virtualized environments, including fairness in inter-VM resource allocation policies, guest cooperation for tiered memory, and using device-side hotness tracking to reduce page conflicts.

Acknowledgments

We would like to thank our shepherd, Marcos K. Aguilera, and the anonymous reviewers for their helpful comments. We also thank Phoebe Lu and Helen Chu for their contributions at the early stage of this work. This work was supported by an Intel gift and NSF grants CNS-1845853, CNS-2104243, CNS-2104292, and CNS-2143868.

References

[1] 2MB Large Memory Pages for Hypervisor and Guest Operating System. https://docs.vmware.com/en/VMware-Cloud-on-AWS/services/vmc-aws-performance/GUID-5658E64F-5606-4363-A18D-9E9175D107F0.html.

[2] AMD Bergamo and Genoa-X EPYC server CPUs crush the competition with sheer performance and efficiency dominance. https://wccftech.com/amd-bergamo-genoa-x-epyc-server-cpus-crush-competition-sheer-performance-efficiency-dominance/. [Accessed 11/14/2023].

[3] AMD, Meta are working on revolutionary tech that could recycle petabytes worth of RAM. https://www.techradar.com/pro/amd-meta-are-working-on-revolutionary-tech-that-could-recycle-petabytes-worth-of-ram-cxl-could-help-save-hyperscalers-tens-of-millions-of-dollars-while-improving-performance. [Accessed 11/14/2023].

[4] AMD Research Instruction Based Sampling Toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit.

[5] ARM - Performance Monitor Unit. https://developer.arm.com/documentation/ddi0500/d/performance-monitor-unit.

[6] AWS EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[7] Azure / Virtual Machines / General purpose virtual machine sizes. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-general.

[8] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[9] CXL Members. https://www.computeexpresslink.org/members.

[10] CXL Smart Memory Controllers. https://www.asteralabs.com/products/cxl-memory-platform/.

[11] DAMON: Data Access MONitor. https://www.kernel.org/doc/html/v5.19/vm/damon/index.html.

[12] Google Cloud Platform / General-purpose machine family for Compute Engine. https://cloud.google.com/compute/docs/general-purpose-machines.

[13] HiBench Suite. https://github.com/Intel-bigdata/HiBench.

[14] Intel Announces 288-Core Sierra Forest CPU, 5th-Gen Xeon. https://www.tomshardware.com/news/intel-announces-288-core-processor-5th-gen-xeon-arrives-december-14.

[15] Intel FPGA Compute Express Link (CXL) IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html.

[16] Intel Performance Counter Monitor. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.

[17] Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[18] Intel® Optane™ Persistent Memory Start Up Guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[19] Intel® Memory Latency Checker (Intel® MLC). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html.

[20] Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[21] Kernel Self Protection Project/Recommended Settings. https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommended_Settings.

[22] Leo CXL Smart Memory Controllers. https://www.asteralabs.com/products/leo/leo-cxl-memory-connectivity-controllers/.

[23] Memcached - A distributed memory object caching system. http://memcached.org.

[24] Memory management notifiers. https://lwn.net/Articles/266320/.

[25] Meta AMD CXL 2.0 Memory Expansion Demo at OCP Summit 2023. https://www.servethehome.com/meta-amd-cxl-2-0-memory-expansion-demo-at-ocp-summit-2023/.

[26] Microchip introduces new CXL smart memory controllers for data center computing enabling modern CPUs to optimize application workloads. https://www.microchip.com/en-us/about/news-releases/products/cxl-smart-memory-controllers. [Accessed 11/14/2023].

[27] Micron CZ120 memory expansion module. https://www.micron.com/solutions/server/cxl.

[28] Montage Technology Delivers the World's First CXL Memory eXpander Controller. https://www.montage-tech.com/Press_Releases/20220506.

[29] NUMA balancing: optimize memory placement for memory tiering system. https://lwn.net/Articles/849095/.

[30] [PATCH 0/5] Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/all/cover.1637778851.git.hasanalmaruf@fb.com/.

[31] [PATCH] AMD perf PMU events for AMD Family 17h. https://lore.kernel.org/lkml/3ee15066-429e-b0f2-1255-aab100fad472@suse.cz/.

[32] Pathfinding Cloud Architecture for CXL with Dan Ernst of Microsoft Azure. https://gestaltit.com/all/stephen/pathfinding-cloud-architecture-for-cxl-with-dan-ernst-of-microsoft-azure-utilizing-tech-4x10/.

[33] PostgreSQL Database Management System. https://www.postgresql.org/.

[34] [QEMU-devel] [PATCH] Call MADV_-HUGEPAGE for guest RAM allocations. https://lists.nongnu.org/archive/html/qemu-devel/2012-10/msg01012.html.

[35] Recording Hardware Performance (PMU) Events. https://learn.microsoft.com/en-us/windows-hardware/test/wpt/recording-pmu-events.

[36] Redis, an in-memory data structure store. http://redis.io.

[37] Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://semiconductor.samsung.com/news-events/news/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module/.

[38] Samsung's memory-semantic CXL SSD brings a 20x performance uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift. [Accessed 11/14/2023].

[39] Shopify/yjit-bench: Set of benchmarks for the YJIT CRuby JIT compiler and other Ruby implementations. https://github.com/Shopify/yjit-bench.

[40] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[41] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[42] TPC Benchmark H (TPC-H). https://www.tpc.org/tpch/.

[43] VMware vCloud NFV OpenStack Edition / Tuning vCloud NFV for Data Plane Intensive Workloads / Huge Pages. https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tunning/GUID-1F05987F-012B-4BC4-9015-CDE3C991C68C.html.

[44] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[45] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 25–32, 2023.

[47] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. Virtual machine allocation with lifetime predictions. *Proceedings of Machine Learning and Systems*, 5, 2023.

[48] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[49] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design tradeoffs in CXL-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[50] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. 42(2):26–35, mar 2008.

[51] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[52] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[53] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[54] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[55] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. BATMAN: Techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems*, pages 268–280, 2017.

[56] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[57] Ampere Computing. AmpereOne 192-core CPU family product brief. https://amperecomputing.com/briefs/ampereone-family-product-brief. [Accessed 11/14/2023].

[58] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[59] Daniel DeLayo, Kenny Zhang, Kunal Agrawal, Michael A Bender, Jonathan W Berry, Rathish Das, Benjamin Moseley, and Cynthia A Phillips. Automatic HBM management: Models and algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 147–159, 2022.

[60] ONNX Runtime developers. ONNX runtime. https://onnxruntime.ai/, 2021. Version: x.y.z.

[61] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[63] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. ACT: Designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 784–799, 2022.

[64] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[65] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. A case against hardware managed DRAM caches for NVRAM based systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, 2021.

[66] Intel. 4th gen Intel Xeon scalable processors. https://download.intel.com/newsroom/2023/data-center-hpc/4th-Gen-Xeon-Scalable-Product-Brief.pdf, 2023.

[67] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan

Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. Optimizing cpu performance for recommendation systems at-scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[68] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked DRAM cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE, 2014.

[69] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, nov 1992.

[70] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[71] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[72] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[73] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1–1. IEEE, 2016.

[74] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[75] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless DRAM cache. *ACM SIGARCH computer architecture news*, 43(3S):211–222, 2015.

[76] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, Boston, MA, July 2023. USENIX Association.

[77] Philip Levis, Kun Lin, and Amy Tai. A case against CXL memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[78] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[79] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and portable virtual NVMe storage on arm SOCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[80] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and Ponnuswamy Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[81] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[82] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-channel attacks on Optane persistent memory. In *32nd USENIX Security Symposium 2023*. USENIX Association, 2023.

[83] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and Misconceptions Around Reducing Carbon Embedded in Cloud Platforms. In *HotCarbon Workshop*, 2023.

[84] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Sparsh Mittal and Jeffrey S Vetter. A survey of techniques for architecting DRAM caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863, 2015.

[86] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[87] Dave Patterson. 10 lessons from a decade of TPUs and ML's carbon footprint. https://www.youtube.com/watch?v=--z1cmq1BCw, 2023.

[88] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, et al. Renaissance: A modern benchmark suite for parallel applications on the JVM. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 11–12, 2019.

[89] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vTMM: Tiered memory management for virtual machines. EuroSys '23, page 283–297, New York, NY, USA, 2023. Association for Computing Machinery.

[91] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. An introduction to the Compute Express Link (CXL) interconnect. *ACM Computing Surveys (CSUR)*, 2024.

[92] Shigeru Shiratake. Scaling and performance challenges of future DRAM. In *2020 IEEE international memory workshop (IMW)*, pages 1–3. IEEE, 2020.

[93] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE, 2008.

[94] Lisa Su. Amd unveils workload-tailored innovations and products at the accelerated data center premiere. https://www.amd.com/en/press-releases/2021-11-08-amd-unveils-workload-tailored-innovations-and-products-the-accelerated, November 2021.

[95] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007.

[96] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[97] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[98] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *ISCA*, June 2024.

[99] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[100] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[102] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. PACT '14, page 381–392, New York, NY, USA, 2014. Association for Computing Machinery.

[103] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–14, 2017.

[104] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing. In *USENIX NSDI*, 2012.

[105] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.

Harvesting Memory-bound CPU Stall Cycles in Software with MSH

Zhihong Luo
UC Berkeley

Sam Son
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley & ICSI

Abstract

Memory-bound stalls account for a significant portion of CPU cycles in datacenter workloads, which makes harvesting them to execute other useful work highly valuable. However, mainstream implementations of the hardware harvesting mechanism, simultaneous multithreading (SMT), are unsatisfactory. They incur high latency overhead and do not offer fine-grained configurability of the trade-off between latency and harvesting throughput, which hinders wide adoption for latency-critical services; and they support only limited degrees of concurrency, which prevents full harvesting of memory stall cycles.

We present MSH, the first system that transparently and efficiently harvests memory-bound stall cycles in software. MSH makes full use of stall cycles with concurrency scaling, while incurring minimal and configurable latency overhead. MSH achieves these with a novel co-design of profiling, program analysis, binary instrumentation and runtime scheduling. Our evaluation shows that MSH achieves up to 72% harvesting throughput of SMT for latency SLOs under which SMT has to be disabled, and that strategically combining MSH with SMT leads to higher throughput than SMT due to MSH's capability to fully harvest memory-bound stall cycles.

1 Introduction

CPU cores are valuable resources in datacenter infrastructure. To meet the ever-growing computation demand, there have been extensive software efforts in *harvesting* idle CPU cycles and keeping cores fully utilized [7,43,52,88,94]. While differing in mechanisms, these works share a similar harvesting scheme: "scavenger" instances (*e.g.,* spot VMs, batch jobs) temporarily run on cores that primary instances are not actively using. Their common performance goal is to have scavenger instances fully utilize the idle cycles without slowing down primary instances. Minimizing negative performance impacts is particularly important for latency-critical services as their increased latencies directly affect user experience.

Unlike prior efforts that harvest cores that are idle for a relatively long period of time, *e.g.,* allocated but unused cores of the primary VM, we focus on memory-bound CPU *stall* cycles. These are cycles that cores transiently stall while waiting for memory accesses to finish. Although each lasts only a few hundred nanoseconds, memory-bound stalls can happen frequently and account for a significant portion of CPU cycles [10,23,45,78]: more than 60% for some widely-used modern applications, which implies substantial benefits harvesting these stall cycles. However, the current hardware harvesting mechanism, simultaneous multithreading (SMT), is unsatisfactory. First, SMT is known to likely lead to significantly *increased latencies*, as it focuses solely on multiplexing instruction streams to best utilize core resources [37, 74, 83, 84]. Moreover, SMT does not allow fine-grained *control* over the tradeoff between primary latency and scavenger throughput, which is needed to maximize CPU utilization under a latency SLO. As a result, for latency-critical services, a common compromise is thus to avoid using SMT for better performance, at the cost of wasting stall cycles [18,19,55,69]. Lastly, there are cases where SMT can not *fully harvest* memory-bound stall cycles: modern CPUs often support only limited degrees of concurrency (*e.g.,* 2 threads per physical core in the case of Intel's Hyperthreading), which are insufficient when concurrent threads frequently incur cache misses [44,45,72].

In view of the significance of memory-bound CPU stalls and the drawbacks of SMT as the hardware harvesting mechanism, our goal is to design a system that harvests these stall cycles in software. This system should meet several requirements. First, it should be *transparent* to applications and require no additional rewriting efforts from developers. As a result, it will resemble SMT in terms of being conveniently applicable to any code, including legacy code. The other requirements then demand improving upon the drawbacks of SMT. Specifically, it should *efficiently* and *fully* harvest the memory-bound stall cycles, and it should do so while introducing *minimal* latency overhead to the primary instance.

A recent proposal [57] discusses the possibility of transparently hiding the latency of cache misses in software with the combination of light-weight coroutines [25,28,64,79] and sample-based profiling [17,47,82]. The former allows interleaving of primary and scavenger coroutines with a switching overhead much smaller than traditional threads of executions like processes and kernel threads; whereas the latter makes it possible to do it transparently, as we could identify likely locations of cache misses via profiling. This is a key realization that our work builds upon. However, there remains to be a set of challenges toward building a software system that harvests memory-bound CPU stall cycles and meets the aforementioned requirements. First, to improve harvesting efficiency, we have to minimize the amount of register savings and restorations for each yield, while ensuring the correctness of program executions. Second, to introduce minimal latency overhead, scavengers need to yield back the core soon after they have consumed enough stall cycles, which is challenging given that programs have complex and dynamic control flows. Third, to fully harvest stall cycles, we need to detect when a higher degree of concurrency is needed and properly interleave the executions of multiple scavengers. Lastly, it is challenging to transparently interleave scavenger executions with a primary binary that has an internal threading structure.

To overcome these challenges, we present Memory Stall Software Harvester (MSH), the first system that transparently and efficiently harvests memory-bound CPU stall cycles in software. MSH makes full use of stall cycles while incurring only minimal latency overhead. MSH fulfills all the requirements with a novel co-design of *profiling, program analysis, binary instrumentation and runtime scheduling*. To use MSH, users simply provide unmodified primary binaries and a pool of scavenger threads, and MSH takes care of running scavenger threads with stall cycles of the primary binaries.

Internally, MSH operates in two logical steps. First, after profiling the primary and scavenger code, MSH statically instruments them at the binary level, by leveraging information obtained via profiling and program analysis. Specifically, for both primaries and scavengers, MSH inserts a prefetch instruction followed by yielding to either a primary or a scavenger coroutine (configured in runtime, discussed below), before selected load instructions that frequently incur cache misses according to profiled data. In addition, MSH places additional yields in scavengers to ensure that they timely relinquish their core. The first two of the aforementioned challenges are resolved in this step. For the primary binaries, MSH carries out various optimizations to reduce the amount of register savings and restorations for each yield by analyzing register usage and program structures. For the scavenger, MSH conducts a forward data flow analysis that also takes in profiled data to decide additional yield points, so that the distance between consecutive yields is bounded to a configurable threshold.

In the second logical step, when executing a primary binary, MSH sets up and dynamically assigns scavengers to active primary threads. The last two challenges regarding scavenger scheduling and concurrency scaling are tackled in this step. MSH intercepts function calls that change the status of primary threads and efficiently adjusts the scavenger assignment. This allows MSH to transparently schedule scavengers on top of the primary's threading structure. To support on-demand concurrency scaling, MSH performs two operations: assigning multiple scavengers to a primary thread and configuring scavengers so that they yield to the right target. For the former, MSH decides the number of scavengers assigned to a primary thread by estimating and bounding the likelihood of not full harvesting stall cycles. For the latter, MSH instruments yields in scavengers that are close to each other to yield to the next scavenger instead of the primary thread. MSH's runtime then takes care of correctly setting up the targets of these yields.

We implement MSH's offline parts on top of Bolt [67], an open-source binary optimizer built on the LLVM framework, and MSH's runtime as a user-level library[1]. We evaluate MSH with unmodified syntactic and real applications and show that MSH is general enough to harvest stall cycles from all of them. Compared with SMT, MSH offers superior harvesting performance in three aspects: first, MSH incurs minimal latency overhead and achieves up to 72% harvesting throughput of SMT, for latency SLOs under which SMT has to be disabled. Second, as a configurable software solution, MSH enables users to have fine-grained control over the tradeoff between primary latency and scavenger throughput. Third, MSH can fully harvest memory-bound stall cycles via concurrency scaling, achieving up to 2x higher throughput than SMT when scavengers frequently stall. Moreover, we show that by strategically combining MSH with SMT, one could achieve higher throughput than SMT due to MSH's ability to fully harvest memory-bound stall cycles. Lastly, we extensively evaluate MSH's main components and show that they play a vital role in achieving MSH's superior performance.

In summary, the contributions of this paper are: (i) a transparent and efficient approach to harvest memory-bound CPU stall cycles in software; (ii) the detailed design and implementation of a system (MSH) based on this approach, which involves a co-design of profiling, program analysis, binary instrumentation, and runtime scheduling; (iii) an evaluation with real applications showing that compared with SMT, MSH can deliver high scavenger throughput under stringent primary latency SLOs and fully harvest memory-bound stall cycles. In addition to presenting the design, implementation, and evaluation of MSH, we extensively discuss other related aspects in §8. These include isolation mechanisms that can be integrated with MSH to ensure memory safety, hardware support that can enhance MSH's performance and so on. Our hope here is to motivate greater efforts in delivering these critical aspects.

2 Background and Motivation

Memory-bound stalls: Memory-bound stalls, where cores stall and wait for memory accesses to finish, were reported to be a dominant source of CPU overhead for datacenter workloads. To see this, we perform a top-down analysis [90] on two latency-critical applications. This analysis classifies CPU pipeline slots into four categories: retiring, frontend-bound, bad speculation and backend-bound. The last three correspond to different overhead, and backend-bound stalls can be further divided into core-bound or memory-bound stalls. Our

[1] MSH is publicly available at https://github.com/sosson97/msh.

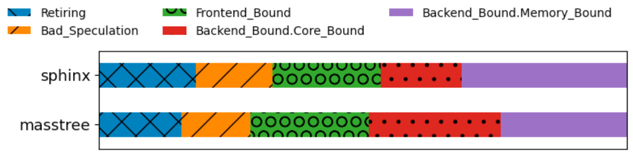
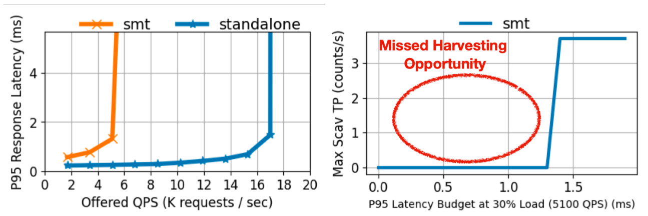


Figure 1: Top-down analysis of Sphinx and Masstree; memory stalls account for 25% and 31% of cycles respectively.



(a)

(b)

Figure 2: (a) P95 latency of Masstree when running by itself vs. co-locating with a Scan scavenger; (b) SMT is unable to harvest stall cycles under low latency SLOs.

analysis confirms the dominance of memory-bound stalls, as they account for 25% and 31% of total cycles for Masstree and Sphinx respectively (Figure 1). While there have been extensive efforts on *reducing* memory stalls, it is generally infeasible to eliminate them (§7). In this work, we focus on the alternative approach of *harvesting* these stall cycles to execute useful work, where simultaneous multithreading (SMT) is the representative hardware mechanism.

Drawbacks of SMT: However, SMT, as a harvesting mechanism, suffers from three main drawbacks[2] that we next show:

- **Latency overhead:** SMT focuses solely on multiplexing instruction streams to best utilize CPU cores. As a result, it significantly increases the primary latency if the scavenger creates notable *contention* on core resources. This is problematic as it is common to co-locate latency-critical tasks that have stringent latency SLOs, with best-effort tasks that are resource-hungry. To see the latency overhead of SMT, we measure the latency of Masstree while running a synthetic Scan scavenger on its sibling cores. Scan is a representative of contending scavengers: by iterating a 4MB array and computing the sum, it consumes L1/L2 caches and core resources like ALU. As shown in Figure 2-(a), compared with running on dedicated cores, harvesting stall cycles via SMT leads to 92x higher latency of Masstree at 40% load. Such a behavior is widely observed in prior studies, thus it is common to avoid using SMT for latency-critical services at the cost of wasting cycles.

- **Lack of Configurability:** Related to the large latency overhead, another drawback of SMT that hinders its uses for latency-critical services is the lack of fine-grained configurability. Given a latency SLO, what is needed to maximize CPU utilization is a knob that controls the *extent* of resource sharing and hence the tradeoff between primary latency and scavenger throughput. However, with SMT, one can only decide whether to turn it on or off, which is too coarse-grained to be useful. To see this, we compute the maximum achievable Scan scavenger throughput under different Masstree latency SLOs for the experiment above. Here we set the SLO to be the latency under 30% load. An ideal mechanism should gracefully harvest cycles proportional to the latency budget given. In contrast, as shown in Figure 2-(b), with SMT, one has to turn off SMT and effectively achieve zero scavenger throughput when the latency SLO is lower than SMT latency. Even after SMT is on, it can not harvest more cycles when looser latency SLOs are given. Neither of these two ends is desirable.

- **Incomplete harvesting:** Lastly, SMT often can not fully harvest memory-bound stall cycles, especially when concurrent threads frequently incur cache misses. This is because the mainstream 2-wide SMT does not have sufficient degrees of concurrency to harvest the bulk of memory stalls. Note that while increasing the width of SMT helps with this issue, it requires dedicating more hardware resources and worsens the already problematic latency overhead issue.

We aim to design a software system that harvests memory-bound stall cycles, is as generally applicable and convenient to use as SMT, and improves upon the drawbacks of SMT.

Software opportunities: There are two capabilities a software mechanism needs for harvesting memory stall cycles: (i) transparently detecting the presence of memory stalls and (ii) efficiently interleaving the executions of primaries and scavengers. The former is challenging, because cache misses are not exposed to software, and manually identifying stalls is burdensome and error-prone. The latter requires much smaller switching overhead than traditional threads of execution like kernel threads. A recent proposal [57] discusses the opportunity of enabling these two capabilities via a combination of light-weight coroutines and sample-based profiling:

- **Sample-based profiling:** By using hardware performance counters in modern CPUs, such as Intel's PEBS [3] and LBR [47], one could profile binaries with no special build and negligible run time overhead. Thanks to these merits, sample-based profiling has been widely used in production for profile-guided optimizations (PGO) [17, 30, 66–68].

- **Light-weight coroutines:** Context switches of coroutines are orders of magnitudes cheaper than traditional threads of execution. This is because as a user-space mechanism within a single process, coroutine context switch requires no system calls nor changes to virtual memory mappings.

Building on these two techniques, MSH is the first software system that transparently and efficiently harvests memory-bound stall cycles. Next, we present an overview of MSH.

[2] These drawbacks apply to SMT of most modern processors (*e.g.,* Intel's and AMD's), with IBM Power as an exception, discussed further in §7.

3 MSH Overview

In this section, we discuss MSH's overarching goals, deployment scenarios, high-level approach as well as overall flow.

Goal: Our goal is to transparently harvest memory-bound stall cycles from any application, while overcoming SMT's performance limitations. We thus distill four requirements that MSH as a software harvesting system should meet:

- **Transparent:** The system should be transparent to applications. It thus requires no rewriting effort from developers and is applicable to any code, including legacy code.
- **Efficient:** The system should efficiently utilize the stall cycles for scavenger executions, which demands extremely low overhead from the harvesting machinery.
- **Latency-aware:** The system should incur minimal latency overhead and allow fine-grained control over the trade-off between primary latency and scavenger throughput.
- **Full-harvesting:** The system should fully harvest stall cycles by interleaving sufficient scavenger executions, especially when scavengers also incur frequent cache misses.

Deployment scenario: System operators can use MSH to harvest stall cycles of any application written in compiled languages. MSH handles scavenger's offline instrumentations and runtime executions. MSH assumes that it is safe to run these scavengers alongside the primaries [92], *e.g.,* they are crash-free and access memory safely. Ensuring safety properties with techniques like verification and information flow control [14, 35, 62] is left to future work. MSH can be seamlessly integrated with existing profiling systems deployed for PGO [17, 30, 68]. MSH is well suited for when latency-critical and best-efforts tasks are co-located in the same machine, a common arrangement in production [27, 55, 61, 93]. In this case, latency-critical tasks serve as the primary, whose stall cycles are harvested for the best-effort tasks.

Approach: MSH uses a novel co-design of binary instrumentation, profiling, program analysis, and runtime scheduling, each of which plays a role in meeting the requirements above:

- **Binary instrumentation:** MSH instruments primaries and scavengers so that they are amenable to stall cycle harvesting. Operating at the binary level provides visibility of low-level information, *e.g.,* register usage and basic block control flows, which is needed by MSH's program analysis.
- **Profiling:** With sample-based profiling, MSH decides locations to harvest stall cycles without requiring efforts from developers. Profiling also allows MSH to use dynamic information, *e.g.,* basic block latency and branching probability, to achieve high accuracy in its program analysis.
- **Program analysis:** MSH leverages program analysis to achieve efficiency, full-harvesting and latency-awareness. For efficiency, MSH minimizes the amount of register savings and restorations for yields. For full-harvesting, MSH directs yields in scavengers that are close to each other to another scavenger. For latency-awareness, MSH bounds the latency between adjacent yields in scavengers.
- **Runtime scheduling:** MSH's runtime schedules scavenger executions on top of the primary's internal threading structure. It enables MSH to fully harvest stall cycles with available scavengers, by assigning multiple scavengers to a primary thread to scale up concurrency and migrating scavengers from blocked primary threads to active ones.

Overall Flow: MSH performs both offline and run-time operations (Figure 3). In the offline phase, MSH transforms the primary and scavenger binaries so that they are amenable to stall cycle harvesting. Specifically, MSH first profiles the binaries and obtains information needed by program analysis and later binary instrumentation: load instructions that incur cache misses, indicating where CPU stalls happen; basic block latencies and execution counts as well as branching probability, which are used by the primary and scavenger instrumentations. After profiling, MSH analyzes the binaries and extracts information that later guides the instrumentations. For each yield site, where a yield is inserted to harvest stall cycles of a delinquent load, MSH identifies a minimal amount of register savings and restorations that still ensures program correctness, by analyzing register usage and program structures. For scavengers, MSH conducts a data flow analysis to decide the locations of additional yields, so that the expected inter-yield latency is bounded. Most of the analysis is designed to be intra-procedural, the complexity of which thus scales only sublinearly with program sizes. Lastly, based on the analysis results, MSH instruments the binaries.

The instrumented primary binaries contain so-called "primary" yields to expose CPU stalls: each primary yield is inserted before a selected load instruction and prefetches the cache line before yielding to a scavenger. As for instrumented scavengers, they also contain primary yields before selected load instructions, with default yield targets being a primary thread. The special case is when primary yields are close to each other: the target of these "special" primary yields is set to another scavenger to scale up concurrency. Scavengers also contain so-called "scavenger" yields, which are placed to ensure that scavengers relinquish their cores in a timely manner. We present the design of primary and scavenger instrumentations in §4.1 and §4.2.

At runtime, MSH interleaves the executions of instrumented primaries and scavengers by dynamically assigning scavengers to active primary threads, which means that MSH does not require pre-determined or static pairings of primaries and scavengers. To do that, MSH tracks the status of primary threads by intercepting relevant function calls and adjusts scavenger assignment accordingly. When a new thread is created, MSH either steals the scavengers of a blocked thread or fetches scavengers from the scavenger pool. If a thread is blocked or ended, MSH marks its scavengers as stealable.

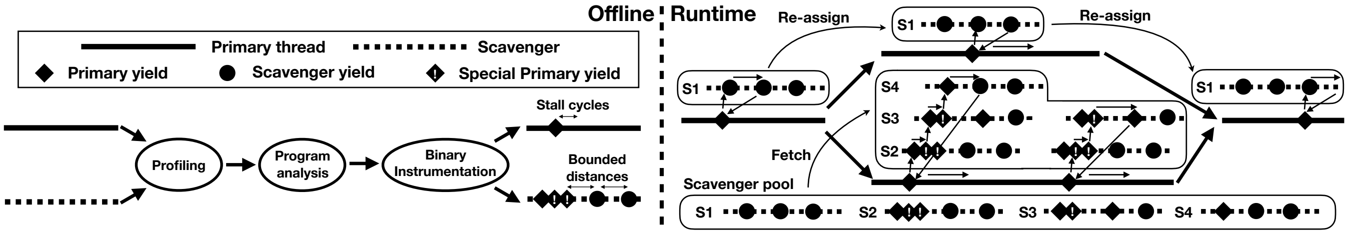


Figure 3: MSH system overview. Offline: MSH profiles and analyzes primaries and scavengers. It then instruments the primaries to yield control to scavengers at likely memory stall sites, with scavengers returning control to primaries within a bounded time. Runtime: MSH sets up a scavenger pool and dynamically assigns scavengers to each active primary thread.

When a thread later resumes, it will first attempt to reuse its previously assigned scavengers, before falling back to getting new scavengers like the thread creation case. Multiple scavengers could be assigned to a primary thread to scale up concurrency. MSH's runtime performs all these operations efficiently, and its design is later presented in §4.3.

4 Design

MSH consists of three components: primary instrumentation (§4.1), scavenger instrumentation (§4.2) and a runtime (§4.3).

4.1 Primary Instrumentation

Primary instrumentation allows MSH to prefetch and yield before load instructions that incur cache misses to expose stall cycles. This should be transparent – requiring no assistance from developers, and efficient – leaving most stall cycles for scavengers. MSH achieves transparency by selecting yield sites based on profiled data, and efficiency by minimizing register savings/restorations for each yield via program analysis.

Profile-guided yield instrumentation: MSH selects locations that both account for a significant portion of memory-bound stalls and have a high likelihood of L3 cache misses: the former indicates substantial stall cycles, and the latter allows less impact to the primary's latency. To support this, MSH obtains two pieces of information via profiling: load instructions with L2/L3 cache misses and execution counts of basic blocks.. MSH then adopts a two-step selection logic. First, MSH sorts load instructions whose cache miss rates are higher than a threshold by their frequencies. Second, MSH estimates the latency overhead for each load instruction by multiplying its frequency with its cache hit rate and the memory access latency. MSH then goes down the sorted list, includes a load instruction if the aggregate overhead falls below a provided bound, and skips otherwise. This selection logic maximizes harvesting opportunities by prioritizing frequent load instructions, while limiting the overall latency overhead. Both the cache miss threshold and overhead bound are configurable parameters that affect the tradeoff between primary latency and scavenger throughput (§6.4).

For each selected load instructions, MSH instruments a prefetch instruction for the same address, followed by a yield that consists of two parts: register savings/restorations and control passing. The former accounts for most of the yielding overhead, and as we will describe next, MSH minimizes it while ensuring correctness of program executions. For control passing, MSH instruments the primary to swap its instruction and stack pointer with the ones of an assigned scavenger that the primary reads from a per-thread data structure (§4.3). The instrumented code also reads a flag that indicates whether to bypass the yield and directly resumes. This allows the runtime to turn off stall cycle harvesting for instrumented primaries and avoid the latency overhead of scavenger executions.

Yield cost minimization: Minimizing the yield cost is important for two reasons. First, it improves harvesting efficiency: the less cycles spent on the yielding machinery, the more cycles available for executing scavengers while the primary stalls. Moreover, it reduces the latency impacts to the primary, especially when an instrumented load instruction results in a cache hit and only stalls for a short amount of time.

Register savings and restorations are the dominant cost for yields. MSH thus performs various optimizations to reduce them while ensuring correctness of the program executions. To avoid preserving every register, MSH first leverages register liveness analysis [71], a form of data-flow analysis that determines for each program point the set of "live" registers whose values will likely be used later. Given that register liveness is conservative, meaning that a register will be identified as live as long as there is any potential program path that may read its current value, by preserving live registers at the yield site, we are guaranteed to not violate the program correctness.

While saving only live registers reduces the yielding overhead, the cost saving is small for functions with non-trivial control flows, where most registers are considered live. To further reduce the cost, MSH builds on an observation: besides *what* registers to preserve, *where* these register savings and restorations take place also plays an important role in the yielding overhead. In particular, the naive approach of placing register savings and restorations at yield sites leads to *unnecessary* overhead. This is because there can be multiple yields between a definition of a register and its corresponding uses that repeatedly save and restore the register's value as the register is indeed live. To fix this, the key insight is to *align* register savings/restorations with register definitions/uses. Intuitively, if we were to save/restore the register at

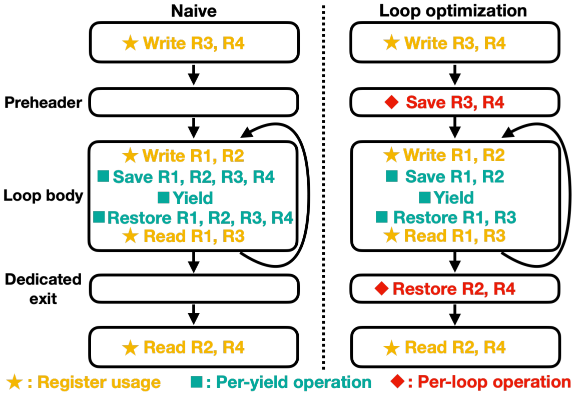


Figure 4: Loop optimization in primary instrumentation.

its definition/use sites, we can remove the redundancy due to having multiple yields in between the definition-use pairs, while still correctly preserving program semantics.

However, placing register savings/restorations at its definition/use sites for arbitrary program structures is highly complicated and potentially undesirable. Specifically, for correctness, one needs to identify all the definition sites, whose definitions are likely to reach the yielding point, as well as all the use sites that likely read these definitions. Instrumenting at all these scattered locations requires a substantial amount of work. Moreover, it is inevitable that some definition-use pairs have paths that do not go through the yield point. This means that there is register saving/restoring overhead even when the function does not yield, which could lead to overall increased overhead, if these cases happen frequently.

Instead of handling arbitrary program structures, MSH focuses on *loops*: it is often the case that a large portion of yields reside in loops, which make them valuable targets for optimizations. More importantly, the unique structure of loops allows MSH to perform *per-loop* register savings or restorations. As shown in Figure 4, most loops can be restructured to have a preheader and some dedicated exits: the former dominates the loop body whereas the latter post-dominate it. As a result, any paths traversing the loop will enter the preheader and leave one of the exits. MSH can thus simply place register savings and restorations at the preheader and exits, respectively, to ensure correctness for yields within the loop. Moreover, as long as more than one loop iteration goes through the yielding point, such a placement leads to strictly fewer register savings and restorations than the yield-site placement. In practice, this improvement is significant as the operation now happens once per loop instead of once per iteration. For registers that only have either uses or definitions within the loop body (R2 and R3 in Figure 4), MSH adopts a hybrid approach that places either saving or storing at the preheader/exit and the other at the yield site. To enable such loop optimizations, besides register liveness analysis, MSH performs reaching definition analysis to track the relevant definitions and uses for live registers, as well as loop simplification to transform feasible loops.

Besides when there are yields directly within a loop, MSH optimizes for another common case, where a function called within loops contains a single yield point. In particular, for a function that has unused callee-saved registers, we need to preserve values of these registers at the function boundary to abide by the calling convention. However, when such functions are called in loops, they incur redundant overhead due to per-iteration saving and restoration. To address this issue, MSH performs an optimization that we call “pseudo-inlining”: MSH effectively inlines the target function by creating a copy of the function, for which the values of unused callee-saved registers are not preserved, and redirecting calls in loops towards this copy. MSH then leverages its loop optimization technique to save and restore the values of these unused callee-saved registers at the loop granularity as much as possible. MSH ensures that the original copy complies with the calling convention, so that other calls to the function take place correctly. Pseudo-inlining thus enables loop optimizations as if the function were inlined, while being easy to implement and creating minimal code expansion since the copy is shared.

In summary, MSH is strategic about what registers to preserve and where operations take place. It achieves the former by identifying live registers and the latter by exploiting per-loop operations. This reduced yield cost then leads to lower primary latency and higher harvesting efficiency (§6.4).

4.2 Scavenger Instrumentation

Scavenger instrumentation allows full stall cycle harvesting, while incurring minimal latency overheads. To minimize latency overhead, MSH places scavenger yields to bound inter-yield distances. To fully harvest stall cycles, primary yields that are too close to each other are directed to another scavenger. Next, we describe the mechanism in detail.

Primary yields: MSH instruments yields for stalling load instructions within scavengers in the same way as primary instrumentation: identifying yield sites via profiling and adopting optimizations to reduce yield costs. By default, these primary yields relinquish the core back to the primary. The special case is when some yields are too close to each other to fully harvest stall cycles (*e.g.,* yields within tight loops). These special primary yields will continue to the next scavenger. To support this, the per-thread data structure managed by runtime contains two targets (*i.e.,* primary thread and next scavenger) for each scavenger (§4.3). Normal and special primary yields are thus instructed to read different targets.

Scavenger yields: With only primary yields, it could take arbitrarily long for scavengers to yield back. MSH thus bounds inter-yield distances via a data-flow analysis that (i) calculates the average distances between a basic block and the current set of scavenger yields and (ii) inserts yields if some distance is over the bound. Note that the accuracy of bounding inter-yield distances affects the latency overhead, but not the correctness of the primary's execution. We next describe the state, transfer function and join operation of the analysis:

- **State:** The state of our analysis is a list of yields and their average *uninstrumented* distances (in terms of time/cycles) to the current program point. If the scavenger were to yield here, these are the expected amount of time the scavenger has consumed before relinquishing the core since the previous yield points. Note that only yields with paths to the current program point that do not contain any other yield are included in the list. Input and output states of a basic block thus represent the uninstrumented distances before and after the basic block execution. MSH focuses on these states as they directly allow bounding inter-yield distances.
- **Transfer function:** This determines how the output state of a basic block is calculated based on its input state. If no new yields are added, the output state is simply the input incremented by the average latency of the basic block. This average latency can be computed with latency samples from profiling, or estimated as the product of the number of instructions and the scavenger's CPI. If any of the incremented distance is larger than the bound, MSH looks for a subset of its incoming edges to instrument yields. As described below, this will change the input (and consequently output) state of the basic block to contain new yield points and hopefully keep all the distances in the output within bound. If no such subset can be found, MSH inserts a yield at the end of the basic block and sets the output state to have only this yield point with zero distance.
- **Join operation:** This determines how the input state of a basic block is calculated based on the output states of its predecessors. For predecessors whose incoming edges are not instrumented, yields in their output states are all included in the basic block's input state, with distances being weighted averages of the corresponding distances in predecessors' output states. The weights are proportional to hotness of incoming edges, obtained via profiling. For instrumented incoming edges, the predecessor's output state will not propagate, instead the inserted yield is added to the basic block's input state with zero distance.

For the analysis, MSH ignores back edges (loops are handled later) and sorts basic blocks topologically, so that output states of predecessors are available before a basic block's turn. MSH sets the input state of the entry basic block to be a pseudo-yield named "function-start" with zero distance. MSH then iteratively computes all the states with the transfer function and join operation. Here, there are two aspects that require careful treatments – loops and function calls:

- **Loops:** For each loop, MSH computes the expected uninstrumented distance as a weighted average of the distances of all uninstrumented paths from the header basic block to the latch basic block, where weights correspond to path hotness. If the distance is zero (*i.e.,* all paths have yields), no loop instrumentation is needed. Otherwise, MSH instruments the back edge so that it yields every bound divided by distance iterations. To do this, MSH uses an induction register if available; otherwise MSH maintains a counter with unused registers or in per-thread data structures.
- **Function calls:** One aspect omitted so far is the treatment of function calls. For calls whose callee are unknown or external, MSH treats them as normal instructions. For uninstrumented external library calls that are known to be expensive, we adopt the standard practice of instrumenting right before and after the calls [13, 58]. Instead, for calls to local functions, MSH considers whether there are uninstrumented paths (*i.e.,* from entry to exits) in the callee – if yes, distances in the basic block's output state are incremented by the average uninstrumented latency of the callee; otherwise, since previous yields will be terminated in this call, MSH resets the output state to have only a pseudo-yield for the call with zero distance. The uninstrumented latency of a callee is computed with the distances for the function-start entry in the output states of its exit basic blocks. To use callee's analysis results, MSH builds a function call graph, ignores some calls to break loops, and analyzes functions in a topological order.

In summary, MSH can scale up concurrency to fully harvest stall cycles (§6.2) and manage latency impacts by enforcing inter-yield distance bounds via data-flow analysis (§6.4).

4.3 MSH Runtime

MSH intercepts function calls and assigns scavengers to active primary threads with minimal runtime overhead using tailored data structures. Next, we present the runtime design.

Function interception: MSH intercepts three types of functions: (i) functions starting a thread: *e.g.,* pthread_create, (ii) functions (likely) blocking a thread, *e.g.,* pthread_mutex_lock, and (iii) functions terminating a thread, *e.g.,* returning from the thread's start routine. Note that if there are unintercepted function calls that alter thread status, MSH's correctness is unaffected: *e.g.,* if a thread gets blocked silently (from the view of MSH), its scavengers will stay with the blocked thread, and harvestings will continue normally once the thread resumes.

Runtime operations: MSH performs different operations before/after intercepted calls to adjust scavenger assignment:

- **Scavenger initialization:** MSH initializes a new scavenger before assigning it to a primary thread, which includes loading the scavenger code, allocating its stack space and setting the return address for MSH to track when it finishes.
- **Scavenger assignment:** MSH assigns scavengers to a primary thread by configuring yield targets. The target for primary threads is a scavenger, and the target for scavengers is a primary thread by default, or another scavenger for special yields. MSH assigns more scavengers to a thread until the product of special yield ratios for scavengers is below a threshold or the scavenger number reaches a maximum.
- **Scavenger stealing:** When a primary thread needs scavengers, MSH first attempts to "steal" existing scavengers.

```
bool steal_scavengers(per_thread_ctx *t) {
    for (per_thread_ctx *it: thread_list) {
        if (CAS(it->stealable, true, false)) {
            it->stolen = migrate_scavengers(t, it);
            it->stealable = true;
            if(!need_more_scavengers(t))
                return true;
        }
    }
    return false;
}
void get_scavengers(per_thread_ctx *t) {
    if(!steal_scavengers(t)) {
        fetch_scavengers_from_pool(t);
    }
}
void enter_blockable_call(per_thread_ctx *t) {
    t->stealable = true;
}
void exit_blockable_call(per_thread_ctx *t) {
    while (!CAS(t->stealable, true, false)) {}
    if (t->stolen) {
        get_scavengers(t);
        update_yield_targets(t->yield_contexts);
        t->stolen = false;
    }
}
```

Listing 1: Pseudocode for key functions of MSH's runtime.

MSH ensures that each scavenger is assigned to at most one active thread at any time, by marking the scavengers of a thread as stealable before the thread gets blocked or terminated and only re-assigning stealable scavengers.

- **Scavenger fetching:** When there are no stealable scavengers, MSH fetches new scavengers from a pool. These scavengers should be initialized before getting assigned.

For functions starting a thread, MSH obtains scavengers via stealing or fetching and initializes them if necessary before assigning them to the thread. For functions (potentially) blocking a thread, MSH marks the thread's scavengers as stealable before the function call. After the call, MSH first attempts to reuse the scavengers previously assigned to this thread. If some scavengers were stolen, MSH obtains new scavengers with the same logic as the one for thread creation functions. Having "sticky scavengers" is good for cache locality, as scavengers mostly remain in the same core unless the primary thread gets migrated by the kernel. Lastly, for functions terminating a thread, before the thread destruction, MSH marks its scavengers as stealable.

Data structures: MSH tailors its data structures to prioritize critical events that are short but take place frequently, as overhead added to them likely leads to performance degradation. We identify two critical events: (i) primary and scavenger yielding and (ii) primary threads quickly resuming after blocking calls. (i) requires primaries and scavengers to quickly check their yield targets. (ii) occurs because a likely blocking function may not block after all (*e.g.,* synchronization calls).

MSH's data structures are shown in Figure 5. For event (i), the goal is allowing primaries and scavengers to quickly check their yield targets. A naive design is to have a per-application data structure that stores the context for each primary thread and scavenger. Such a context includes its stack and instruction pointers, and a runtime allocated stack in the case of a scavenger. Each primary thread has a per-thread data structure that stores pointers to contexts. Such a design, while intuitive, adds indirection overhead for yields: each primary thread or scavenger first reads its pointer in the per-thread data structure, in order to read its target's stack and instruction pointers (in a different cache line) from the per-application structure. Given the high frequency and small time budgets of yields, such a design is undesirable.

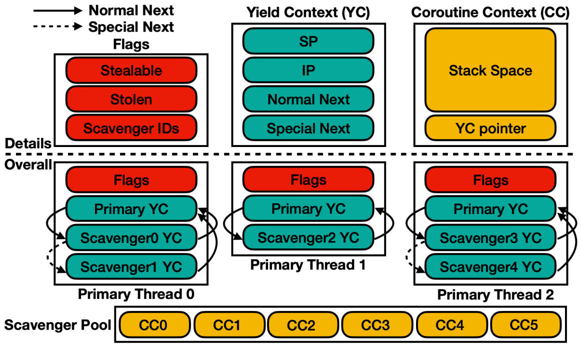


Figure 5: Data structures managed by MSH runtime. Some fields are omitted due to space constraints.

In contrast, MSH adopts a design that effectively removes the indirection overhead for yields. MSH divides a scavenger context into two parts: a "yield context", containing information needed for yielding to the scavenger, *i.e.,* its stack and instruction pointers; and a "coroutine context", containing other relevant information, *e.g.,* the scavenger stack and a pointer to the yield context. The coroutine context of each scavenger is stored in a per-application data structure, as it is in the naive design. As for the yield context, it is augmented with indexes of its targets (so effectively pointers), and the augmented yield contexts of the primary thread and its scavengers are stored contiguously on the primary's per-thread data structure. With this arrangement, each primary thread or scavenger yields by reading two yield contexts, one of itself and the other of its target. MSH minimizes the size of yield contexts, so that these two yield contexts often reside in the same cache line, resulting in little overhead. Moreover, since scavenger stacks reside in the shared data structure, MSH can easily migrate scavengers by setting up the targets in the per-thread data structures, without having to copy their stacks.

For event (ii), MSH strives to minimize the overhead for when a primary thread quickly resumes with no blocking and no scavengers stolen. A naive design is to maintain the status of each scavenger, whether it is stealable or has been stolen, in a per-application data structure. This makes scavenger stealing simple by just looking for stealable scavengers and changing their status to stolen. However, such a design complicates the operations that a primary thread needs to perform before and after a (likely) blocking call, which includes reading and setting the status of all the assigned scavengers. This process is unnecessarily expensive when there is no blocking.

In contrast, MSH optimizes for this case by leveraging two per-thread flags: a "stealable" flag indicating whether this thread is blocked, and a "stolen" flag indicating whether some scavengers were stolen. As shown in Listing 1, before a primary thread enters a blocking function, it simply sets the stealable flag to be true. If it does not get blocked, it (i) waits for the stealable flag to become true (explained later), which will be immediate in this case, and (ii) resumes its execution if the stolen flag is false. As a result, a primary thread that quickly resumes at a blocking function only performs a read, a write, and a CAS operation on a single cache line, which is significantly less work than the baseline design.

To steal scavengers, a new thread attempts to compare-and-swap the stealable flags of other threads from true to false. If succeeded, this means that (i) that thread is blocked and (ii) no other thread is stealing from this thread. The new thread then steals the blocked thread's scavengers by looking at their yield contexts – if a scavenger's yield context is valid, it copies the yield context to its own per-thread structure before invalidating the context. The new thread ends its stealing by setting both the stolen and stealable flags of the blocked thread as true. Once the blocked thread resumes, it finds out that some of its scavengers get stolen via the stolen flag, which triggers the slow path of replacing its stolen scavengers with new ones. In essence, by using per-thread flags, MSH expedites the cases where the per-thread flags are untouched due to short or no blocking. The cost of more complex scavenger stealing is acceptable given that stealings happen infrequently.

To sum up, MSH is capable of dynamically assigning scavengers to primary threads for unmodified multi-threaded applications (§6.2) and does so with minimal overhead (§6.4).

5 Implementation

We prototype MSH's offline parts on top of Bolt [67], a binary optimizer, as well as perf [24], a sample-based profiler; and MSH's runtime as a user-level library. Next, we describe how the four main components are implemented:

Offline profiling: MSH adopts the same set of profiling practices as prior sample-based profiling works [17, 30, 31, 41, 66–68]: sampled inputs are used for profiling, and in the case of input changes leading to notable performance degradations, different profiling runs happen in the background. In practice, MSH's performance is observed to be consistent across different inputs. This is because programs often have a fixed set of delinquent load instructions that trigger cache misses, an insight that has been observed and exploited in cache prefetching works [11, 41, 54]. MSH parallelizes profile processing across multiple cores to speed up the process.

Primary instrumentation: There are three phases: a profiling phase, where we profile load instructions causing cache misses via PEBS and basic block execution counts via LBR, and parse profiled data; an analysis phase, where program analysis results (*e.g.,* what registers to save) are annotated in relevant program points (*e.g.,* load instructions, loops); and an instrumentation phase, where binaries are finally altered. We reuse register liveness and reaching definition analysis from Bolt, and implement loop optimizations and pseudo-inlining.

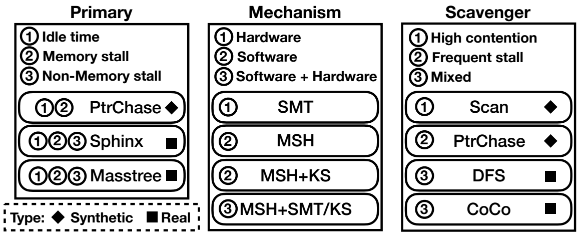


Figure 6: Primaries, scavengers and mechanisms evaluated.

Scavenger instrumentation: This takes place in the same three phases. In the profiling phase, we obtain the basic block latency via LBR. Given that LBR reports the latency between different branching instructions, which does not always correspond to a basic block's latency, we implement a script to map LBR samples to basic blocks. In the analysis phase, we construct call graphs and implement the data-flow analysis.

MSH Runtime: We use the LD_PRELOAD dynamic linker feature [73] to override pthread functions, and implement in a shared library MSH's runtime operations before/after calling the original pthread functions. For per-thread data structures, the runtime sets their base addresses in the GS segment register upon thread creations, so that they can be accessed by primaries and scavengers via GS-based addressing [53].

6 Evaluation

In this section, we present our evaluation setup (§6.1) and investigate three key questions regarding MSH: (i) how well does MSH perform compared to SMT? (§6.2), (ii) how does MSH change the landscape of cycle harvesting? (§6.3) and (iii) how do different components of MSH contribute to its performance? (§6.4). We answer (i) and (ii) by evaluating different mechanisms with both synthetic workloads and real applications, (iii) by carefully testing the specific component.

6.1 Evaluation Setup

As shown in Figure 6, we carefully select primaries, scavengers and mechanisms to allow a comprehensive understanding of MSH's behaviors and the cycle harvesting landscape.

Harvestable cycles: To set up evaluations, it is important to realize that there are three main classes of harvestable cycles. The first class is idle time, which occurs at low loads when an application does not have enough work for its cores. Software mechanisms like kernel scheduling (KS) focus on harvesting these cycles. As the load increases, idle time reduces and CPU stalls become the main harvestable cycles. CPU stalls

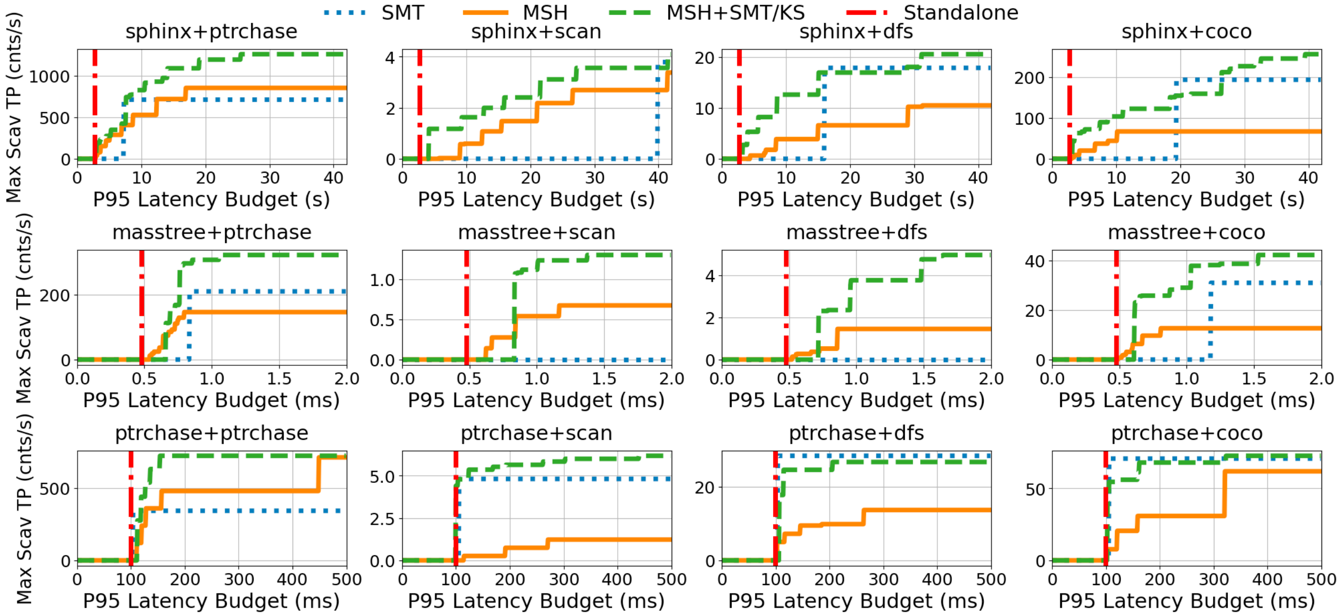


Figure 7: Maximum scavenger throughput vs. P95 Latency budget at 80% load. The red line denotes the standalone latency.

can be divided into either memory stalls, which often account for a significant portion of cycles (§2) and can be efficiently harvested by MSH, or non-memory stalls (*e.g.,* core-bound or frontend stalls), which remain to be private territory of SMT.

Primaries: For primaries, we include a synthetic pointer-chasing workload (PtrChase), which has most of its active cycles bounded by memory. It thus allows us to study how well MSH harvests memory stalls in comparison to SMT. We also have two real latency-critical applications: Masstree [59], an in-memory key-value store, and Sphinx [87], a speech recognition system. With these workloads, we evaluate harvesting mechanisms on realistic mixes of memory and non-memory stalls. Masstree and Sphinx are configured to use the same dataset as Tailbench [46] with 6 and 24 threads respectively. PtrChase has 8 threads, each iterating over its own 16MB array via random pointer chasing upon new requests.

Mechanisms: SMT harvests all three classes of harvestable cycles, but suffers from high latency overhead, lack of configurability, and incomplete harvesting (§2). MSH harvests memory-bound stalls and overcomes the drawbacks of SMT. Building on MSH's superior performance, we complement it with KS and SMT to also harvest idle time and non-memory stalls: KS adds little overhead to MSH but allows idle time harvesting; MSH+SMT/KS enables SMT with MSH if the primary latency meets the SLO, disables SMT and runs KS otherwise. This allows exploiting SMT's ability to harvest non-memory stalls, while managing its latency impacts.

SMT[3] runs scavengers on the sibling cores of the primary. MSH interleaves scavenger executions within the primary. MSH+KS schedules scavengers to run on the primary's logical cores with lower real-time priority, so that these scavengers run when the primary is idle. MSH+SMT/KS runs other scavengers on sibling cores when SMT is enabled.

Scavengers: SMT performs poorly for scavengers that contend for core resources or frequently stall, causing large latency overhead and incomplete harvesting respectively. We thus include synthetic workloads with such behaviors: Scan – creating contention by scanning a 4MB array and computing the sum; PtrChase – frequently stalling due to iterating through a 16MB array in random order via pointer chasing, to evaluate whether MSH can handle such challenging cases. We also include two graph analysis workloads: DFS and Connected Component (CoCo), from the CRONO benchmark [1] as representatives of scavengers with mixed behaviors.

Testbed and Metrics: We conduct experiments using a dual-socket server with 56-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz[4]. We measure at different loads the 95 percentile primary latency as well the scavenger throughput in terms of the number of scavengers finished per second.

6.2 MSH performance

Summary: We extensively evaluate MSH and show that it provides three main performance benefits over SMT:

- MSH can harvest up to 72% scavenger throughput of SMT, for latency SLOs under which SMT has to be disabled.
- MSH can further trade off primary latency for higher scavenger throughput if looser latency SLOs are given.

[3] We focus on Intel's SMT implementation (*i.e.,* Hyper-threading) in our evaluation. As we will discuss in §7, drawbacks of SMT stem from the lack of (software-controllable) prioritizations and the limited degrees of concurrency, which are common among most commercial SMT implementations. We thus expect our results to be representative of common SMT behaviors.

[4] Applications use memory from the local node in our evaluation. Under a NUMA setup, MSH can be configured to efficiently harvest the longer stalls caused by remote accesses, *e.g.,* by using larger inter-yield distances (§6.2).

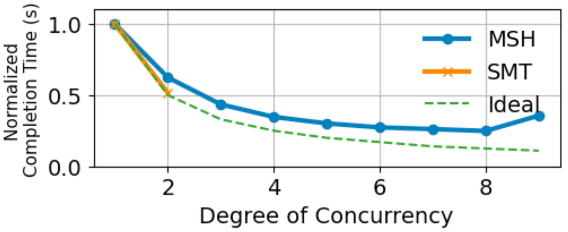


Figure 8: Time to completion for a fixed number of pointer-chasing jobs with different degrees of concurrency.

- Unlike SMT, MSH can fully harvest memory stalls when scavengers stall and achieve up to 2x higher throughput.

MSH provides these benefits with its capabilities like fine-grained configurability and concurrency scaling, which we will elaborate further on §6.4. Here we focus on presenting MSH's performance characteristics in comparison to SMT.

The whole picture: As shown in Figure 7, for each of the primary and scavenger combinations, we report the maximum achievable scavenger throughputs under different primary latency SLOs, which is defined as the latency budget at 80% loads. Note that, the comparisons among harvesting mechanisms remain unchanged for different latency metrics (e.g. average, 99 percentile) at other loads (other than 80%). As discussed below, MSH can be flexibly configured to achieve different scavenger throughputs depending on the primary latency budgets. These results thus allow us to have a holistic understanding of MSH's performance in comparison to SMT. Here one could make several key observations:

First, MSH harvests substantial stall cycles for latency SLOs under which SMT effectively achieves zero scavenger throughput (*i.e.,* disabled). This is especially valuable when contentious scavengers cause significant slowdown for SMT: *e.g.,* for Sphinx with Scan, MSH achieves up to 72% of SMT scavenger throughput with lower than SMT primary latency. Such behaviors exist for Sphinx and Masstree with all the evaluated scavengers, indicating the general usefulness of MSH as a harvesting mechanism under stringent latency SLOs.

Second, unlike SMT, which achieves the same scavenger throughput regardless of the latency SLO given, MSH can trade off primary latency for higher scavenger throughput. This capability, together with the aforementioned ability to harvest stall cycles under stringent latency SLOs, makes MSH a highly elastic harvesting mechanism that can be combined with other mechanisms, as we will describe in §6.3.

Lastly, MSH can fully harvest memory stalls even when scavengers frequently stall. Specifically, for the PtrChase scavenger, with both Sphinx and PtrChase primaries, MSH manages to achieve higher scavenger throughput than SMT without incurring much latency overhead. Given that SMT harvests both idle time and non-memory stalls, which MSH does not handle, this indicates that MSH can better harvest memory stalls with higher degrees of concurrency.

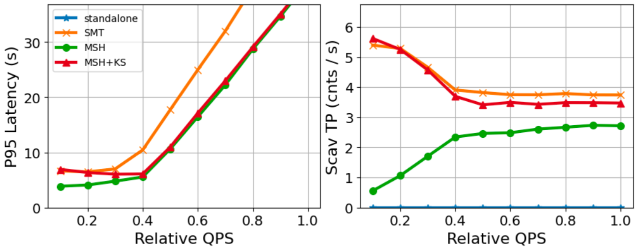


Figure 9: SMT, MSH and MSH+KS for Sphinx+Scan.

Full harvesting: To verify this, we conduct an experiment with a fixed number of jobs, where each job traverses a 128 MB array via random pointer chasing and thus frequently incurs memory stalls. We then measure the total completion time of these jobs with a single physical core. For SMT, we either run one job at a time or co-locate two concurrent jobs. For MSH, we interleave these jobs with various degrees of concurrency. The normalized completion times are shown in Figure 8. In the ideal case, the completion time is one over the concurrency degree. Although SMT-2 is close to ideal thanks to hardware efficiency, it does not have enough concurrency to further harvest memory stalls. In contrast, while having larger interleaving overhead, MSH reduces SMT's completion time by roughly a half (*i.e.,* 2x throughput) with a concurrency degree of eight. This shows that compared with SMT, MSH can harvest more memory stalls via concurrency scaling. When the degree of concurrency goes beyond eight, the completion time of MSH increases due to the aggregate yielding overhead outweighing the benefits of additional multiplexings.

6.3 Cycle Harvesting Landscape

With various desirable properties, MSH can be efficiently combined with other harvesting mechanisms to re-shape the CPU cycle harvesting landscape. To see this, we evaluate two compound mechanisms that leverage MSH for memory stalls: MSH+KS and MSH+SMT/KS, and compare that with SMT.

- **MSH+KS:** KS complements MSH with idle time harvesting. MSH+KS thus achieves much higher scavenger throughput than MSH at low loads, while adding small latency overhead (Figure 9). As the load increases, idle time reduces, and MSH+KS behaviors converge to MSH's. Note that MSH in this figure is only one configuration.

- **MSH+SMT/KS:** MSH+SMT/KS strives to utilize SMT's ability to harvest non-memory stalls, and falls back to KS if SMT incurs excessive latency overhead. As shown in Figure 7, MSH+SMT/KS delivers superior performance, with higher scavenger throughput than SMT under *almost all* latency SLOs. The reason is that: (i) for scavengers that frequently stall, SMT can be safely enabled with minimal latency overhead, the combination of SMT and MSH can harvest idle time, non-memory and memory stalls to the full extent; (ii) for contentious scavengers, the combination of KS and MSH then efficiently harvests both idle time and memory stalls for latency SLOs where SMT is disabled.

6.4 Performance Breakdown

Summary: We test MSH's configurability and performance of its components, the results of which are outlined below:

- **Configurability:** MSH offers fine-grained control over the latency-throughput trade-off via (i) yield site selections in primary instrumentation, (ii) inter-yield distances in scavenger instrumentation and (iii) concurrency degrees in runtime. Since the effects of concurrency scaling have been studied in Figure 8, we focus on the other two knobs. We measure the primary latency and scavenger throughput for Sphinx and Scan with different configurations, with results shown in Figure 10. For the primary, MSH estimates the overhead of each load instruction with its cache miss rate and bounds the aggregate overhead when selecting yield sites (§4.1). We increase this overhead bound from 5% to 15% and observe a clear latency-throughput trade-off as more yields are instrumented. For the scavenger, increasing the target inter-yield distance also leads to higher scavenger throughput at the cost of larger primary overhead. Besides the latency-throughput trade-off, such configurability allows MSH to mitigate some inherent issues of instruction interleaving, such as increased memory contention and effectively partitioned caches, by controlling the extent and locations of interleaving.

- **Primary instrumentation:** MSH reduces the yield cost by minimizing the amount of register savings and restorations per yield. To measure how this affects its harvesting performance, we conduct an experiment with Sphinx and Scan, where we measure Sphinx's latency for different inter-yield distances of Scan, with and without our optimizations. As shown in Figure 11, reduced yield costs do lead to up to 23% lower primary latency. Note that the improvement first increases with scavenger inter-yield distances before dropping, because (i) the larger yield cost (without optimizations) does not affect the primary latency until the duration of the interleaved scavenger execution (*i.e.,* inter-yield distance plus yield cost) exceeds the cache hit latency, and (ii) as the inter-yield distance further increases, yield cost plays a smaller part in the overall overhead.

- **Scavenger instrumentation:** MSH accurately enforces target inter-yield distances via its data-flow analysis (Figure 12-(a)). As for overhead, a unique source of overhead for scavengers is the loop instrumentation overhead – using an in-memory iteration counter is expensive for tight loops. MSH thus attempts to reuse induction registers or maintain a counter with unused registers before spilling to memory. This optimization reduces the overhead by 130% and 15% for CoCo and DFS respectively (Figure 12-(b)).

- **MSH runtime:** MSH harvests stall cycles via dynamic scavenger assignment. It does so with low overhead: 10 ns for thread resuming with unstolen scavengers, which does not cause noticeable impacts on our evaluated applications.

- **Profiling overhead:** Even with sample-based profiling using hardware performance counters, sampling events at high frequencies can still slow down the primary application. In MSH, we confirm that accurately capturing delinquent load instructions incurs minimal profiling overhead. Specifically, for Masstree, using the default sampling frequency and following the yield site selection logic (§4.1), MSH selects the *same* set of load instructions as if it were to sample 100x more frequently. As a result, while using a 100x higher sampling rate would slow down the application by 25%, the slowdown from MSH's profiling is negligible.

- **Analysis complexity:** MSH instruments only selective loads and performs mostly intra-procedural analysis, which finishes less than a minute for all the evaluated workloads.

7 Related Work

Reducing memory stalls: Orthogonal to harvesting efforts like MSH, there has been extensive research on reducing memory stalls. Beyond out-of-order executions, there are two lines of techniques based on *load slices*, *i.e.,* instructions that generate the address of a load instruction. One technique is prefetching [2, 4, 8, 12, 22, 39, 42, 56], where the cache line is prefetched after the end of its load slice; and the other technique is criticality-aware instruction scheduling [5, 6, 16, 77], where the processor prioritizes the executions of load slices, which requires hardware changes. For both techniques, there is a trade-off between capability and deployability. Simple techniques like stream prefetchers [39, 75] and prefetch insertion via static analysis [4, 20] have limited capability (*e.g.,* unable to handle complex access patterns); whereas advanced proposals like runahead prefetchers [26, 33] often have requirements that hinder wide adoptions (*e.g.,* excessive hardware complexity, source code modification). Moreover, a key requirement for both techniques to reduce stalls is that load slices end *sufficiently ahead* of the load instruction. As a result, for cases where load slices are close to the load instruction, neither technique can help. In contrast, MSH is easily deployable, requiring no hardware changes nor rewriting efforts, and harvests stall cycles for any access pattern.

SMT: For the three drawbacks of SMT (*i.e.,* latency overhead, lack of configurability and incomplete harvesting), the first two stem from the lack of prioritizations, whereas the last one is due to limited degrees of concurrency. Most modern processors from Intel and AMD have these two issues, which leads to unsatisfactory harvesting performance (§6.2). An exception is IBM Power processors [50, 63], as they (i) support assigning hardware threads with priorities that determine the ratio of physical core decode slots allotted to them, and (ii) have wider SMT with up to eight threads per core, at the cost of more complex and resource-consuming SMT design.

Given this context, the value of MSH is two fold. First,

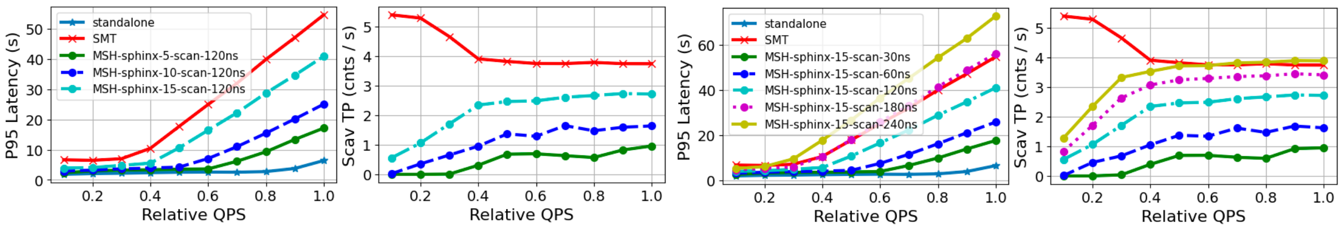


Figure 10: The effects of the aggregate yield overhead bound (left) and the scavenger inter-yield distance (right) on the primary latency and the scavenger throughput in Sphinx+Scan.

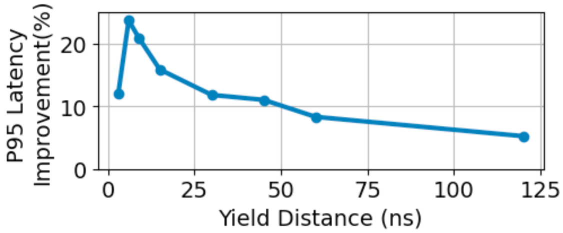


Figure 11: Latency improvement made by the yield cost optimizations in the primary instrumentation on Sphinx+Scan.

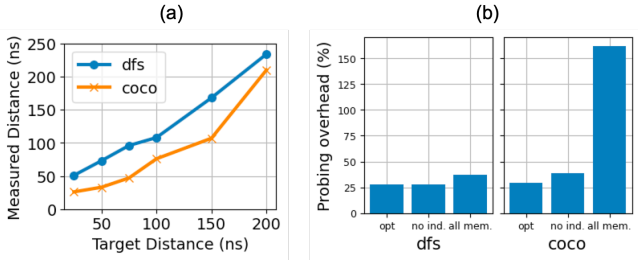


Figure 12: (a) Inter-yield distance of scavenger instrumentation; (b) overhead of loop instrumentation: *opt* uses induction registers and unused registers, *no ind.* uses only unused registers, and *all-mem* uses in-memory iteration counters.

for most modern processors, MSH allows harvesting memory stall cycles in software without the drawbacks of their SMT mechanisms. Second, for processors like IBM Power and Cray Threadstorm [48, 49] that support massive multithreading and fine-grained parallelism, MSH raises the question of whether certain functionality should be implemented in hardware or software, *e.g.,* concurrency scaling in MSH happens *on-demand*, without requiring dedicated thus likely wasted resources, such as die area and power.

Software efforts: Some work focuses on utilizing SMT with latency-critical services, by disabling it when high latency or resource interference is detected [29, 60, 70, 89]. However, they do not address SMT's high latency overhead and lack of configurability, and are thus unable to harvest stall cycles when SMT violates latency SLOs. As for software harvesting efforts, prior work shows that if done correctly, prefetching and yielding before load instructions can lead to increased throughput for memory-intensive workloads [21, 34, 44, 72]. However, they either require manual identification of yield sites and source code modification, or instrument every load instruction at the cost of high latency. Moreover, none of them can enforce low latency overhead and full harvesting from diverse scavengers, which MSH provides with scavenger instrumentation and runtime operations. In short, MSH is the first software system that enables transparent and general memory stall harvesting with competitive performance.

8 Discussion

Isolation mechanism: In MSH, the primary and its scavengers reside in the same process to benefit from fast yielding, which necessitates mechanisms other than hardware isolation to ensure memory safety under this setup. This turns out to be an extensively studied problem, with solutions falling into two main categories: (i) software-based fault isolation (SFI) [76, 81, 86], which establishes logical protection domains by inserting dynamic checks at the binary level; and (ii) language-based isolation, where a program is accepted in the form of a safe language (*e.g.,* WebAssembly [32, 36, 85], Rust [15, 51, 65, 92]) and validated by the type checker and compiler. Operating at the binary level, MSH easily coexists with either isolation mechanism: SFI can be a better fit as it is applicable to code written in different languages, including legacy code, which is a merit that MSH shares. Moreover, a recent work [91] shows lower runtime overhead with a lightweight SFI implementation than existing language-based solutions. Integrating MSH with some isolation mechanism and evaluating the resulting system is left for future work.

Further evaluation: In §6, we demonstrated and dissected the desirability of MSH as a harvesting mechanism. Next, we discuss directions for more thorough evaluation of MSH.

- **Additional workloads:** We focus on evaluating a set of representative workloads with distinct characteristics, *e.g.,* scavengers that either create large contentions, or frequently stall, or exhibit mixed behaviors. This appraoch allows us to interpret the performance differences caused by (i) the distinct characteristics of the primary-scavenger pairs and (ii) the differences in harvesting mechanisms. One could extend with more real workloads
- **Cache prefetching:** As discussed in §7, MSH can harvest memory stalls that are not hidden by cache prefetching, and prefetching techniques that are easy to deploy usually have limited capability. It will thus be interesting to evaluate the

effectiveness of MSH with software prefetching techniques used in production [40]. That being said, most delinquent loads in our evaluated workloads exhibit pointer-chasing behaviors, which are inherently challenging to prefetch.

- **Datacenter efficiency:** The effect of MSH on the overall CPU efficiency of a datacenter is hard to estimate, as it depends on various factors such as workload characteristics, colocation arrangements, and SLO policies. This necessitates large-scale evaluation and profiling [45].

Efficacy of profiling: In terms of profiling overhead, we have shown that MSH can capture delinquent load instructions with a low sampling rate (§6.4). The other natural question is whether profiling is consistently effective for the purpose of harvesting stall cycles in MSH. Similar to prior works that leverage profiling for cache prefetching [40, 41, 95], we show positive results with our evaluated workloads (§6.2). One conjecture is that, while whether a particular load invocation will trigger a cache miss is highly random, the two pieces of information MSH needs from profiling – namely, (i) the set of load instructions that account for a significant portion of memory stalls and (ii) their likelihoods of cache misses, are often stable across runs and inputs. Evaluating a wider range of applications can help further validate this conjecture.

Hardware support for MSH: We identify two aspects that MSH can benefit from hardware support. First, an overhead that MSH inevitably incurs is when an instrumented load causes cache hits. MSH mitigates this with the selection logic in primary instrumentation, which enforces a lower bound on cache miss rate and an upper bound on aggregate overhead (§4.1). To do better, what is needed is *dynamic visibility* of cache misses, *e.g.,* indicating if a cache line is in L2 cache. This allows yields to be conditional on whether cache misses actually happen. We expect conditional checking overhead to be on the scale of L2 cache latency, much faster than scavenger executions configured to harvest memory stalls.

Another aspect that hardware can offer support is reducing yield overhead. MSH minimizes the amount of register savings and restorations for each yield, which leads to lower latency overhead (§6.4). One useful hardware feature here is to save/restore multiple registers to/from memory with a single instruction for lower instruction fetch costs, which is already provided in ARM with LDM/STM instructions [9]. Prior works also propose hardware support for fast saving and restoration of process state during context switches [38, 80].

9 Conclusion

We presented MSH, a software system that transparently and efficiently harvests memory stall cycles. With a co-design of profiling, program analysis, binary instrumentation and runtime scheduling, MSH fully harvests stall cycles, while incurring minimal latency overhead and offering fine-grained control of the latency-throughput tradeoff. MSH is thus a preferable solution for harvesting memory stalls and brings valuable changes to the CPU cycle harvesting landscape.

References

[1] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55. IEEE, 2015.

[2] Sam Ainsworth and Timothy M Jones. An event-triggered programmable prefetcher for irregular workloads. *ACM Sigplan Notices*, 53(2):578–592, 2018.

[3] Soramichi Akiyama and Takahiro Hirofuchi. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, pages 1–8, 2017.

[4] Hassan Al-Sukhni, Ian Bratt, and Daniel A Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–100. IEEE, 2003.

[5] Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer, and Rakesh Kumar. Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 424–434. IEEE, 2020.

[6] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 716–721. IEEE, 2019.

[7] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing {SLOs} for {Resource-Harvesting}{VMs} in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[8] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. Data prefetching by dependence graph precomputation. *ACM SIGARCH Computer Architecture News*, 29(2):52–61, 2001.

[9] ARM. Arm developer suite assembler guide. https://developer.arm.com/documentation/dui0068/b/Writing-ARM-and-Thumb-Assembly-Language/Load-and-store-multiple-register-instructions/ARM-LDM-and-STM-instructions, 2023.

[10] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[11] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[12] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411. IEEE, 2019.

[13] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

[14] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with blast. In *International Conference on Fundamental Approaches to Software Engineering*, pages 2–18. Springer, 2005.

[15] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19, 2020.

[16] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 272–284, 2015.

[17] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.

[18] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 347–364, 2023.

[19] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[20] William Y Chen, Scott A Mahlke, Pohua P Chang, and Wen-mei W Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 69–73, 1991.

[21] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the killer microsecond. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 627–640. IEEE, 2018.

[22] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: Long-range prefetching of delinquent loads. *ACM SIGARCH Computer Architecture News*, 29(2):14–25, 2001.

[23] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[24] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[25] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[26] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75, 1997.

[27] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[28] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN*

International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pages 68–84, 2021.

[29] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[30] Google. Propeller: Profile guided optimizing large scale llvmbased relinker. https://github.com/google/llvm-propeller, 2020.

[31] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 692–708, 2023.

[32] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[33] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[34] Yongjun He, Jiacheng Lu, and Tianzheng Wang. Corobase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment*, 14(3):431–444, 2020.

[35] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press, 2012.

[36] Pat Hickey. How fastly and the developer community are investing in the webassembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem/, 2020.

[37] Joel Hruska. Maximized performance: Comparing the effects of hyper-threading, software updates. https://www.extremetech.com/computing/133121-maximized-performance-comparing-the-effects-of-hyper-threading-software-updates, 2012.

[38] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 17–25, 2021.

[39] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408. IEEE, 2006.

[40] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 577–590, 2024.

[41] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[42] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[43] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A blackbox batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the ACM symposium on cloud computing*, pages 272–285, 2019.

[44] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment*, 11(11):1702–1714, 2018.

[45] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[46] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[47] Andi Kleen. An introduction to last branch records. https://lwn.net/Articles/680985/, 2016.

[48] Petr Konecny. Introducing the cray xmt. In *Proc. Cray User Group meeting (CUG 2007). Seattle, WA: CUG Proceedings*, 2007.

[49] Andrew Kopser and Dennis Vollrath. Overview of the next generation cray xmt. In *Cray User Group Proceedings*, pages 1–10, 2011.

[50] Hung Q Le, JA Van Norstrand, Brian W Thompto, José E Moreira, Dung Q Nguyen, David Hrusecky, MJ Genden, and Michael Kroener. Ibm power9 processor core. *IBM Journal of Research and Development*, 62(4/5):2–1, 2018.

[51] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[52] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106, 2010.

[53] Linux. Using fs and gs segments in user space applications. https://www.kernel.org/doc/html/next/x86/x86_64/fsgs.html, 2023.

[54] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. Crisp: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 300–313, 2022.

[55] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[56] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, 1996.

[57] Zhihong Luo, Silvery Fu, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Out of hand for hardware? within reach for software! In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 30–37, 2023.

[58] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 305–319, 2024.

[59] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[60] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing qos and throughput for colocated server workloads on smt cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–27. IEEE, 2019.

[61] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[62] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[63] Alessandro Morari, Carlos Boneti, Francisco J Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose, and Mateo Valero. Smt malleability in ibm power5 and power6 processors. *IEEE Transactions on Computers*, 62(4):813–826, 2012.

[64] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[65] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. {RedLeaf}: isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39, 2020.

[66] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244. IEEE, 2017.

[67] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[68] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 119–130, 2021.

[69] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[70] Aidi Pi, Xiaobo Zhou, and Chengzhong Xu. Holmes: Smt interference diagnosis and cpu scheduling for job co-location. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 110–121, 2022.

[71] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 35–44. IEEE, 2002.

[72] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment*, 11(CONF):230–242, 2017.

[73] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

[74] Steven E Raasch and Steven K Reinhardt. Applications of thread prioritization in smt processors. In *Proc. of the Workshop on Multithreaded Execution And Compilation*. Citeseer, 1999.

[75] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, 2003.

[76] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[77] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. Long term parking (ltp) criticality-aware resource allocation in ooo processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 334–346, 2015.

[78] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[79] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient coroutines for the java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 20–28, 2010.

[80] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. μmanycore: A cloud-native cpu for tail at scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[81] Gang Tan et al. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*, 1(3):137–198, 2017.

[82] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 157–173. Springer, 2010.

[83] Dean M Tullsen and Jeffery A Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 318–327. IEEE, 2001.

[84] Antonio Valles, Matt Gillespie, and Garrett Drysdale. Performance insights to intel® hyper-threading technology. *Source:< https://software. intel. com/enus/articles/performance-insights-to-intel-hyper-threadingtechnology*, 2009.

[85] Kenton Varda. Webassembly on cloudflare workers. `https://blog.cloudflare.com/webassembly-on-cloudflare-workers/`, 2018.

[86] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[87] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition, 2004.

[88] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 1–16, 2021.

[89] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling:{Fine-Grain} principled borrowing from {Latency-Critical} workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 309–322, 2016.

[90] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[91] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 649–665, 2024.

[92] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[93] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.

[94] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. {History-Based} harvesting of spare cycles and storage in {Large-Scale} datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 755–770, 2016.

[95] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. Rpg2: Robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 999–1013, 2024.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications

Lei Chen[†*] Shi Liu[ψ*] Chenxi Wang[†] Haoran Ma[ψ] Yifan Qiao[ψ] Zhe Wang[†]
Chenggang Wu[†] Youyou Lu[‡] Xiaobing Feng[†] Huimin Cui[†] Shan Lu[θ] Harry Xu[ψ]
University of Chinese Academy of Sciences[†] *UCLA*[ψ] *Tsinghua University*[‡] *Microsoft Research*[θ]

Abstract

With rapid advances in network hardware, far memory has gained a great deal of traction due to its ability to break the memory capacity wall. Existing far memory systems fall into one of two data paths: one that uses the kernel's paging system to transparently access far memory at the page granularity, and a second that bypasses the kernel, fetching data at the object granularity. While it is generally believed that object fetching outperforms paging due to its fine-grained access, it requires significantly more compute resources to run object-level LRU and eviction.

We built Atlas, a hybrid data plane enabled by a runtime-kernel co-design that simultaneously enables accesses via these two data paths to provide high efficiency for real-world applications. Atlas uses *always-on* profiling to continuously measure page locality. For workloads already with good locality, paging is used to fetch data, whereas for those without, object fetching is employed. Object fetching moves objects that are accessed close in time to contiguous local space, dynamically improving locality and making the execution increasingly amenable to paging, which is much more resource-efficient. Our evaluation shows that Atlas improves the throughput (*e.g.*, by 1.5× and 3.2×) and reduces the tail latency (*e.g.*, by one and two orders of magnitude) when using remote memory, compared with AIFM and Fastswap, the state-of-the-art techniques respectively in the two categories.

1 Introduction

Today's datacenters commonly suffer from low memory utilization [21]; yet, datacenter applications are increasingly memory-constrained [19, 36, 42, 62] due to their need to hold large datasets in memory for quick data analytics [11, 76] or machine learning [8, 53]. Thanks to the high bandwidth and low latency provided by modern network fabrics such as InfiniBand, far memory techniques [9, 25, 57, 67–69] enable an abstraction of unlimited memory for applications by allowing them to use available memory on remote servers, thereby simultaneously improving application performance and datacenters' overall memory utilization.

Although techniques such as RDMA enable fast network accesses, each remote access is still at least an order of magnitude slower than a local access. As such, it is paramount to optimize the remote access data plane so that applications can benefit from increased memory capacity without suffering a significant performance hit. A major line of work for accessing remote memory is using the kernel's paging system, exemplified by techniques such as InfiniSwap [25], Fastswap [9], Canvas [68] and Hermit [55]. These techniques allow applications to transparently access far memory at the *page* granularity, using the kernel's swap system to swap pages in and out between local and remote memory.

While paging works well for applications that perform bulk data movement and exhibit clear (sequential or strided) access patterns, its coarse granularity incurs substantial *I/O amplification* (*i.e.*, pages loaded only contain a small amount of useful data) for applications that exhibit irregular (or random) access patterns, such as Memcached [5] and graph applications [34]. To reduce I/O amplification, a recent line of work exemplified by AIFM [57] and Kona [13] advocates to access data at a much finer (object) granularity using a user-space runtime system. Swapping objects, rather than pages, can significantly reduce the amount of useless data swapped, leading to higher efficiency. Furthermore, since objects are the data abstraction for developers to write programs, they carry semantics (*i.e.*, user intention) that can be exposed to and used by the runtime to perform additional optimizations, such as data-structure-based prefetching.

Fetching objects at runtime, however, comes at a cost. A drawback that was often overlooked by existing works is that object fetching requires *non-trivial compute resources* to profile object usage, identify patterns, and perform object-level LRU and eviction. For instance, running an object-level LRU algorithm is **one order of magnitude** more expensive than page-based LRU due to a huge number of objects to be processed and the lack of hardware support for tracking object accesses. This overhead is significantly more pronounced in real-world scenarios where CPUs are all busy with executing application threads—given a tight time budget, memory

[*] Contributed equally.
Corresponding authors: Chenxi Wang and Harry Xu.

management threads cannot scan enough objects to make accurate LRU and eventually have to evict arbitrary objects.

As a result, the right access mechanism is essentially the result of a tradeoff between program locality (*i.e.*, how bad I/O amplification can be) and the amount of compute resources available (*i.e.*, how many cores can be dedicated to object-level memory management tasks). For programs with poor locality, the overhead of object-level memory management can be offset from the large gains of reducing I/O amplification. On the other hand, for programs with good locality and insignificant I/O amplification, the overhead of object fetching stands out, especially in an environment where applications have taken all compute resources (see §3).

There is a recent line of compiler-based techniques (as exemplified by Mira [26]) that profile a program *offline* to understand such a tradeoff, so that compiler can statically choose the mechanism for each data access when compiling the program. However, offline profiling hinges upon program input. For interactive applications such as Memcached, their input data comes from users and keeps changing, rendering a dry-run-based technique ineffective.

Major Insight. The main question we ask in this paper is: can we enable *always-on* profiling for an application to identify its access patterns and dynamically switch between paging and object fetching to adapt to the observed patterns? This approach, if implemented efficiently, has two advantages over the state-of-the-art techniques. First, its continuous profiling identifies patterns *on-the-fly* for different computation stages or parallel threads accessing different data structures, even if the program input keeps changing. As a result, it can quickly change the access path to use a more efficient fetching mechanism. Second, for programs with irregular patterns, object fetching moves objects that are accessed close in time into contiguous memory space, dynamically improving locality as the program executes. This makes it possible for the execution to *be increasingly amenable to paging*, which has higher resource efficiency (see §3).

Although promising, realizing this insight requires overcoming three major challenges, as elaborated below:

The *first challenge* is how to continuously and accurately profile an application with low overhead. Kernel-based page-level profiling, though efficient, does not provide sufficient information with respect to fine-grained data locality. For example, if one single hot object on a page keeps getting accessed but none of other objects do, the kernel-based profiling would identify the page as a hot page although the page clearly possesses poor locality and its accesses should go through object fetching, not paging.

To enable fine-grained profiling, Atlas divides a page into a set of *cards*, each of which is a unit for our locality measurement. We leverage the runtime (and in particular, a *read barrier*) to compute a *card access table (CAT)* (§4.3) for each page, which is a bitmap where each bit corresponds to a card (*i.e.*, consecutive 16 bytes) on the page and a set bit represents that the card has been accessed since the page was allocated or last swapped in. A page with a high *card access rate* (CAR, measured as the percentage of the set bits in its CAT) is deemed to possess good locality and should be accessed with paging, while a page with a low CAR has poor locality and should be accessed with object fetching.

The *second challenge* is how to dynamically switch access mechanisms. Atlas uses a read barrier at each smart pointer dereference. The barrier quickly checks a per-page *path selector flag* (PSF) for the remote page to be accessed. Each PSF is a 1-bit flag, set to either `runtime` or `paging`. `runtime` indicates that the runtime path should be used to fetch individual objects (like AIFM), while `paging` means that the paging path is taken to fetch an entire page. The PSF of a page is updated only when the page is evicted based upon the page's CAR—it is set to `runtime` if the page's CAR is low, indicating the page exhibits poor locality, and `paging` otherwise, indicating good locality.

Although Atlas supports both object fetching and paging at *ingress*, it evicts data only at the page granularity at *egress*, to reduce the high overhead associated with object-level profiling and LRU. While evicting pages may introduce I/O amplification for workloads with poor locality, this impact is insignificant under Atlas, because accesses in these workloads would likely go through the object fetching path, which improves locality by moving objects accessed close in time into contiguous local space. The enhanced locality effectively mitigates the negative impact of page-level eviction.

To reduce fragmentation resulting from dead objects, Atlas runs *concurrent evacuation* tasks that periodically move live objects into contiguous memory space. During each evacuation, Atlas groups recently-accessed objects into contiguous pages to further improve data locality.

The *third challenge* is how to synchronize the two access paths. Since the kernel and the runtime are not coordinated (*e.g.*, the kernel does not inform the runtime of the start or the completion of a page-fault handling), special care must be taken to prevent the two access paths from creating inconsistent data copies. In particular, correctness issues may arise from a set of ingress and egress events (*i.e.*, object-in, page-in, and page-out) that occur simultaneously. Atlas solves the problem with a synchronization protocol (see §4.2), implemented with a combination of runtime and kernel support.

Results. We have evaluated Atlas with a set of eight applications that cover a full range of memory access patterns: sequential, random, and mixed. Our results show that Atlas enables these applications running on remote memory to achieve an overall of 1.5× and 3.2× throughput improvement, compared with AIFM [57] and Fastswap [9], respectively. Atlas reduces the tail latency by one and two orders of magnitude when compared with AIFM and Fastswap. Atlas is available at https://github.com/wangchenxi7/Atlas.

2 Background on Object Fetching

Object fetching is motivated by two observations on the inefficiencies of paging. First, fetching data at the page granularity often leads to I/O amplification [13]. Second, managing data in the kernel space is agnostic to program semantics, resulting in missed optimization opportunities [57, 65, 67]. As such, work has been proposed to manage data with a language runtime at a finer-grained object (or cache-line) granularity [13, 43, 57, 66, 67, 69]. Unlike paging, the runtime can only manage objects in user space, which results in two consequences: (1) the runtime must change the virtual address of an object when moving it and hence must change all its pointers; and (2) the runtime must maintain all metadata itself (*e.g.*, LRU), which used to be maintained by the kernel. Here we focus our discussion on AIFM [57]. AIFM proposes two abstractions for developers to manage remote memory: *remoteable pointer* and *dereference scope*.

Remoteable pointer. AIFM extends the *smart pointer* abstraction of C++ to implement remoteable pointers (`RemPtr`) for remote data management. There are two types of `RemPtr`: 64-bit unique remoteable pointers (similar to `std::unique_ptr`) and 128-bit shared remoteable pointers (similar to `std::shared_ptr`). Developers need to explicitly declare data as remote type and manage them via the `RemPtr`. For example, each unique `RemPtr` has 64 bits—the lower 47 bits are used as the virtual address of the data, and the upper 17 bits are used to record metadata, such as dirty (D), present (P), hot (H), evacuated (E), *etc.* When accessing data via a `RemPtr`, AIFM checks the metadata of the `RemPtr` to detect its status, *e.g.*, checking the P bit to see if the object is in local memory. Next, AIFM masks the `RemPtr` to obtain the actual virtual address.

Dereference scope. Each smart pointer dereference and subsequent raw pointer accesses must be enclosed by a dereference scope, which works as an *evacuation fence* to guarantee correctness. AIFM performs periodical *concurrent object evacuation* that swaps out cold objects to remote memory and compacts local memory to improve data locality. It is challenging to move objects when they are being used by other threads since moving objects requires updating all their pointers. Smart pointers solve this problem because these pointers can be recorded in object headers and updated after moves are conducted. However, an application may read raw pointers from smart pointers and store them in registers or on the stack, which cannot be updated by the runtime.

To guarantee correctness for pointer updating, AIFM requires developers to explicitly declare dereference scopes for each object, which define where raw pointers of the object may exist. Evacuation of the object never happens concurrently with the execution of any of its dereference scopes that started before the evacuation decision. A dereference scope serves as a synchronization mechanism between an event that moves the object and another that uses it.

3 Motivation

We now motivate the necessity of a hybrid data plane. We first demonstrate the diverse memory access patterns of real-world cloud applications and explain the underlying reasons. Next, we compare fetching performance between using a runtime and the kernel's paging system. For the runtime approach, we re-implemented applications with AIFM [57]. For paging, we used Fastswap [9]. Finally, we discuss the opportunities provided by *dynamic* path switching.

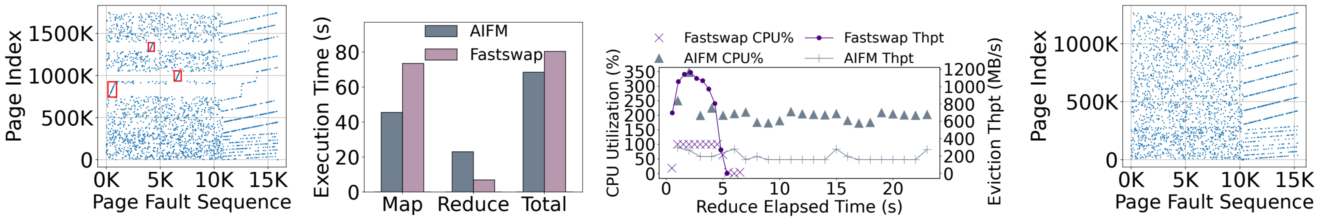
Diverse memory accesses. Real-world applications exhibit complicated memory access patterns, which are a combination of multiple primitive patterns such as sequential, strided, skewed, and random. Access patterns depend on at least two factors: (1) the computation model and (2) the data model. Next we elaborate on these factors:

On one hand, many applications are phase-changing and each phase follows a distinct computation model. On the other hand, the same phase may exhibit varied access patterns when processing different data structures.

An example is data-processing applications [76, 78] that implement MapReduce. We experiment with Metis [44], a MapReduce framework optimized for multicore architectures, with a Page View Count (PVC) program [29, 56] and report its page fault sequence in Figure 1(a). Since PVC is executed with 8 cores, we launch 8 threads for each (Map or Reduce) phase to exploit data parallelism. During the Map phase, each thread loads chunks of input data from the disk and initializes loaded website URLs and users as memory data. Next, PVC shuffles URLs into different buckets of a hash table based on their hash values. The Reduce phase scans each entry to count each URL's users.

The left/right part of Figure 1(a) illustrates the page fault sequence of the Map/Reduce phase. The Map phase (left) inserts URLs into the hash table, and accesses there are mostly random. However, given that the dataset used to run this program is *skewed*, there are several ranges of sequential accesses in the Map phase, as highlighted in the boxes (*i.e.*, certain hash buckets are much larger than others and hence traversing these buckets exhibits sequential patterns). During the Reduce phase (right), each task that aggregates users of URLs scans entries in a bucket sequentially, resulting in a clear sequential access pattern, as shown in the second (right) half of Figure 1(a).

Granularity-performance tradeoff. Object fetching minimizes I/O amplification by fetching fine-grained objects [14, 57]. However, compared to paging, object fetching does not always show clear benefits—for workloads with good locality, data on the same pages are accessed close in time and the kernel can already effectively and accurately prefetch data. When benefits are insignificant, the overhead for object-level memory management stands out. To compare fetching efficiency between the runtime and the kernel, we run the Metis



(a) Page Fault Trace 1 (b) Throughput (c) Memory Eviction (d) Page Fault Trace 2

Figure 1: Statistics of Metis PageViewCount (MPVC): (a) access patterns, (b) performance comparisons between AIFM and Fastswap, (c) comparisons of eviction throughput (dotted lines) and CPU usage (crosses and triangles) between AIFM and Fastswap, and (d) access patterns when input is changed to Wikipedia Italian [6]. For these experiments, 25% of the working set resides in the compute server's local memory. Sequential accesses (due to skewness) in the Map phase are highlighted in red boxes in (a), while in (d) such patterns do not exist.

PVC benchmark on AIFM and Fastswap, respectively. Figure 1(b) reports their performance comparisons.

Since a MapReduce program has clear phases, we broke down the execution time into Map and Reduce. AIFM outperforms Fastswap by 1.6× in the Map phase due to object fetching—most remote accesses in Map are random as words are inserted into different buckets of the hash map. On the contrary, AIFM underperforms Fastswap by 3.3× in the Reduce phase, which exhibits clear sequential patterns.

Object eviction cost. The main reason why object fetching underperforms paging for programs with good locality is the high cost associated with profiling objects and maintaining object-based LRU for eviction. For example, eviction must be done quickly as it blocks further memory allocations [55]. As a result, AIFM constantly maintains dozens of profiling/eviction threads to track the hotness of (billions of) objects and evict cold objects. However, if these threads cannot obtain enough CPU resources from the application, they end up scanning only a small percentage of objects before time runs out and then evict objects with limited hotness information, resulting in data thrashing (*i.e.*, hot objects get swapped out and quickly swapped back in).

Figure 1(c) compares the eviction throughput and CPU utilization for eviction of AIFM and Fastswap during the Reduce phase. AIFM continuously performs object-level hotness tracking and eviction with around 200% (up to 350%) CPU usage in the entire Reduce phase. On the contrary, Fastswap finishes most of the page eviction task within the first five seconds and consumes no more than 100% CPU resources during the eviction. Overall, Fastswap consumes *an order of magnitude* less compute (cycles) than AIFM for eviction over the Reduce phase. Even with significantly fewer CPU resources, Fastswap's eviction throughput is still ∼**5**× higher than that of AIFM, due to the low memory management cost associated with paging.

Necessity of online profiling and path switching. Offline profiling techniques [26, 37, 41, 54, 70] were proposed to analyze program semantics and data accesses. However, these techniques are ineffective in identifying the optimal solution for a real-world application for two major reasons.

On the one hand, even if the application's computation phases may be analyzed by an offline profiling technique, its access patterns can change dramatically in response to *inputs*. As Figure 1(d) demonstrates, when fed with a different dataset (which does not exhibit skewness), the program's access patterns change significantly—*e.g.*, due to the lack of skewness, the Map phase no longer exhibits sequential patterns. In fact, for any interactive applications including Memcached [5], DataFrame [46], or streaming data systems [17, 34, 64], their behaviors and access patterns vary significantly with different user requests and workloads.

On the other hand, as discussed earlier, object fetching consumes extensive CPU resources. This may be acceptable when CPU resources are not fully saturated but becomes problematic as soon as all CPU cores are occupied (*e.g.*, another tenant starts using the server). Clearly, offline profiling is not able to predict such environmental changes.

These issues necessitate a dynamic technique that can continuously profile program executions and perform runtime data path switching as new behaviors and/or environmental changes are detected. Our main objective is to use object fetching to minimize I/O amplification and enhance locality, paving the way for subsequent accesses to operate on data with established locality and thus benefit from paging that is considerably more resource efficient.

4 Atlas Design and Implementation

This section presents Atlas's design. Like AIFM, Atlas requires programs to use smart pointers (*i.e.*, to implement barriers) and declare dereference scopes for objects (inspired by C++ weak pointers [4] and Folly RCU guards [1]). Objects are managed by Atlas's hybrid data plane. Atlas can also take the same user-defined programming/offloading hints and object-level prefetching logic as required by AIFM. Atlas uses such hints in the object fetching path.

4.1 Overview

Inspired by the design of the Java heap [51], Atlas divides a page into *cards* to enable fine-grained profiling for accesses. For each page, Atlas builds a card access table (CAT), which is a bitmap where each set bit represents a card that has been accessed since the page was allocated or last swapped in.

CATs for contiguous pages are allocated contiguously in a separate memory space. This design enables not only fine-grained access profiling, but also simple mapping from a virtual address to its CAT entry—this can be done with efficient bit-wise operations on the address. Each card represents 16 consecutive bytes, which provides a fine enough granularity as most objects are at least 16 bytes in our workloads.

Atlas maintains a 1-bit *path selector flag* (PSF) for each page, which works as an indicator of the data path for data access on the page. A `runtime` value indicates that data should be retrieved by the runtime at the object granularity (*i.e.*, runtime path). A `paging` value indicates that data should be paged in by the kernel (*i.e.*, paging path). Atlas updates the PSF of each page to `runtime or paging` at the moment the page is swapped out if its CAR (*i.e.*, the percentage of the set bits among all bits in a CAT) goes below or above a threshold (*i.e.*, 80% used in our evaluation, see §5.4).

Ingress. Atlas uses a *read barrier* that executes at each *smart pointer dereference*. The barrier first checks whether the accessed data is remote. In AIFM, this is done by using a bit in each pointer to encode the location of the referenced object—these pointers are updated once the objects they point to are swapped in or out. Atlas, however, cannot adopt this approach due to the use of the hybrid data plane—when data is *paged* out, Atlas cannot update any pointers. To solve the problem without incurring the cost of checking with the kernel at every read, Atlas leverages *hardware transaction memory* and, in particular, Intel's TSX [31], to run a quick check—Atlas accesses the address in a hardware transaction, which aborts if the address is not on a mapped page.

Upon an abort, the barrier reads the PSF for the page to be accessed and determines which path (runtime *vs.* paging) the access should take. If the runtime path is taken, the object is moved (*i.e.*, address changed) to a local page on the compute server and its pointers are updated; otherwise, the page containing the object is swapped in as a whole and the address of the object remains the same (without requiring pointer updating).

Egress. Given that the majority of the object-fetching overhead comes from the need to find and evict cold objects, Atlas utilizes a single path, *i.e.*, paging, to swap out data. This approach achieves a sweet spot in balancing overhead and benefits—on one hand, it significantly reduces the compute resource usage for object fetching because of the elimination of maintaining an object-level LRU; on the other hand, given that object fetching gradually improves locality (by moving together objects accessed closely in time), the amount of useless data in each swap-out (and thus the I/O amplification) is reduced progressively during execution.

Another reason to not evict objects individually is that it can potentially *hurt locality*—after objects are fetched in, those that were scattered in remote memory but accessed together were moved into contiguous local space; however, these objects may not be evicted at the same time; evicting them individually would make them go to unrelated locations in remote memory, disrupting established locality.

Synchronization. Allowing the two paths to co-exist in harmony requires overcoming the following three synchronization challenges: (1) *ingress synchronization* between object-in and page-in, (2) *egress synchronization* between object-in and page-out, and (3) *move synchronization* between object-in and evacuation. AIFM already solves the third problem with the declaration of dereference scopes, while the other two are unique challenges that we target in Atlas.

4.2 Synchronization of the Two Paths

Atlas builds its object fetching path upon the same two abstractions used by AIFM: the smart pointer (which is an extension of C++ smart pointer) and the dereference scope. This section elaborates on the synchronization mechanism between the object fetching path and the paging path.

```
class AtlasUniquePtr<T>{
  struct AtlasMetadata{
    unsigned long is_moving : 1;
    unsigned long access : 1;
    unsigned long reserve : 2;
    unsigned long offload : 1;
    unsigned long size : 12;
    unsigned long addr : 47;
  } metadata; // 64 bits
  AtlasUniquePtr(T* obj);
  T* get_raw();
}
```

Figure 2: Atlas unique pointer metadata.

Pointer Metadata. Before discussing our barrier logic, we first present the format of Atlas pointers, which are built on C++ smart pointers. Atlas uses two types of smart pointers: unique pointers (similar to `std::unique_ptr`) and shared pointers (similar to `std::shared_ptr`). Figure 2 shows the layout of an Atlas unique pointer. These fields are added for the purpose of synchronization and pointer updating.

Each such pointer has 64-bit metadata, in which 47 bits (`addr`) store the object's raw pointer, 12 bits (`size`) record its size, 1 bit (`access`) represents whether the object has been accessed since the last evacuation (which will be used by the evacuator to group recently accessed objects, see §4.3), 1 bit (`offload`) indicates whether a function is being invoked on the object on the remote side, and 1 bit (`is_moving`) indicates whether the object is being moved (*e.g.*, due to evacuation); this bit will be used for synchronization between two threads trying to move the same object. The remaining 2 bits (`reserve`) are reserved for future use. Note that 12 bits can represent a size up to 4KB. Objects larger than that are placed in the huge-object space of the heap for which paging is the only option. `get_raw` retrieves the raw pointer from a smart pointer.

A shared pointer allows aliasing. Atlas treats the first shared pointer of an object as the main pointer. A shared pointer's layout is similar to a unique pointer, except that it has an additional 8 bytes to chain all pointers—when the main pointer is being released, Atlas follows the chain to select a new main pointer. If an object is referenced by shared pointers, Atlas needs to update all of them (by following the chain).

Developers need to explicitly declare data types with smart pointers. Developers can access data with raw (regular C++) pointers by first retrieving such raw pointers from smart pointers. However, this can only be done within an explicitly declared dereference scope. Figure 3 illustrates an example of retrieving and manipulating data from Atlas smart pointers, confined by a dereference scope. As discussed in §2, dereference scopes synchronize with object migration tasks—once raw pointers are retrieved and actively used, their objects are not allowed to move, and vice versa. Atlas executes a `pre_scope_barrier` and a `post_scope_barrier` at the beginning and the end of the dereference scope, respectively.

```
deref_scope (smart_ptr) {
  pre_scope_barier(smart_ptr); // Algorithm 1
  Data * object = smart_ptr.get_raw();
  /* Operations using the object */
  ...
  post_scope_barrier(smart_ptr); // Algorithm 2
}
```

Figure 3: Dereferencing an Atlas unique pointer in a deref scope.

Synchronization invariants. We present a set of high-level invariants that Atlas maintains to solve the three synchronization problems: (1) preventing an object from being fetched from the two paths simultaneously (object-in *vs.* page-in), (2) preventing pages containing objects that were just runtime-fetched from being immediately swapped out (object-in *vs.* page-out), and (3) preventing an object from being simultaneously runtime-fetched and moved by the evacuator (object-in *vs.* evacuation).

Invariant #1: Object-in vs. page-in. At any moment, all data on the same page must go through the same access path as guided by the page's PSF. In other words, Atlas prohibits scenarios where certain requests are served by paging while others are served by the runtime for the same page. Given that Atlas changes PSF only at page-out (as opposed to setting it while the page is in local memory), such scenarios can never occur and this invariant is guaranteed by design.

Note that there is no issue if two threads fetch the same page from the paging path—the kernel's swap system guarantees only one page can be mapped. Fetching the same object from two threads with the runtime path is not a concern either: it is a solved problem in the literature of moving garbage collectors [43] where pointer updating is used as a synchronization point and only one object is retained.

Invariant #2: Object-in vs. page-out. Since swap-out events can occur at any time with the runtime path uninformed, Atlas enforces that pages containing objects whose dereference scopes are actively executed cannot be swapped out. This is because if such pages are swapped out before their dereference scopes finish, these objects may be fetched back in immediately from the runtime path, requiring pointer updating. Pointer updating cannot be done when the raw pointers of these objects are active on the stack. As a result, these pages cannot be swapped out until none of their objects are executing their dereference scopes.

Algorithm 1: Atlas Pre-Scope Barrier (Simplified).

```
/* derefcnt > 0 precludes the page's swap-out     */
atom_inc(find_page_meta(addr).derefcnt)
if not tsx_check_local(addr) then     /* Remote object */
    if take_runtime_path(addr) then   /* Runtime path */
        new_addr ← find_addr(addr, this.size)
        /* Inc/dec the new/old page's derefcnt     */
        atom_inc(find_page_meta(new_addr).derefcnt)
        atom_dec(find_page_meta(addr).derefcnt)
        alloc_copy_update(addr, new_addr, this.size)
        this.metadata.addr ← new_addr
        addr ← new_addr
    end
    else                                 /* Paging path */
        * (char*) addr
    end
end
```

Algorithm 2: Atlas Post-Scope Barrier.

```
atom_dec(find_page_meta(this.addr).derefcnt)
```

Atlas achieves this by maintaining a per-page *deref count*, which is incremented when any object on the page enters a dereference scope and decremented when the scope finishes. Any page with a non-zero deref count is skipped when the kernel looks for swap-out victims. Note that this does not create much impact on performance because the pages whose objects are actively used are usually hot pages and unlikely to be selected as swap-out victims anyway.

One issue that may arise from this protection is a potential live lock on the object-fetching path: either an ill-defined large dereference scope or many active dereference scopes in a parallel application may potentially lead to too much data getting pinned in local memory, which may result in out-of-memory errors. To tackle this issue, Atlas monitors the pinned data and forces the flipping of their containing pages' PSFs (to use paging) upon memory pressure. Once these pages are swapped out, they will be paged in—this solves the problem as page-in does not need pointer updating.

Invariant #3: Dereference scope vs. evacuation. Evacuation threads may move an object while another thread is executing the object's dereference scope. This must not occur because evacuation requires pointer updating, which cannot be done when a dereference scope is being executed (and raw pointers are used). To this end, Atlas uses the page's deref count to synchronize between evacuation threads and dereference scopes. A non-zero dereference count prevents the page from being evacuated.

Compared to AIFM, Atlas employs a slightly different definition of dereference scope. AIFM chose to decouple dereference scopes from the barrier—it allows one dereference scope to cover multiple smart pointer dereferences, serving as a coarse-grained fence between application threads and the evacuator. On the contrary, Atlas employs *fine-grained* dereference scopes, each of which is associated with one single smart pointer dereference. This choice was made based on our observation of frequent evacuations; using coarse-grained dereference scopes would require constant synchronizations between application and evacuation threads, leading to performance and latency impact. Fine-grained dereference scopes not only reduce the degree of blocking but also help alleviate potential live locks. Although a finer granularity increases barrier overhead, this overhead is often amortized by a large number of raw pointer accesses and computation within each scope. A detailed overhead analysis can be found in §5.2 and §5.4.

With the invariants discussed above, we proceed to presenting our barrier logic, which is shown in Algorithm 1 and Algorithm 2. As illustrated in Figure 3, Atlas executes Algorithm 1 and Algorithm 2 at the beginning and the end of a dereference scope, respectively.

Pre-scope barrier. Atlas first atomically increments the deref count for the page containing the object (Line 1). This indicates that the page has an object whose dereference scope is being executed, preventing the paging system from swapping out the page (*i.e.*, Invariant #2). This step must be done before the barrier starts to guarantee that (1) if the page is local, it cannot be swapped out from this point on, or (2) if the object is remote, once it is fetched in, its containing page cannot be swapped out.

Atlas uses Intel's TSX [32] to efficiently check if the address `addr` is local. Atlas starts an RTM transaction, which contains nothing but a dereference of the object. If the object's containing page is unmapped, the RTM transaction will abort with a special status captured by Atlas, which verifies the status by checking with the kernel. This hardware-based check is ∼14× faster than a purely software-based approach that relies on a system call that walks the page table and checks whether the page is local based on its PTE. A `true` value (*i.e.*, the object is local) returned by TSX directs the execution to exit the barrier immediately. Otherwise, Atlas checks the PSF corresponding to the address (Line 3) to decide whether this access should take the runtime (Lines 4-9) or the paging path (Line 12).

Using TSX to check object location may introduce false positives—a transaction may abort even if data is local. Since such cases are rare (*e.g.*, less than 1/10000 in our experiments), Atlas takes an optimistic approach to handle them. Upon a TSX abort, Atlas sends an RDMA read to access the remote object and simultaneously issues a page table walk to verify the object's location. If the verification fails (indicating the object is local), the fetched object is discarded. This approach introduces only a negligible overhead (*i.e.*, a small number of unnecessary RDMA reads).

Runtime path. `take_runtime_path` in Algorithm 1 checks the PSF of the page corresponding to `addr` and returns `true` if the PSF is `runtime`, indicating that object fetching should be performed. For ease of presentation, Algorithm 1 is significantly simplified to *not* show details of how to synchronize between threads to guarantee the absence of race condition when multiple threads fetching the same object. Atlas first finds a new address to which the object will be moved (Line 4). Since this address is on a new page, before moving the object, the deref count of the new page must be incremented (Line 5) to ensure that from this point on, the new page cannot be swapped out until the dereference scope finishes (*i.e.*, Invariant #2). The barrier also needs to decrement the deref count of the old page (Line 6) that was incremented earlier in Line 1.

Next, Atlas fetches the object by allocating a new object of the same size (using our log-structured allocator discussed in §4.3), copying the object's data into the new object, and updates its pointers (Line 7). Atlas subsequently changes the `addr` field of the pointer to the new address (Line 8). Pointer updating is done by retrieving the object's pointer from its header and updating their addresses, in a way similar to how it is done in AIFM. If it is a shared pointer, all other pointers will be retrieved from the main one and updated accordingly. The object's `is_moving` field is used to synchronize between pointer updating events performed by multiple threads. The synchronization details are omitted for simplicity. After the object is moved to a local page, future accesses to the object will follow the PSF of the new page.

Paging path. The paging path simply touches the object (Line 12) to ensure that the page fault handling is *completed* after the execution passes this line.

Post-scope barrier. The post-scope barrier has much simpler logic, as shown in Algorithm 2. All it needs to do is to atomically decrement the page's deref count, indicating the finishing of the dereference scope. When its deref count becomes zero, this page is subject to swap-out again (*i.e.*, Invariant #2).

4.3 Memory Management

Atlas's heap is composed of a *normal-object* space, a *huge-object* space, a *metadata* space, and an *offload* space. Atlas manages the normal-object space via a log-structured allocator [57, 58] and maintains a background evacuator to reduce fragmentation by compacting live objects. Atlas does not handle huge objects that cannot fit into a page, placing them into the huge-object space and delegating their management to the kernel directly since they are too large to benefit from object-level management. Metadata such as CATs are accessed by both the runtime and paging system, and hence, it is shared between the user and kernel space. The offload space stores objects whose functions can be offloaded to the remote side. We will discuss it shortly.

Object allocation. The log-structured allocator maintains thread-local allocation buffers (TLAB) to reduce the global lock contention during parallel object allocation. The TLAB is managed at the granularity of log segment which is aligned with a page to guaranteed that no object can go cross the page boundary. Atlas allocates objects contiguously on the TLAB as prior research [65, 70] shows that objects allocated close in time exhibit similar usage patterns. In doing so, objects with temporal proximity are naturally grouped into the same log segment (page), enhancing locality.

Metadata allocation. Metadata such as dereference counters and card tables is allocated in a dedicated metadata space. Atlas maintains a card table for each page to record the object access information. Each card table is a bitmap where each bit represents a consecutive range of 16 bytes. Our experiments show that the sizes of most objects are larger than 8 bytes, making 16 bytes a natural choice for the card size. Each card table is allocated and initialized during the allocation of a log segment. It is freed along with the log segment. The space needed by the card tables is 1/128 of the total memory. In summary, the space overhead is less than 2%.

Object evacuation. The log-structure allocator [58] supports defragmentation via a copying-based evacuator, a technique widely used in modern garbage collectors [20]. In Atlas, we extend the evacuator to improve the temporal locality of pages by grouping hot objects into contiguous log segments (pages) during the evacuation. The evacuator runs concurrently with the application to reduce fragmentation.

The evacuator periodically scans log segments and evacuates a log segment with a high garbage ratio by copying its live objects to a newly allocated target segment. As a result, the target segment is free of fragmentation, and the source log segment can be freed right away. When moving an object, the evacuator maintains its corresponding card table values, *i.e.*, if the object was recently accessed on the source page, the evacuator marks its card bit on the target page during evacuation. Furthermore, Atlas improves evacuation efficiency by prioritizing log segments in local memory and delaying the processing of remote log segments until they are accessed or the free space runs out [67].

The Atlas runtime tracks whether an object has been accessed since the last evacuation via the `access` bit in the smart pointer (see Figure 2). This bit is set by the read barrier when the object is dereferenced and cleared by the evacuator at the end of each evacuation. The evacuator segregates objects that have been accessed since the last evacuation into a set of contiguously allocated log segments. We found this approach to be particularly effective in improving temporal locality for real-world workloads with *skewness* (*e.g.*, 90% of accesses hit 10% objects). The `access` bit allows Atlas to distinguish hot and cold objects in such workloads, leading to a substantial performance boost. Note that this operation is significantly more efficient than maintaining an object-level LRU for eviction. As opposed to ranking objects based on hotness, Atlas's `access` bit simply serves as an evacuation location indicator. Its functionality is similar to CAT but used differently; CAT is read and cleared by the kernel at page eviction while the `access` bit is read and cleared by the runtime at evacuation.

Computation offloading. As shown in many existing far-memory systems, such as Semeru [66], Mako [43], AIFM [57], and Mira [26], offloading memory-intensive operations to the remote side can effectively reduce the data movement overhead. A unique challenge for Atlas is how to enable offloading when paging is used. Under paging, remote memory is managed as a swap partition of a set of swap slots. These slots are agnostic about the remote server's memory addresses. Pointer addresses contained in a page are with respect to the compute server while the page can reside at a completely different address on the remote server. This address mismatch precludes the correct execution of a function on an object directly on the remote server.

To solve the problem, Atlas uses an approach that is similar to Semeru [66]—we reserve a dedicated offload space in the heap. Developers need to explicitly define remoteable data structures and functions (which are similar to those in AIFM). Objects registered as *remotable* are all allocated into this space. Pages in this space have guaranteed virtual address alignment between the compute and remote servers—we modify the paging system to ensure that a page at a virtual address A on the compute server is guaranteed to be still at address A on the remote server when evicted. Atlas requires users to guarantee a remotable data structure cannot reference a non-remotable object. This property ensures address consistency when a function is called remotely.

The offload space is an *object-in, page-out* space, which allows objects to be fetched only through the runtime. This is due to the need to synchronize between the servers for safe remote execution. When a remote function is being invoked on an object, the `offload` field in its smart pointer is used for synchronization—the runtime can not fetch the object until the remote function is finished (and the `offload` bit is cleared). Remotable objects can only be fetched into the offload space to ensure the above-stated properties.

5 Evaluation

5.1 Setup and Methodology

We wrote 7,675 lines of C/C++ code to implement Atlas's runtime library, and added support in the Linux kernel (version 5.14-rc5) for page management (*e.g.*, path synchronization). We ran experiments with one compute server and one memory server connected by a 200 Gbps Infiniband switch. Each server has 2 Intel Xeon Gold 6342 CPUs (24 physical cores each), 256 GB of memory, and a 100 Gbps Mellanox ConnectX-5 InfiniBand adapter. All evaluated systems ran on Ubuntu 18.04. We configured the servers following common practice for low latency [52], disabling Turbo Boost, CPU frequency scaling, and transparent huge pages.

Baselines. Atlas was implemented based on Fastswap and AIFM. For the paging path, Atlas uses unmodified Fastswap with added tasks of profiling and synchronization. For the runtime path, Atlas uses AIFM's ingress algorithm and paging at egress. For evaluation, we used AIFM [57] and Fastswap [9] as our baselines for object fetching and paging, respectively. For Fastswap, we ran the original applications to avoid unnecessary runtime overhead. For AIFM, we used the performance-tuned versions of applications, where all optimizations were enabled including per-thread access pattern tracking, object hotness tracking, and non-temporal programming hints [57]. We turned off offloading when evaluating throughput and latency, leaving its evaluation to §5.4.

Workloads. As shown in Table 1, we evaluated six real-world applications and two synthetic applications, including Metis [44]—an optimized MapReduce framework for multicore architectures, Aspen [17]—a purely functional tree-based graph processing framework, GraphOne [34]—a data store for real-time analytics on evolving graphs, as well as Memcached [5]—an in-memory key-value store. We ran Memcached with two different workloads: a real-world workload (MCD-CL) that comes from Meta's cache system CacheLib [12] and a synthetic workload (MCD-U) generated by YCSB [15] that follows a uniform distribution. We also employed two synthetic applications developed by AIFM's authors to compare Atlas and AIFM. These applications include one batch application, DataFrame [46], and one latency-critical application, WebService.

Covering a wide spectrum of domains and memory access patterns (*i.e.*, sequential, random, skewed, and mixed patterns), these applications can be divided into four categories:

First, both Memcached workloads exhibit random access patterns, leading to significant I/O amplification under paging. The real-world workload MCD-CL has a high level of skewness with *churn* behaviors. *Churn* refers to the phenomenon that hot data in the working set changes rapidly over time. On the contrary, the synthetic workload MCD-U demonstrates completely random behaviors, with no skewness and hot data. As a result, MCD-CL is more amenable to Atlas's dynamic locality improvement than MCD-U.

Second, GraphOne and and Aspen are evolving graph systems, which are representatives of applications that perform analytics over frequently updated datasets. GraphOne uses adjacency lists and edge lists to store an input graph while Aspen utilizes compressed purely-functional trees to store a graph, which supports a higher update rate. The working sets of these applications change continuously. Their accesses are very complex: the first stage builds the graph in memory, exhibiting a random pattern. The second stage runs iterative algorithms where the first iteration does not have locality and thus performs random accesses; the subsequent iterations would enjoy better locality if it runs on Atlas, which dynamically improves the locality during the first iteration. However, updates to the input graph disrupt the locality and hence there can also be many random accesses in the middle of the iterations. We used these two graph frameworks to evaluate how well Atlas can dynamically adjust the data layout and improve locality.

Third, Metis (MapReduce) and DataFrame represent bulk data processing systems with clear phase-changing behaviors (discussed in §3). These workloads are used to evaluate whether Atlas can accurately recognize access patterns and switch to the proper data path. DataFrame is additionally used to evaluate compute offloading due to its memory-intensive operations (§5.4).

Finally, WebService is an interactive web application exhibiting mixed access patterns, from random, pointer-chasing, to sequential accesses.

For Atlas to run these applications, we modified 263 lines of code for Metis, 278 lines for Aspen, 219 lines for GraphOne, and 391 lines for Memcached; the additional code was used to declare smart pointers and dereference scopes. It took one developer a few hours to port each program.

Memory setup. Each application was run with five local memory configurations: 13%, 25%, 50%, 75% and 100%, each representing a specific percentage of an application's working set that can fit into local memory. These configurations were enforced using `cgroup`. The first four configurations were employed to evaluate the performance of the three systems when using different amounts of remote memory, while the 100% (all local memory) configuration was used to assess the runtime overhead of Atlas and AIFM, introduced by the barriers (for smart pointer dereferencing), dereference trace recording (for object-level prefetching), and evacuation (for defragmentation), as well as other bookkeeping overheads; see Table 2 for more details.

5.2 Throughput

We first measured the throughput of the applications with varying local memory ratios. Overall, Atlas outperforms Fastswap and AIFM, respectively, by 3.2× and 1.5×, over the eight real-world applications using remote memory (from 13% to 75% local memory). When running locally (100% local memory), Atlas and AIFM incur an overall overhead of 19.1% and 14.0%, respectively, of which 10.2% and 2.3% are from the barriers. This section reports the overall performance and runtime overhead. We show a detailed overhead breakdown in §5.4.

MCD-CL and MCD-U. Both workloads were configured with the same operation ratios, *i.e.*, 87.4% get and 12.6% set. As shown in Figure 4(a), for a highly-skewed workload like MCD-CL, both Atlas and AIFM outperform Fastswap (by 6.4× and 3.2×, respectively). The performance difference comes primarily from the reduced I/O amplification—Fastswap fetches 26× and 30× more data than Atlas and AIFM, respectively, resulting in wasted memory (for storing unused data) and significantly more swaps. Under 100% local memory, Atlas and AIFM introduce an over-

Application	Dataset	Size	Characteristics
Memcached CacheLib [5] (**MCD-CL**)	Meta CacheLib [12]	50M records	Skewness with churn
Memcached Uniform (**MCD-U**)	Synthetic, uniform distribution [15]	50M records	Random access
GraphOne PageRank [34] (**GPR**)	Twitter 2010 [35]	1.5B Edges, 41.7M Vertices	Evolving graph
Aspen TriangleCount [17] (**ATC**)	Friendster [73]	1.8B Edges, 65.6M Vertices	Evolving graph
Metis Word Count [44] (**MWC**)	The News Crawl Corpus [72]	5.1GB	Phase-changing
Metis PageViewCount (**MPVC**)	Wikipedia English [6]	15GB	Phase-changing with mixed patterns
DataFrame [46] (**DF**)	NYC Taxi [3]	16 GB	Phase-changing with offloading
Web Service [57] (**WS**)	Synthetic [57]	10GB hashmap, 16GB array	Mixed patterns with offloading

Table 1: Applications used for our evaluation.

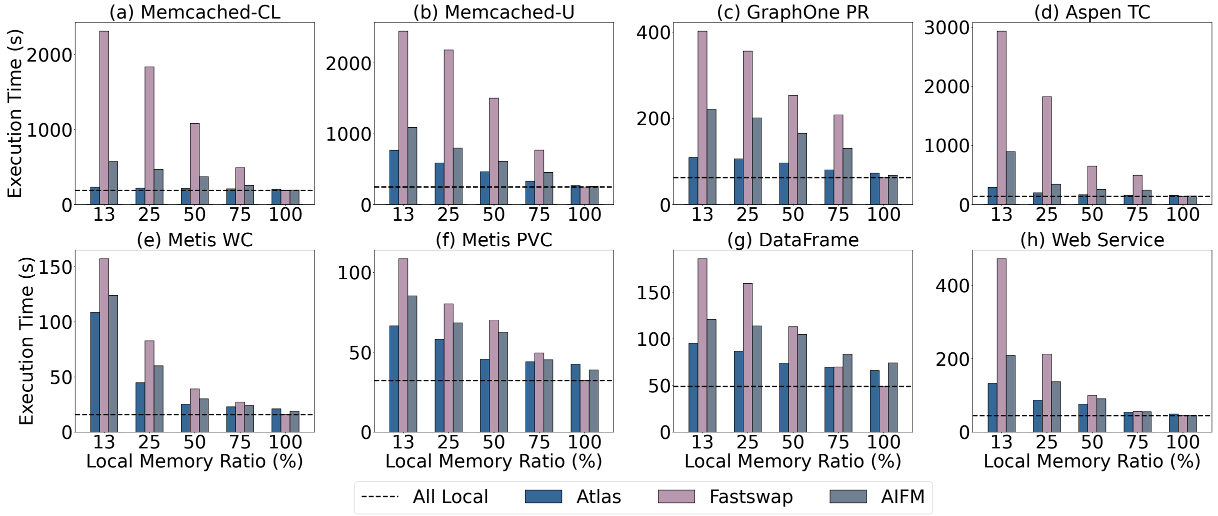


Figure 4: Throughput comparison between Atlas, Fastswap and AIFM with varying local memory ratios. "All Local" lines represent the performance of unmodified applications under 100% local memory.

all overhead of 9.0% and 3.2%, respectively, compared to Fastswap. The primary source of the overhead is the barriers, taking 6.2% and 1.5% of the execution time, respectively. Given that Memcached spends a substantial portion of its execution on communication, the barrier overhead, which is associated with the in-memory processing, is insignificant.

Compared to AIFM, Atlas further improves the performance by 1.2×, 1.8×, 2.2×, 2.5×, under the four different memory configurations (75%, 50%, 25%, and 13%). This improvement stems from a much higher eviction throughput (on average 4.6× higher) in Atlas due to the elimination of object eviction. In addition, Atlas's concurrent evacuator (§4.3) improves the temporal locality by segregating hot objects into contiguous pages, leading to an overall of 18% more accesses that go through the paging path (§5.4). This result was achieved when AIFM used 20 eviction threads while Atlas only used one single swap-out thread in the paging path. MCD-U performs random accesses with no hot data, hindering opportunities for Atlas to improve locality. Hence, the usefulness of the hybrid data plane is limited. However, Atlas still outperforms AIFM by up to 1.4× due to more efficient eviction, as shown in Figure 4(b).

GPR and ATC. To execute an evolving graph engine, we divided the input datasets [35] into three batches, which are incrementally fed to the graph engine. For each batch, the graph engine conducts the following three steps: load the updates, update the graph, and execute the analytics.

As Figure 4(c) shows, in the presence of remote memory, Atlas outperforms AIFM and Fastswap by an average of 1.8× and 3.1×, respectively, on GPR. As stated earlier, graph updating and the first iteration of analytics exhibit random access patterns. As such, GPR's throughput under AIFM is 1.7× higher than under Fastswap. For Atlas, when the analytics starts, objects are accessed and reordered by the object fetching in the first few iterations; in the subsequent iterations, pages storing edge objects are switched to using the paging path due to the gradually established locality. As a result, up to 82% of pages have their PSFs changed during the execution (from object fetching to paging), as demonstrated in Figure 7(b). This improves the analytics throughput.

ATC's computation stages and access patterns are both similar to those of GPR. For ATC, the trees storing the graph data are dynamically reorganized by Atlas's runtime path, leading to ∼38% of pages changing their PSFs (from object fetching to paging). In addition, evacuation improves locality by segregating hot objects from these trees into a few pages, reducing remote memory accesses by 24%. As demonstrated in Figure 4(d), ATC's overall throughput is 2.0× higher under Atlas than under AIFM.

When running on 100% local memory, Atlas's barrier overheads for both GPR and ATC are modest, 8.2% and 4.3%, due to the high ratio between raw pointer accesses and smart pointer dereferences. Oftentimes, one object dereference (*e.g.*, obtaining a vertex that contains a series of edges) is followed by dozens of raw pointer accesses (*e.g.*, to individual edges). Each dereference scope contains an average of *21* raw pointer accesses. In addition, for ATC, the barrier overhead is further diluted due to its higher computation and memory access costs (from poor spatial locality).

MWC and MPVC. Figure 4(e) and (f) respectively show the performance of MWC and MPVC. As discussed in §3, MPVC exhibits a two-phase behavior that can benefit from adaptive path switching, leading to a 1.2× and 1.4× improvement, compared with AIFM and Fastswap, respectively. MWC has a similar two-phase behavior with MPVC but exhibits more random accesses in its map phase, resulting in almost no page that can be flipped to `paging`. Compared to AIFM and Fastswap, MWC has 1.2× and 1.5× performance improvement, respectively.

For these two applications, the runtime overhead is relatively high—32.0% (Atlas) and 19.2% (AIFM), under 100% local memory. These two Metis workloads are both memory-intensive—they keep scanning data with high parallelism, leading to both high barrier overhead and profiling overhead (*e.g.*, for card profiling and access trace recording, see §5.4). Atlas's barrier overhead reaches up to 16.1% and 17.4% for MPVC and MWC, respectively, which are about 4× higher than that of AIFM.

DF. DF is a table-structured in-memory data structure with hundreds of columns and millions of rows, popularized in Pandas [48]. Users can slice data in different ways and run various statistics. As Figure 4(g) shows, Atlas outperforms AIFM by 1.2∼1.4× in the four remote-memory settings. We ran a client, developed by the AIFM authors, to conduct a series of *Copy* and *Shuffle* operations on *DF*. Similarly to Metis, *DF* demonstrates clear phase-changing behaviors when processing different operations—a *Copy* operation copies data from a column, exhibiting excellent spatial locality and a clear sequential pattern, while a *Shuffle* operation reorders rows for each column, exhibiting random patterns. Atlas achieves superior performance to AIFM and Fastswap, due to its adaptive access path selection.

AIFM suffers a higher runtime overhead (51.4%) compared to Atlas (34.7%) despite having a lighter barrier. The reason is that AIFM maintains a remote vector on the memory server for every DataFrame vector to support the eviction of individual objects with varied sizes. During the execution, DataFrame vectors keep getting allocated and resized. As a result, the remote data structure also needs to be frequently resized to maintain a valid mapping from local objects to their remote memory locations. Resizing is a heavy operation as it requires allocating memory and moving all existing objects. Therefore, it becomes a major source of overhead, which can take two-thirds of the runtime overhead under 100% local memory. On the other hand, under Atlas, eviction is handled by the Linux kernel at a fixed page size and there is no need to maintain any remote data structures. Note that frequent resizing of data structures was not observed in other applications. For example, for WS, the hash table array is allocated at the start of the application and its size remains fixed throughout the execution.

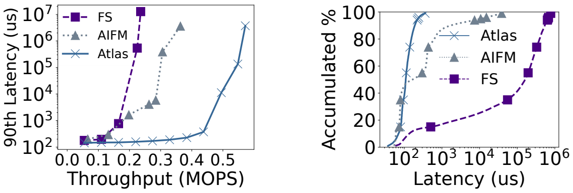
WS. WS is implemented by AIFM's authors to simulate a distributed workload. Each client (thread) sends 32 requests to look up keys in an in-memory hash table and fetches a single 8KB element from an array. This element is then encrypted with Crypto++ [7] and compressed using Snappy [23] before being sent back to the client. We use a 26GB dataset for the evaluation, which is consistent with the dataset used in AIFM [57]. Client requests are generated by following a Zipfian distribution.

As Figure 4(h) shows, compared to AIFM, Atlas improves WS' performance by an average of 1.3× with remote memory. This is due to an extremely large number of objects on the LRU list that must be analyzed by AIFM. AIFM's performance degradation is primarily due to the compute resource contention between application and evacuation threads (discussed in §3), making it hard for evacuation threads to quickly identify and evict cold objects. Consequently, AIFM ends up evicting arbitrary objects to reclaim memory, resulting in data thrashing. By using paging for eviction, Atlas improves the eviction throughput by 5.8×, lifting data eviction efficiency to 5.9 cycles/byte, which is 7.4× higher than that of AIFM (43.7 cycles/byte).

Atlas and AIFM have relatively low overhead for WS due to the coarse-grained data fetching (8KB element) and the subsequent compute-intensive encryption. As a result, Atlas and AIFM introduce a 10.1% and 1.9% runtime overhead under 100% local memory, respectively.

5.3 Latency

This section evaluates the latency distribution using the two latency-critical applications: WS and MCD-CL. The 25% local memory ratio was used in these experiments.



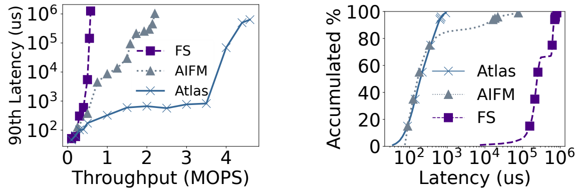
(a) 90th latency-throughput curve.

(b) Latency CDF.

Figure 5: (a) 90th latency as a function of throughput; (b) Latency CDF under 0.23 MOPS offered throughput. FS stands for Fastswap.

Web Service (WS). Figure 5(a) compares the tail latency among the three systems. Fastswap's tail latency rapidly grows due to page thrashing caused by severe access amplification. AIFM reduces amplification so that requests are less blocked by eviction. Despite the reduced amplification, AIFM still has to rank and evict individual key-value pairs, and hence the system saturates at 0.36 MOPS.

Atlas fetches individual key-value pairs initially via the runtime path and places those pairs which belong to the same request together on the same page (because these KV pairs are accessed close in time). As the execution progresses, Atlas switches to paging that can load multiple key-values pairs at the same time. Meanwhile, page-level eviction continuously offers a much higher eviction throughput so that it never blocks swap-ins. As a result, Atlas's tail latency stays low until 0.45 MOPS and can finally reach a peak throughput of 0.57 MOPS. As shown in Figure 5(b), the latencies of AIFM and Atlas are comparable until the 50th percentile, where the application starts accessing many remote objects leading to increased object management overhead. On the contrary, due to the optimized data layout which enables the efficient use of paging, Atlas experiences fewer remote accesses.

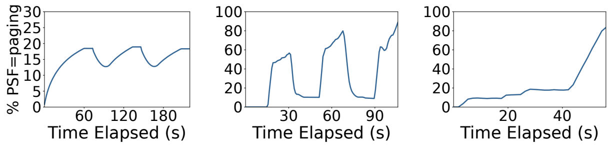


(a) 90^{th} latency-throughput curve. (b) Latency CDF.

Figure 6: (a) 90^{th} latency as a function of throughput; (b) Latency CDF under 1 MOPS offered throughput. FS stands for Fastswap.

MCD-CL. Memcached CacheLib is similar to Web Service as they both access key-value pairs from a hash table. The difference is that every request key in MDC-CL follows a Zipfian distribution, as opposed to accessing key-value pairs always in groups of 32. Figure 6 compares the tail latency among the three systems. It is clear that Atlas outperforms the other two systems. In addition to the same reasons explained above, MCD-CL is a skewed workload and hence a substantial portion (40%) of the improvement comes from the evacuation that groups hot objects in contiguous pages, making these pages amenable to paging.

5.4 Performance Drill Down



(a) MCD-CL (b) GraphOne PR (c) MPVC

Figure 7: The percentage of pages with PSF=`paging` in the memory footprint changes with the elapsed execution time.

Adaptive path switching. To understand the effectiveness of Atlas's adaptive path switching, we measured the percentage of the pages whose PSF is `paging` during the execution. Figure 7 demonstrates how this percentage changes during the execution for three applications: Memcached CacheLib (MCD-CL), GraphOne Pagerank (GPR) and Metis PageViewCount (MPVC). As Figure 7(a) shows, the number of pages that go through the paging path rises and falls over the time due to the *churn* behavior in MCD-CL discussed in §5.1. Since the workload is highly skewed, most accesses fall on a small number of hot objects, which stay in local memory and are moved into contiguous pages (with a high CAR) until the hot spot shifts.

As discussed in §5.1, the execution of GPR has experienced three batches of updates to the input graph, each of which contains two steps: graph building and analytics. During graph building, applying edge-level updates exhibits random access patterns, which can disrupt locality and leave many pages with a low CAR; these pages would have to go through the object fetching path. However, the subsequent analytics (like PageRank) runs multiple iterations; Atlas can quickly improve locality in the first few iterations, making pages turn their PSF to `paging` in subsequent iterations. This pattern can be clearly seen in Figure 7(b).

MPVC has a clear two-phase behavior (see Figure 1(a)) which can be accurately recognized by Atlas —the number of pages that go through the paging path increases dramatically as the phase change is detected by Atlas (shown in Figure 7(c)). To understand the individual contributions of object fetching and evacuation to the locality, we disabled the `access` bit tracking and let the evacuator move live objects without guidance. This reduces the overall percentage of pages that go through paging by 4% on average.

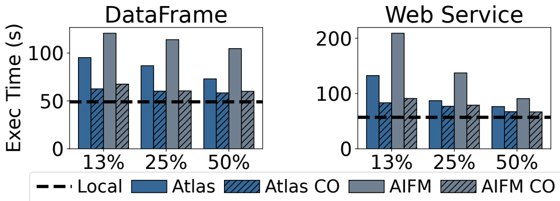


Figure 8: Throughput comparisons of DataFrame (DF) and Web Service (WS) when Atlas and AIFM enable compute offloading. CO stands for variants with compute offloading.

Computation offloading. We compared the offloading performance between Atlas and AIFM using DF and WS. Figure 8 shows the results of Atlas and AIFM with and without offloading. 18 cores were reserved on the remote side for both Atlas and AIFM, which is consistent with the offloading settings used by AIFM [57]. For DF, we offloaded the memory-intensive operations, *i.e.*, `Copy` and `Shuffle`, to the remote side. For WS, we offloaded the heavyweight array processing (on the 16GB data array). Compared to the setting where offloading is disabled (Figure 4 (g) and (h)), the throughputs of Atlas and AIFM are both dramatically improved (by up to 1.5× and 1.9× for DF, and 1.6× and 2.3× for WS, respectively), due to reduced remote accesses and data movement. On the other hand, Atlas and AIFM achieve comparable performance. This is because Atlas focuses on fetching efficiency; offloading reduces the need for fetching, making Atlas's benefit less significant.

Runtime overhead analysis. To understand the performance penalty introduced by the runtime of Atlas and AIFM, we break down and compare the runtime overhead by sources. When running with all local memory, the runtime overhead of Atlas and AIFM can be divided into five major components, listed in Table 2. Note that the overhead reported here represents the **worst-case scenario** for Atlas when compared against AIFM. When there is remote memory, part of Atlas's runtime overhead can be eliminated by switching to the paging path—dereference trace profiling is not used for paging as its goal is to analyze dereference traces for prefetching objects. Meanwhile, AIFM incurs more profiling overheads that do not exist under the all local memory setting, such as maintaining the object-level LRU for eviction.

Sources of overhead	Functionality	Affected systems
Barrier (Dereferencing)	Correctness guarantee, such as location check & synchronization	Atlas and AIFM
Card Profiling	Offering data path switching hints.	Atlas
Dereference Trace Profiling	Offering object-level prefetching hints	Atlas and AIFM
Evacuation	Defragmentation	Atlas and AIFM
Remote Data Structure Management	Managing object-level eviction	AIFM

Table 2: Major types of runtime overheads, operations involved in each type, and their affected systems.

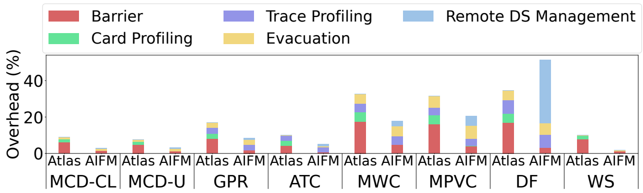


Figure 9: Runtime overhead breakdown: overhead is calculated as the ratio between the extra execution time introduced and the execution time under 100% local memory.

As shown in Figure 9, compared to Fastswap, the extra tasks in Atlas incur a runtime overhead of 7.7-34.7%, while AIFM's overhead is 1.9-51.4%. The overall overheads of the two systems are 19.1% and 14%, respectively. The primary source of overhead for both systems is the barrier (except for DF with AIFM, for which the reasons are explained in §5.2). Specifically, the Atlas barrier accounts for half of the total overhead (∼10%), and its cost is 4.4× of that of AIFM. Note that this overhead correlates with an application's memory access behavior: the most memory-intensive applications suffer the heaviest barrier overhead (MWC, MPVC, DF).

Although Atlas uses a heavier barrier, it underperforms AIFM by *only 4% under 100% local memory*. The reason is three-fold: (1) the barrier overhead is effectively amortized across the computation and raw pointer accesses (§5.2); (2) AIFM's use of coarse-grained dereference scopes leads to higher synchronization costs than Atlas; and (3) there are other operations that also contribute to the runtime overhead. Since the first item has been discussed earlier in this section, here we elaborate on the second and third items.

The barrier conducts two basic tasks, object location checking and synchronization. For location checking, Atlas has a much higher overhead than AIFM due to the use of TSX to detect an object's location whereas AIFM checks a bit on each reference. However, for synchronization, AIFM's coarse-grained dereference scopes incur a higher cost, which effectively reduces the performance gap between the barriers of the two systems. After selecting the victim segments, AIFM's evacuator must wait until all application threads exit their dereference scopes to avoid compacting objects being accessed through raw pointers. This design does not work well for big data applications with high object allocation rates, such as MWC, MPVC and Memcached. On the contrary, Atlas's fine-grained dereference scope design enables evacuation threads to skip the segments (each aligned to a page in Atlas) whose *deref count* is non-zero (indicating they are being used in active dereference scopes) instead of blocking the whole evacuation, leading to significantly reduced synchronization efforts. In fact, Atlas's CPU yield rate caused by synchronization is *an order of magnitude lower* than that of AIFM due to our non-blocking design.

Another major source of overhead is the dereference tracing (to provide prefetching hints), accounting for 14% and 19% of the total overhead for Atlas and AIFM, respectively. Among our applications, DF, MWC, MPVC and GPR use array data structures which are amenable to prefetching. As a result, there is a relatively high tracking overhead (accounting for 34% overhead on average) for both Atlas and AIFM. Other applications such as WS and Memcached use hash maps and small objects as their data structures, which are not as amenable to prefetching as arrays. Hence, for most of their memory accesses, the locations are not tracked and their tracing overhead is much lower. Note that with remote memory, the dereference tracing overhead is significantly lower under Atlas than under AIFM because a large amount of data (*e.g.*, up to 82% for GPR) goes through the paging path, which utilizes the lightweight page-level prefetcher.

CAR threshold. Figure 10 shows the influence of CAR threshold on the throughput of three applications. Picking the right CAR threshold is a tradeoff between fetching efficiency and resource waste. We used 80% as the CAR threshold for flipping PSF in our evaluation. A higher CAR is often too conservative. For example, in the case of MCD-CL, when the threshold is set to 100%, we observed that few pages can be flipped to `paging`. Therefore, most remote objects still have to be fetched individually instead of fetched in batches with page faults, leading to a 25% decrease in throughput. On the contrary, a lower CAR may result in

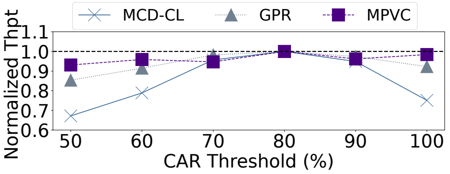


Figure 10: Sensitivity of the CAR threshold.

premature use of paging, leading to I/O amplification. As shown, the best performance is achieved when the threshold is between 80% and 90%. As such, we used the lower bound 80% based on the observation that the bandwidth of a modern network such as InfiniBand [49] is already high and will only become higher in the future, making it possible to transfer (slightly) more data with little overhead.

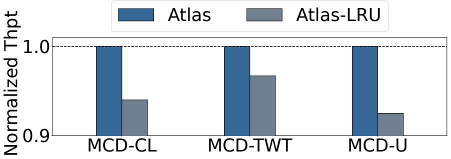


Figure 11: Normalized throughput of Memcached workloads running on Atlas and Atlas-LRU under 25% local memory.

Hotness tracking. Atlas uses an `access` bit on each smart pointer to segregate hot and cold objects during evacuation, offering benefits to workloads that exhibit skewness. We evaluated the effectiveness of Atlas's `access` bit with three skewed workloads, *i.e.*, highly-skewed (Meta, MCD-CL) [2], moderately-skewed (Twitter, MCD-TWT) [74] and uniformed without skewnewss (MCD-U) [15]. We compared Atlas with a baseline (Atlas-LRU) equipped with an LRU-like policy from CacheLib [12], which represents a more accurate approach to identifying hotness.

As shown in Figure 11, Atlas's single-bit design outperforms the LRU-like design by 7.5%, 3.3% and 6.0%, respectively. The LRU-like policy trades compute resources for accuracy by maintaining the logical ordering of objects via a linked list. Each dereference triggers a promotion that moves the object to the head of the LRU list. In order to reduce the overhead, we adopted *flat combining* [30] (to reduce thread lock contention) and ignored the dereferences of an object within 10s (to reduce promotion frequency for extremely hot objects) [12]. However, although an LRU-like policy can reduce the frequency of remote access, it incurs a maintenance overhead of up to 9% due to a huge number of objects.

Of course, the more bits used, the higher accuracy they bring. Atlas allows developers to customize the hotness tracking policy with the two reserved bits in each smart pointer (Figure 2). For our applications, we did not observe significant performance variations between using one and two access bits—likely the ability of distinguishing hot and cold objects is not increased much with two access bits.

6 Related Work

Disaggregation. Resource disaggregation has become a trending architecture for datacenters to improve resource utilization. Its key idea is to break the server hardware boundary and unstrand idle resources of remote servers by leveraging advanced network hardware [22, 28]. Existing systems have demonstrated the viability of disaggregated storage [33, 38], accelerators [50, 63, 75], network [60], and memory [25, 59]. For a memory-disaggregated system, memory spans across multiple servers. The efficient data path of Atlas can speed up the data transfer between servers.

Paging-based far memory. A practical way to deliver far memory is to leverage the paging system to access far memory. Google and Meta have reported their successful deployment of such systems in their datacenters [36, 71]. Many optimizations to the kernel data path have been proposed for improved efficiency, including but not limited to bypassing the block layer [9, 55], prefetching more accurately [45], and reducing interference [68]. The design of Atlas is orthogonal to the underlying paging systems and can directly benefit from optimizations within these systems.

Object-based far memory. Many runtime libraries offer new primitives for object-granularity far memory management, making them a more efficient alternative for scattered data on far memory. For example, AIFM [57] proposed remoteable data structures, FaRM [18] offered key-value interfaces, and Grappa [47] builds a software distributed memory. Atlas focuses on the cooperative use of its two data paths and benefits directly from existing optimizations.

Emerging hardware. Emerging hardware technologies unlock new opportunities for efficient far memory. Clio [27], StRoM [61], and RMC [10] offload functionalities to their customized hardware to reduce network traffic. Finally, CXL [16, 24, 39, 40, 77] and Project PBerry [13, 14] enable far memory access at the cache-line granularity. Atlas directly benefits from the throughput and latency advancements of new hardware technologies. Besides, for hardware solutions with a fixed access granularity, Atlas can improve data locality to improve data transfer efficiency.

7 Conclusion

We present Atlas, a hybrid dataplane that enables efficient far memory for bulk data and scattered objects simultaneously. Atlas outperforms both the state-of-the-art object-based and paging-based far memory systems.

Acknowledgement

We thank the reviewers for their comments and are particularly grateful to our shepherd Malte Schwarzkopf for his feedback. This work is supported by National Key Research and Development Plan of China under grant 2022YFB4500400, National Natural Science Foundation of China under grant 62090024, US National Science Foundation under grants CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254, as well as supports from Cisco and Tencent Big Data.

A Artifact Appendix

A.1 Overview

Atlas is a kernel-runtime co-designed system to enable a hybrid remote memory data plane. The artifact includes the custom Linux kernel and the runtime library to enable Atlas-managed applications. To run the artifact, two servers with Intel CPUs connected by InfiniBand are required. The server running the application is the CPU server, while the other server providing remote memory is the memory server. Detailed instructions can be found in Atlas code repository.

A.2 Checklist

- **Hardware:** Two servers with Intel CPUs with TSX, connected by InfiniBand
- **Software Environment:** Ubuntu 18.04, 20.04 or 22.04, with the specified version of MLNX_OFED driver and provided Linux kernel described below
- **Public Link to Repository:** https://github.com/wangchenxi7/Atlas
- **Code License:** MIT License

A.3 Building the Linux Kernel

```
## all operations are performed on both
servers unless specified
cd linux-5.14-rc5
cp config .config
sudo apt install -y build-essential
bc python2 bison flex libelf-dev
libssl-dev libncurses-dev libncurses5-dev
libncursesw5-dev
./build_kernel.sh build
./build_kernel.sh install
./build_kernel.sh headers-install
## edit GRUB_DEFAULT="Advanced
options for Ubuntu>Ubuntu, with Linux
5.14.0-rc5+", or whatever the new kernel
version code is
## edit GRUB_CMDLINE_LINUX="nokaslr
transparent_hugepage=never
processor.max_cstate=0
intel_idle.max_cstate=0 tsx=on
tsx_async_abort=off mitigations=off"
sudo vim /etc/default/grub
sudo update-grub
sudo reboot
```

A.4 Setting up InfiniBand Connection

```
## use Ubuntu 18.04 as an example below
wget https://content.mellanox.com/ofed/
MLNX_OFED-5.5-1.0.3.2/MLNX_OFED_LINUX-5.5-
1.0.3.2-ubuntu18.04-x86_64.tgz
tar xzf MLNX_OFED_LINUX-5.5-1.0.3.2-
ubuntu18.04-x86_64.tgz
cd MLNX_OFED_LINUX-5.5-1.0.3.2-
ubuntu18.04-x86_64
sudo apt install -y bzip2
sudo ./mlnxofedinstall
-add-kernel-support
sudo /etc/init.d/openibd restart
sudo update-rc.d opensmd remove -f
sudo sed "s/# Default-Start:
null/# Default-Start:  2 3 4 5/g"
/etc/init.d/opensmd -i
sudo systemctl enable opensmd
sudo service opensmd start
## assign IPs to InfiniBand interfaces on
both servers
sudo nmtui
```

A.5 Building Atlas Runtime

```
## use gcc-9
cd atlas-runtime/third_party
git clone -depth 1 -b
54eaed1d8b56b1aa528be3bdd1877e59c56fa90c
https://github.com/jemalloc/jemalloc.git
cd ../bks_module/remoteswap
## on memory server
cd server && make
## on CPU server
cd client && make
cd ../../bks_drv && make
cd ../..  && mkdir build && cd build
cmake ..  && make -j
```

A.6 Running Atlas Applications

```
cd atlas-runtime/bks_module/remoteswap
## on memory server
cd server
##./rswap-server <memory server IB ip>
<memory server IB port> <memory pool size
in GBs> <CPU server core count> e.g.,
./rswap-server 172.16.16.1 9999 48 96
## on CPU server
cd client
## edit `mem_server_ip`,
`mem_server_port` and
`SWAP_PARTITION_SIZE_GB` to be consistent
with memory server parameters
vim manage_rswap_client.sh
bash manage_rswap_client.sh install
## run a test
cd atlas-runtime/build/tests/
runtime/unique_ptr
bash test.sh ./unique_ptr_test
```

References

[1] Facebook Folly RCU Library. https://github.com/facebook/folly/blob/main/folly/synchronization/Rcu.h.

[2] Meta CloudLib. https://cachelib.org.

[3] Nyc taxi trips - exploratory data analysis. https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook.

[4] std::weak_ptr. https://en.cppreference.com/w/cpp/memory/weak_ptr.

[5] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[6] Konect networks data. http://konect.cc/networks/, 2021.

[7] free c++ class library of cryptographic schemes. https://www.cryptopp.com, 2022.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[9] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[10] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 3844, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Apache. Apache cassandra. https://cassandra.apache.org, 2021.

[12] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[13] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[14] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Compute express link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[17] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019. Association for Computing Machinery, 2019.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[19] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[20] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9, 2016.

[21] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini. Memory-harvesting VMs in cloud platforms. In *ASPLOS*, pages 583–594, 2022.

[22] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[23] Google. Google's fast compressor/decompressor. https://github.com/google/snappy, 2020.

[24] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[25] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[26] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692708, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417433, New York, NY, USA, 2022. Association for Computing Machinery.

[28] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 260269, New York, NY, USA, 2008. Association for Computing Machinery.

[30] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355364, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Intel Corporation. Transactional Synchronization Extensions. In *Intel® 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture*, pages 16–1, Santa Clara, CA, 2021. Intel Corporation.

[32] Intel Corporation. XBEGIN. In *Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z*, pages 5–611, Santa Clara, CA, 2021. Intel Corporation.

[33] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash $\approx$ local flash. In *ASPLOS*, pages 345–359, 2017.

[34] P. Kumar and H. H. Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, Feb. 2019. USENIX Association.

[35] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[36] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[37] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[38] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[39] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, pages 574–587, 2023.

[40] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[41] Y. Li, R. Melhem, A. Abousamra, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, 2010.

[42] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[43] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 92107, 2022.

[44] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical report, Massachusetts Institute of Technology, 5 2010.

[45] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[46] H. Moein. C++ dataframe for statistical, financial, and ml analysis. https://github.com/hosseinmoein/DataFrame, 2020.

[47] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

[48] I. NumFOCUS. Pandas. https://pandas.pydata.org/, 2022.

[49] NVDIA. Nvidia connectx infiniband adapters. https://www.nvidia.com/en-sg/networking/infiniband-adapters, 2023.

[50] Nvidia. Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[51] Oracle. The java virtual machine. https://www.java.com/en/download, 2023.

[52] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, volume 32, 2019.

[54] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in numa architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 369380, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, Apr. 2023. USENIX Association.

[56] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[57] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, 2020.

[58] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 116, USA, 2014. USENIX Association.

[59] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[60] Y. Shan, W. Lin, R. Kosta, A. Krishnamurthy, and Y. Zhang. Optimizing hardware-based network computation dags for multiple tenants with supernic. *arXiv preprint arXiv: Arxiv-2109.07744*, 2021.

[61] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[62] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[63] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352367, New York, NY, USA, 2022. Association for Computing Machinery.

[64] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 237251, New York, NY, USA, 2017. Association for Computing Machinery.

[65] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 347362, 2019.

[66] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280. USENIX Association, Nov. 2020.

[67] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 35–53, 2022.

[68] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[69] C. Wang, Y. Shan, P. Zuo, and H. Cui. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology*, 38(5):949–969, 2023.

[70] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173, 2015.

[71] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609621, New York, NY, USA, 2022. Association for Computing Machinery.

[72] WMT. Statistical and neural machine translation. https://statmt.org, 2011.

[73] J. Yang and J. Leskovec. Friendster social network and ground-truth communities. https://snap.stanford.edu/data/com-Friendster.html, 2012.

[74] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, 2020.

[75] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach. Automatic virtualization of accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 5865, New York, NY, USA, 2019. Association for Computing Machinery.

[76] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[77] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.

[78] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, 2015.

DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency

Haoran Ma[†⋆] Yifan Qiao[†] Shi Liu[†⋆] Shan Yu[†] Yuanjiang Ni[Ψ] Qingda Lu[Ψ] Jiesheng Wu[Ψ]
Yiying Zhang[‡] Miryung Kim[†] Harry Xu[†]

UCLA[†] *UCSD*[‡] *Alibaba Group*[Ψ]

Abstract

Despite being a powerful concept, distributed shared memory (DSM) has not been made practical due to the extensive synchronization needed between servers to implement memory coherence. This paper shows a practical DSM implementation based on the insight that the ownership model embedded in programming languages such as Rust automatically constrains the order of read and write, providing opportunities for significantly simplifying the coherence implementation if the ownership semantics can be exposed to and leveraged by the runtime. This paper discusses the design and implementation of DRust, a Rust-based DSM system that outperforms the two state-of-the-art DSM systems GAM and Grappa by up to **2.64**× and **29.16**× in throughput, and scales much better with the number of servers.

1 Introduction

The concept of distributed shared memory (DSM) received significant attention during the early years of distributed computing systems. This era witnessed a plethora of pioneering efforts, as exemplified by seminal works such as [10, 16–18, 31, 36, 49, 50, 56, 61–63, 80]. DSM offers the power of parallel computing using multiple processors and machines and, more crucially, streamlines the development of distributed applications with a unified, contiguous memory view.

The initial enthusiasm for DSM was tempered by significant performance bottlenecks, primarily due to the low network speeds prevalent during its nascent stages. Recent advances in hardware and networking technologies [3, 7, 12, 19, 23, 29, 33, 38, 40, 42, 46, 51, 54, 64, 66, 74, 78] have revitalized the DSM explorations. Several new DSM systems [14, 45, 60, 77, 81, 88] were proposed in recent years to take advantage of these enhanced networks. However, these systems are still far from achieving satisfactory performance, exhibiting poor scalability and substantial slowdown compared to their single-machine counterparts. This is mainly due to the intensive synchronization operations needed to ensure memory coherence across servers.

State of the art. The majority of existing DSM systems [6, 14, 45, 88] adopt an approach to achieve data consistency by adhering to the following invariant: for each data block to be accessed, the block is either located on a single node with potential read and write access, or it is replicated across multiple nodes with each having read access only. Prior to a server attempting to access a block, a DSM system checks the state of the block, invalidates copies of that block on all other servers, and then transmits the block to the requesting server. This synchronization process necessitates multiple network round trips. Even with RDMA, the incurred latency is still orders of magnitude higher compared to a single local access, significantly degrading overall performance. Effectively reducing the number of synchronizations is, therefore, crucial for minimizing DSM overhead and rendering it feasible for real-world deployment.

A practical strategy to minimize synchronization overhead involves implementing high-level protocols to guarantee exclusive access for each server. For instance, Apache Spark [91] utilizes an immutable data structure known as a resilient distributed dataset (RDD) for distributed access. However, RDD only facilitates coarse-grained distributed access, limiting each server to accessing a distinct partition of an RDD. While increasing access granularity enhances performance, it comes at the expense of reduced generality—Spark is tailored for bulk processing of batch data and is incapable of supporting distributed applications requiring object-level accesses, such as social networks where objects of various types and sizes (*e.g.*, images, connections, *etc.*) are created and manipulated upon each user request.

Insights. Our main observation is that synchronization overheads in existing DSM systems are introduced primarily due to the use of a generic approach that overlooks semantic information from programs. For example, many real-world concurrent programs are engineered with a single-writer-multiple-reader (SWMR) discipline to ensure correctness during concurrent operations. Leveraging such information

[⋆] Part of the work was done when Haoran Ma and Shi Liu interned at Alibaba Group.

can potentially eliminate the need to check the state of remote data blocks before accessing them, leading to dramatically improved performance. A major challenge is, however, how to expose such semantics in a sensible way so that the DSM system can see and act upon it.

One approach to convey such semantics, as demonstrated by AIFM [73] and Midas [68], involves exposing APIs that developers can invoke to specify program regions accessible only by a single writer. However, this process is cumbersome and error-prone, demanding a profound understanding of potential executions and involving substantial program writing. Our key insight in this endeavor is that the SWMR programming paradigm aligns seamlessly with *ownership types*, which have already been integrated into programming languages like Rust [75]. Rust is widely employed in the system community for dependable and secure implementation of low-level systems code.

Rust's ownership type inherently upholds SWMR properties in any compiled Rust program. The fundamental concept behind the ownership type is that each value is ensured to have a single unique variable as its owner throughout the execution. While multiple references to a value are allowed, only the owner and mutable references can modify the value. Moreover, only one of these references is permitted to be used for modifying the value at any given point.

When developing a DSM system on top of an ownership-based language like Rust, SWMR semantics are inherently embedded in any Rust program *by design*. Effortlessly extracting such information becomes possible with basic compiler support, sparing developers from the need for code rewriting. Utilizing the SWMR semantics from the program leads to a considerably simplified process for accessing data in DSM. In the case of a write access, the ownership type ensures exclusive access to the data. Consequently, DRust can move the data to the requesting machine, performing the write there without explicitly invalidating its copies on other machines. In the case of a read access, data can be efficiently replicated to (and cached in) each requesting machine, benefiting from the compiler-provided assurance of freedom from concurrent writes.

This paper presents DRust, an efficient Rust-based DSM implementation that enables object-level concurrent accesses by leveraging the SWMR semantics made explicit by Rust's ownership type. DRust automatically turns a single-machine Rust program into a DSM-based distributed version *without requiring code rewriting*. While extracting the ownership semantics appears straightforward, leveraging it to implement a distributed coherence protocol correctly and efficiently presents two main challenges.

The first challenge is *how to manage memory correctly and efficiently*. Rust's ownership type system is inherently designed for a single-machine environment, where the memory address of an object remains constant post-creation. This assumption is disrupted in a distributed environment, where objects may be migrated or duplicated on different machines. Such actions can lead to the risk of dangling pointers, potentially breaking memory coherence.

To tackle these issues, DRust builds a global heap spanning multiple servers based on the idea of partitioned global address space [21]. Each object in the heap has a unique global address in the address space, which can be used for accessing the object from any server. DRust re-implements Rust's memory management constructs to allocate objects in the global heap. Given that a server can have cached objects (to accelerate reads), DRust carefully crafts an ownership-based cache coherence protocol upon the global heap abstraction to achieve both memory coherence and efficiency (§4.1.1).

In a nutshell, our coherence protocol leverages the ownership semantics to eliminate the need for explicit cache invalidation. It allows multiple readers to fetch a copy of the object from its host server and cache it, but disallows any change to the global address and the value of the object. When a write access occurs, it must first borrow the ownership, at which point DRust moves the object in the global heap to a new address on the server issuing the write. The address change of the object automatically invalidates cache copies that use the stale address and triggers the subsequent readers to update the cache by fetching the object from its latest address.

The second challenge is *how to support transparency in programming*. Rust's standard libraries and programs were originally built for running on a single machine, and they cannot deal with distributed resources in a cluster. For example, a Rust program running on server A cannot spawn a thread on another server B, let alone synchronize threads between A and B. To enable a Rust program to run *as is* under DRust, we provide distributed threading utilities by restructuring critical elements of the Rust standard library, including threading, communication channels, and shared-state locks (§4.1.2). Our adapted libraries offer the same interfaces, making them compatible with single-machine Rust programs, but internally invoke our distributed scheduler, which determines where to run the thread and facilitates cross-server synchronization. We built them atop the ownership-based memory model, enabling the DRust runtime to safely pass references of objects between threads and automatically fetch the value from the global heap upon dereferencing.

With our programming abstractions, a Rust application can start on a single server and gradually spawn its threads to other servers. Under the hood, DRust employs a runtime to manage distributed physical compute and memory resources for the application. The runtime runs as a process on each node in the cluster, and they work cooperatively for cross-server memory allocation and thread scheduling. The runtime prioritizes the current server for object allocation and thread creation, but it will schedule the resource allocation request to another server under memory pressure (§4.2.1). To make cluster-wise decisions such as deciding the target server for global memory allocation and thread creation, DRust has a global controller that is launched together with the application. The global controller communicates with DRust runtime on each node to collect resource usage information and applies adaptive policies to achieve load balance (§4.2.2).

Results. We evaluated our system on four real-world applications in an eight-node cluster. Our evaluation demonstrated an average of 2.02× and 9.48× (up to 2.64× and 29.16×) speedup compared with two state-of-the-art DSM systems GAM and Grappa, respectively. Furthermore, DRust incurred a mere 2.42% slowdown compared to the original Rust program on a single machine with sufficient resources. DRust is available at `https://github.com/uclasystem/DRust`.

2 Background in Ownership

Over the past decades, numerous programming languages have been designed to provide safe memory management and data sharing. At the core of such a design is often a tradeoff between memory abstraction level and management efficiency. The ownership concept, and the Rust programming language built upon, are considered promising solutions that achieve a sweet spot between abstraction and efficiency. This section provides an overview of these techniques and explains how ownership can benefit DSM implementations.

Ownership Type. The ownership model has a long history in pursuit of memory-safe language designs and type systems [8, 9, 27, 43, 58, 83]. It has also inspired many systems for safe and efficient resource management [13, 41, 59, 89]. At a high level, ownership enhances a language's type system in a way that guarantees the memory and thread safety of a program with type checking done at compile time. The ownership model encompasses a range of concepts, among which the most important are *lifetimes* and *borrowing*.

An ownership-based type system uses lifetimes to control the allocation/deallocation of objects. It enforces that each object must have one and only one owner at a time. This allows the compiler to statically track an object's lifetime via its owner, and immediately deallocate the object once its owner goes out of scope, preventing memory leaks without using garbage collection that can introduce disruptive pauses to program execution.

To access an object, a program can create a reference from its owner, but the reference must "borrow" the permission from the owner, and "return" it to the owner after the access. Specifically, the type system allows the creation of multiple *immutable references* to an object from its owner for concurrent reads but prohibits any write with these references. It allows only one mutable reference to the object only when no other (mutable or immutable) references exist. Through borrowing, the ownership type disallows simultaneous writers and hence prevents data races. In addition, references must return the borrowed permission when they go out of scope. For any program that demonstrates type soundness, the type checker guarantees that references to an object can only reside within the object's lifetime; the object can be safely and automatically deallocated when its owner goes out of scope, by which time it has already lost all its references.

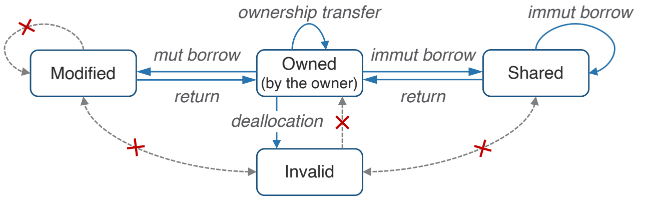


Figure 1: State machine for Rust's ownership-based memory model.

Finally, ownership can be transferred from one owner to another—*e.g.*, at a function call, the creation of a thread, or message passing (*i.e.*, via `channel`). However, the type system enforces that ownership transfer must occur in the absence of "borrowing". In other words, no other references can exist in scope when transferring the ownership, preventing data races during ownership transfers.

The guarantees provided by the ownership model with respect to object lifetime and data sharing can be summarized with the following four invariants:

1. **Singular Owner**: each value has one single owner at any time (which must also belong to one single thread).
2. **Safe Borrowing**: All references are created from the owner; permission borrowing and returning guarantees that references that can be used to access the object must be valid.
3. **Single Writer**: Each object allows one mutable reference at most, and it cannot coexist with any other references in the same scope.
4. **Multiple Reader**: Multiple references are permitted only when all of them are immutable.

The last two invariants are commonly called the single-writer-multiple-reader (SWMR) property in the DSM literature [57].

Rust Language. Rust offers a practical implementation of ownership and is designed with a range of zero-cost abstractions for efficient fine-grained resource management. Figure 1 depicts the state machine for Rust's ownership-based memory model. At a high level, this model restricts that the owner is always in the O (owned) state, and transitions between M (modified), S (shared), and I (invalid) must go through the O state[1]. Clearly, a distributed implementation of this approach avoids broadcasts or snooping, and only requires peer-to-peer message passing.

Listing 1 exemplifies a simple accumulator implemented in Rust (Lines 1–7). The `Accumulator` struct keeps an integer `val` and exposes an interface `add` to increment the value. Rust uses a smart pointer type `Box<T>` to store values on the heap; this pointer serves as the initial owner of the referenced value, as shown in Line 10 and 11. Line 13 instantiates `Accumulator` `a`, where the ownership is implicitly transferred from `val` to `a.val` during its initialization. Rust allows the creation of mutable and immutable references to access the value. For

[1] A transition from M to S is also possible as an optimization in Rust.

```
1  pub struct Accumulator { pub val: Box<i32>, }
2  impl Accumulator {
3    pub fn add(&mut self, delta: &i32)->i32 {
4      *self.val += *delta;
5      *self.val
6    }
7  }
8  fn main() {
9    // Allocates two integers in the heap.
10   let val: Box<i32> = Box::new(5); // val is an owner.
11   let mut b: Box<i32> = Box::new(0); // b is an owner.
12   // Ownership is transferred from val to a.val
13   let mut a = Accumulator{val};
14   { // Only one mutable reference is allowed.
15     let mutr: &mut i32 = &mut *b;
16     // No other reference is allowed now.
17     /* let another_r = &*b; */ // COMPILE ERROR!
18     *mutr = 10; // b == 10
19   }
20   { // Multiple immutable references are allowed.
21     let (b_r1, b_r2): (&i32, &i32) = (&*b, &*b);
22     // Mutable reference is prohibited now.
23     /* let b_mutr = &mut *b; */ // COMPILE ERROR!
24     // Passing by references won't transfer ownership.
25     let sync_add = a.add(b_r1); // a.val == 15
26     let sync_add = a.add(b_r2); // a.val == 25
27   }
28   {// Ownership of a and b is moved to the new thread.
29     // No reference should or can borrow a or b now.
30     let async_add = thread::spawn(move ||
31       a.add(&*b) // a.val == 35
32     ).join(); // lifetime of a and b ends
33     // Current thread cannot access a and b anymore.
34     /* println!("{}", a.val); */ // COMPILE ERROR!
35   }
36 }
```

Listing 1: A simple accumulator implementation in Rust.

example, Lines 14–19 create a singular mutable reference (`&mut`) to `b` and set its value to 10. Similarly, Lines 20–27 create two immutable references (`&`) to `b` and add them to `a` via two function calls. Note that passing references as arguments in function calls does not transfer their ownership.

Finally, Rust allows spawning new threads for concurrent programming, as shown in Lines 28–35. A new thread is created via `thread::spawn`, where the use of `move` captures `a` and `b` in the current scope and transfers their ownership to the newly spawned thread. Rust performs shallow copying for inter-thread communication, where only the pointers stored in `a` and `b` are transferred to the child thread while the actual values on the heap are not moved. Rust guarantees memory safety of `a` and `b` by tracking their ownership. At Line 32, when the child thread finishes its closure (*i.e.*, not necessarily after `join`), and `a` and `b` exit the scope (to which their ownership belongs), their lifetimes terminate and Rust deallocates them from the heap.

3 Motivation

DSM was proposed to eliminate the barrier of distributed programming by offering the same memory consistency model as single-machine shared memory. The core of its design is a software-based cache coherence protocol, which mimics a hardware-based approach on multi-core CPUs and synchronizes memory states on different servers by sending control messages between them. However, it is notoriously hard to implement cache coherence efficiently at the software level due to the high communication latency between physically disjointed servers.

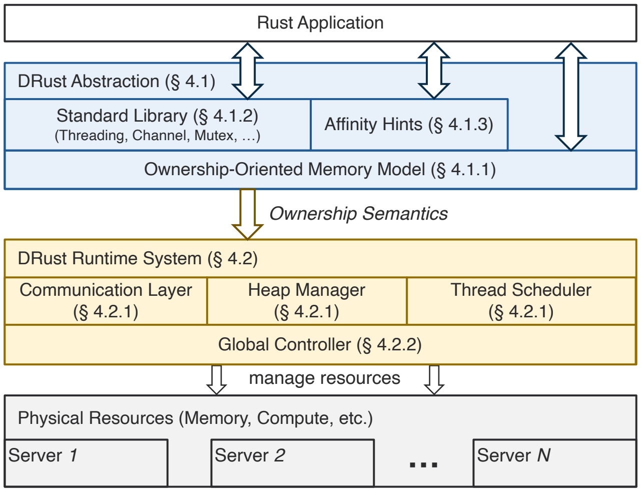

Rust Application
DRust Abstraction (§ 4.1)
Standard Library (§ 4.1.2)
(Threading, Channel, Mutex, ...)
Affinity Hints (§ 4.1.3)
Ownership-Oriented Memory Model (§ 4.1.1)
Ownership Semantics
DRust Runtime System (§ 4.2)
Communication Layer
(§ 4.2.1)
Heap Manager
(§ 4.2.1)
Thread Scheduler
(§ 4.2.1)
Global Controller (§ 4.2.2)
manage resources
Physical Resources (Memory, Compute, etc.)
Server 1
Server 2
...
Server N

Figure 2: Design overview of DRust.

High Synchronization Overheads for Coherence. To gain a high-level understanding of how much improvement can be achieved by improving the cache coherence protocol, we performed an analysis by running a real-world application DataFrame [67] with a state-of-the-art DSM system GAM [14] with a fast network. We first ran Dataframe on a single server with 16 CPU cores and 64GB memory. We then ran it with GAM on eight servers connected by a 40Gbps Infiniband network by evenly distributing the same amount of resources to eight servers (*i.e.*, each server uses 2 CPU cores and 8GB memory). Our experiments show a **2.4**× slowdown when Dataframe runs on eight servers.

A detailed examination reveals that such a slowdown stems primarily from its complicated coherence protocol. GAM runs a directory-based protocol, which assigns each DSM cache block a home node. Upon each object read/write, the home node tracks the state of its cache block and updates all cache copies for the state change, incurring extensive computation and network overhead. We broke down the average time spent on each component when accessing one object in the DSM. Reading a 512-byte (*i.e.*, GAM's default cache block size) uncached object in GAM takes 16μs, while the actual time to read the object over the network is only 3.6μs. In other words, maintaining cache coherence takes **77%** of the total time. This large memory access overhead significantly increases operation latency, hindering the practical deployment of distributed shared memory. With the single writer invariant inherent in the ownership model, we expect that most of this overhead can be eliminated, leading to significant ($>2\times$) speedups for each access.

4 Design

DRust is an efficient DSM framework atop the Rust programming language. As shown in Figure 2, it consists of

```
1  // Unmodified Rust code.
2  pub struct Accumulator { pub val: Box<i32>, }
3  impl Accumulator {
4    pub fn add(&mut self, delta: &i32)->i32 {
5      *self.val += *delta;
6      *self.val
7    }
8  }
9  fn main() {
10   // Allocates two integers in the distributed heap.
11   let val: Box<i32> = Box::new(5);
12   let b: Box<i32> = Box::new(10);
13   let mut a = Accumulator{val};
14   // a.val and b will be fetched to local.
15   let local_add = a.add(&*b); // a.val == 15
16   // Only refs to a and b are shipped to remote.
17   let remote_add = thread::spawn(move ||
18     a.add(&*b)).join(); // a.val == 25
19 }
```

Listing 2: DRust seamlessly transforms an unmodified accumulator implemented in Rust into a distributed version.

Rust-based programming abstractions for DSM (§4.1) and a runtime (§4.2) that manages distributed physical resources.

DRust is compatible with standard Rust. Listing 2 illustrates how the accumulator (shown in Listing 1) runs on DRust distributively without requiring code rewriting. The program starts running on a single machine A and the DRust runtime gradually allocates its memory and spawns new threads on different machines. Specifically, Lines 10–13 create `Accumulator` `a` and `b` where `a.val` and `b` are in the global heap. We use a global allocator to allocate objects in the global address space and hence these objects may be allocated on a different server. Line 15 synchronously adds `b` to `a` by fetching both values `a.val` and `b` to A's local memory (if they are allocated somewhere else). Line 17 spawns a new thread and ships the function closure to perform `add` asynchronously. This thread will be scheduled on a different server B if A's compute power has been saturated. In this case, DRust performs shallow copying and only ships the pointers stored in `a` and `b` to B without actually moving objects in the global heap. The newly-created thread relies on the DRust runtime to detect data locations and fetch objects upon dereferencing.

4.1 DRust Programming Abstraction

DRust provides each thread with a local stack and abstracts distributed memory as a shared global heap. Each server allocates thread stacks and backs one partition of the global heap with its physical memory. DRust re-implemented core memory management constructs including `Box`, `&`, and `&mut` for transparent heap access. This approach hides the complex details of memory allocation/deallocation, moving objects, and coherence maintenance (§4.1.1). DRust supports distributed threading and synchronization by adapting Rust's standard libraries atop the core language constructs (§4.1.2). Furthermore, DRust offers affinity annotations that allow developers to build more efficient applications by expressing data affinity semantics (§4.1.3).

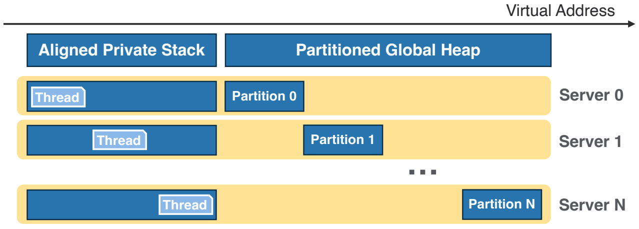


Figure 3: The address space layout of DRust. The stack is private to each thread but they share an aligned address space to ease migration, while the heap is globally shared and partitioned across servers.

4.1.1 Memory Management

Next, we discuss how DRust (re)implements the memory-related language constructs in Rust to achieve memory safety and memory coherence.

Address Space. As shown in Figure 3, DRust maintains an identical address space layout on all servers. It exposes distributed memory as a coherent shared heap to applications. Embracing the idea of partitioned global address space (PGAS) [21], it partitions the heap space and assigns each server a unique address range. The stack, in contrast, is private to each thread. However, DRust aligns the stack space on each server and pads stacks to avoid overlapping. This streamlines thread migration between servers as it allows a thread to keep its private stack address unchanged when being moved.

Coherence Protocol in a Nutshell. For efficiency, DRust employs a *call-by-reference* model for newly created threads. Upon creation of a thread, the DRust runtime only passes references or `Box` pointers to objects to the newly created thread. Upon dereferencing, objects are fetched to the server where the thread is executed.

When a *read access* of an object is issued on a server, our runtime simply fetches a copy of the object from its hosting server and places it in its *local cache*. As a result, multiple copies of the same object may exist on different servers. This allows multiple servers to read the object at the same time from their respective cached copies. Fetching a copy of the object for read does not change the object's address in the global space. When a *write access* occurs on an object, the server issuing the write must first obtain the object's write access permission through a *mutable borrow*. Our reimplementation of mutable borrow (discussed shortly) *moves*[2] *the object in the global heap to a new address* that belongs to that server. In doing so, the object's cached copies on other servers are automatically invalidated without sending explicit invalidation messages—subsequent reads on these servers must obtain an immutable reference to the object through an immutable borrow from its owner pointer, which has been updated to the new address immediately after

[2] The term "copy" is used to describe the process of adding an object into the cache without changing its global address. The term "move" means relocating the object into a server's heap partition, which requires changing its global address.

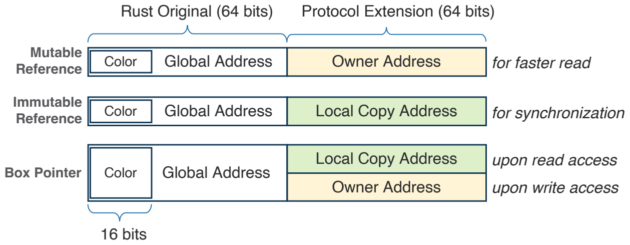


Figure 4: DRust repurposes Rust pointers and references to contain a global heap address and an extension field for its coherence protocol.

the mutable borrow returns. Upon identifying the owner's address change, each immutable borrow would direct a server to fetch a fresh version of the object from the new address as opposed to relying on a stale copy residing in its cache.

Note that this is a general protocol that covers the case that the object is on the same server that issues the write—as long as the server moves the object into a different location in the global heap, no other servers can read the stale copies of the object. However, this is not efficient as each local write requires moving the object to a new address. To address this inefficiency, DRust employs a pointer-coloring technique, inspired by the designs of many concurrent garbage collectors [1, 52]. Discussed at the end of this subsection, this technique offers a more efficient solution for handling local writes.

Pointer Layout. In order to support this protocol, each pointer must remember not only the object's global address, but also the address of the cached copy in a server's local cache (to avoid redundant remote fetches). As such, we modify Rust's pointer structure, as illustrated in Figure 4. DRust internally extends each Rust `Box` pointer and reference with an additional 64-bit field, which is used differently for read and write access. At a high level, the field records the address of the cached copy for faster read accesses; for write accesses, this field records the address of the object's owner for post-write synchronization. Additionally, DRust reserves the highest 16 bits in the global address field as "color" bits. These bits record the version number of the pointer and play a crucial role in DRust's efficient handling of local writes.

Next, we discuss how DRust reimplements Rust's ownership operations to realize the distributed coherence protocol. For ease of presentation, this subsection focuses on a simplified version of the protocol. A complete coherence protocol and its proof of memory coherence are available in [53].

Mutable Borrow. Mutable borrow creates a mutable reference that holds exclusive access to the referenced object for writing. Algorithm 1 outlines the procedures for both dereferencing and dropping a mutable reference. When performing dereferencing, DRust first checks the object's location (Line 2) and performs direct access if the object's address belongs to the heap partition of the machine A that executes the access. Otherwise, DRust *moves* it to A's heap partition (as opposed to caching it) (Line 3). The move, conducted in the following three steps, changes the object's global address. DRust (1) copies the object into A's heap at an address p, (2) updates the mutable reference with the address p, and (3) asynchronously requests the remote server that previously stored the object to deallocate the original object.

Algorithm 1: Access logic for mutable references.

```
Input: A mutable reference $m$ containing a global address
       $m.g$ and the owner address $m.o$.
Output: A local memory address to be written to.
1 Function DEREFMUT ($m$):
2     if $\neg$ISLOCAL ($m.g$) then
3         $m.g \leftarrow$ MOVE (CLEARCOLOR ($m.g$))
4     return CLEARCOLOR ($m.g$)
5 Function DROPMUTREF ($m$):
6     $c' \leftarrow$ GETCOLOR ($m.g$) $+1$
7     WRITE ($m.o$, APPENDCOLOR ($m.g$, $c'$))
```

A challenge arises with its original owner `Box`, which now becomes a dangling pointer, pointing to an invalid memory location. Fortunately, the integrity of the system is maintained by the single-writer invariant (referenced as Invariant 3). This invariant ensures that while the mutable reference remains alive, no other entity, including the original owner, can access the data. To ensure correctness, when this new reference is dropped, DRust synchronously updates the original owner `Box`, redirecting it to the new address p (Line 7). As a result, the original owner always possesses the latest view of the object. Additionally, all modifications made through this mutable reference are visible in all subsequent accesses, as they necessitate borrowing permission from the updated owner `Box`. The single-writer invariant also eliminates the possibility of simultaneous updates to the owner, ensuring that updating the owner is free from concurrency issues.

Immutable Borrow. Immutable borrowing allows concurrent reads to the same object from immutable references on the same or different servers. As detailed in Algorithm 2, DRust handles the dereferencing of immutable references by first checking the object's location (Line 2). For remote objects, DRust creates a local copy in the *per-node read-only "cache"* and records its local address in the reference's extension field (see Figure 4). This preserves the original global address of the object, ensuring that any new immutable reference—whether it is derived from the owner `Box` or from another immutable reference—can always access the original object from the global heap.

As opposed to being a separate memory space, our "cache" provides a "virtual" aggregation of all local copies maintained on each server. These copies reside in the regular heap, managed by a per-node hashmap H. This hashmap maps each global address to a pair of its local address and the number of local immutable references to the local copy. To prevent redundant copies of an object on the same server, DRust checks the hashmap H before creating a new local copy

Algorithm 2: Access logic for immutable reference.

```
Input: A shared immutable reference $r$ containing a global
       address $r.g$ and a local copy address $r.l$, and a local
       cache hashmap $H$.
Output: A local memory address for reading.
1  Function Deref ($r$,$H$):
2      if IsLocal ($r.g$) then
3          return ClearColor ($r.g$)
4      else
5          if $r.l = Null$ then
6              Atomic {
7              if $r.g \in H$ then
8                  $\langle l',cnt\rangle \leftarrow$ GetEntry ($H, r.g$)
9                  $r.l \leftarrow l'$
10                 UpdateEntry ($H, r.g, \langle l', cnt+1\rangle$)
11             else
12                 $r.l \leftarrow$ Copy (ClearColor ($r.g$))
13                 InsertEntry ($H, r.g, \langle r.l,1\rangle$)
14             }
15         return $r.l$
16 Function DropRef ($r$,$H$):
17     if $r.l \neq Null$ then
18         Atomic {
19         $\langle l', cnt\rangle \leftarrow$ GetEntry ($H, r.g$)
20         UpdateEntry ($H, r.g, \langle l', cnt-1\rangle$)
21         }
```

(Line 7). If a local copy is already present, DRust increments its reference count in H and updates the extension field in the immutable reference to point to this copy (Lines 8–10). If no existing copy is found, a new one is created (Lines 12–13). Since the hashmap uses objects' global addresses as keys, if an object has been modified by another server since its last read, its global address must have changed, making cache lookup fail even if a (stale) local copy exists.

DRust actively updates the reference count of each local copy when an immutable reference is either dereferenced or dropped, as outlined in Lines 10 and 20. Utilizing these counts, the DRust runtime periodically scans the "cache" and lazily reclaims unreferenced copies (*i.e.*, those with a zero reference count) under memory pressure (§4.2.1). This mechanism, in conjunction with the safe borrowing invariant (2), prevents the local cache from memory leaks or illegal accesses.

Owner Access without Borrow. DRust treats a direct memory access via the owner `Box` as a pair of mutable/immutable borrow and return. Depending on the reference type, DRust uses the extension field of `Box` accordingly and executes the read/write dereferencing logic. A special case arises when a mutable owner is immutably borrowed and becomes immutable until all borrowed references return. In this case, the owner can only cache the object during the borrow and delay the move until the borrow finishes. This would not create any correctness issues because the owner cannot be used for write access during this period.

Algorithm 3: Utility functions for pointer coloring.

```
Function GetColor ($g$):
    return $g \gg 48$
Function ClearColor ($g$):
    return $g$ & $((1 \ll 48) - 1)$
Function AppendColor ($g$, $c$):
    return ClearColor ($g$) | $(c \ll 48)$
```

Ownership Transfer. Similar to Rust, DRust does not move the actual value during the transfer and only copies the `Box` pointer. DRust additionally checks and resets the pointer's extension field and frees the cached copy in the executing machine's cache to avoid cache leakage.

Memory Deallocation. Like Rust, DRust tracks the lifetime of an object via its owner. Given that ownership transfer is implemented by only evicting the cached copy of the object (without changing its global presence), the memory safety of DRust's global heap is preserved by the singular owner invariant (1). In other words, DRust still guarantees that when an object's owner goes out of scope, the object must be unreachable (and dead) and can be safely deallocated.

Consistency Model. Our protocol, together with Rust's ownership model, offers *sequential consistency* for cross-server memory accesses in safe Rust programs (*i.e.*, following the original Rust, no guarantees can be provided when Rust `Unsafe` is used), which is a strong consistency order. Therefore, it allows any safe Rust program to preserve its memory consistency on DSM. Sequential consistency necessitates a coherent memory system, requiring not only the SWMR invariant but also the *data-value* invariant [57]. In simple terms, the data-value invariant requires that the latest write to a value is immediately visible to subsequent readers. As discussed earlier, DRust's protocol moves an object upon a write and updates the owner immediately. Therefore, the latest value is globally visible after each mutable borrow finishes. Subsequent read accesses, either in the Owned state or the Shared state, are hence guaranteed to see the moved object and read its latest value.

Optimizing for Local Writes. A special case is that a server issues a write to an object that resides in its own heap partition. While the coherence protocol still guarantees safety, requiring moving an object in its local heap each time it is written clearly brings inefficiencies. To optimize for local writes, DRust adopts a pointer-coloring method, inspired by the design of concurrent garbage collectors in a managed runtime system such as JVM [1, 52]. Several utility pointer coloring functions are shown in Algorithm 3 which are used when dereferencing and dropping a reference. We reserve the first 16 bits of a global address as a "color". The color value stored in the object's owner gets incremented upon the expiration of a mutable reference, as detailed in Lines 6–7 in Algorithm 1. Any subsequent immutable borrow would look up the cache with the object's global address. Even if the actual address remains the same, its color changes if a write has occurred. As such, the lookup would not return any stale copy from the local cache.

The 16-bit `color` field may overflow when the pointer keeps being borrowed for local writes on the same server. DRust implements a *move-on-overflow* strategy that moves the object to a new address and resets its color to zero once the maximum color value is reached (2^{16}), thereby preventing overflow and maintaining system integrity and performance.

Writing Unsafe Code in DRust. Rust allows developers to bypass compiler safety checks and write *unsafe* code for low-level operations such as accessing raw pointers and mutating shared variables at their own risk [44, 70]. Since DRust relies on SWMR semantics enforced by Rust's ownership types, DRust ensures consistency and memory safety only in the "safe" Rust code. DRust does not cache objects in unsafe code but allows developers to implement their own cache. Developers must ensure that they do not violate consistency in unsafe code blocks where type safety is not enforced. This caution mirrors practices in other managed languages, like native code in Java and unsafe code in C#. To assist developers, DRust offers primitives such as `dalloc`, `dread`, and `dwrite` for managing data on the global DSM heap.

4.1.2 Adapting Rust Standard Libraries

To further reduce the barrier for programs to run distributively, we reimplement several standard Rust libraries atop DRust's core memory constructs covering four categories: threading for distributed computation (`std::thread`), inter-thread channel for communication (`std::sync::mpsc`), reference-counted pointers for ownership sharing (`std::sync::Rc` and `std::sync::Arc`), and shared-state locks for concurrency control (`std::sync::Mutex` and `std::sync::atomic`).

Threading. DRust's threading library enables Rust threads to run distributively with two major adaptations. First, it enables distributed thread launching by re-implementing the `spawn` interface. Internally, it captures the thread body as a closure during compile time and forwards it to the runtime. During execution, the runtime launches the thread according to each server's load (details in §4.2.1). Second, DRust performs implicit ownership transfers between the parent and the child threads at the start or the end of the child thread execution. Thanks to the distributed ownership transfer support provided by DRust's memory model, the implementation in the threading library is hidden from developers and preserves type soundness and memory safety. Additionally, DRust is compatible with advanced thread utilities such as `thread::scope`, which allows for the spawning of scoped threads that can borrow non-static data. These utilities ensure that all threads are joined at the end of their scope and can internally utilize DRust's functions for spawning and joining threads, thus extending their applicability to the distributed setting.

Inter-Thread Channel. DRust extends Rust's channel to connect two distributed threads for message passing. DRust internally builds a network-based message queue for cross-server messages. Benefiting from the shared global heap, `Box` pointers and references can be safely copied and remain valid across servers. Therefore, the sender can push an object into the channel *as is* without serialization, even if it may contain `Box` pointers. DRust forwards the object binary bytes to the receiver over the network, and the receiver can recover the object from the binary by direct type conversion without deserialization.

Ownership Sharing. Rust allows multiple owners to share an object via reference-counted smart pointers, which count the number of live owners. In this case, smart pointers only have read access, and the object lifetime terminates when all owners die and the reference count hits zero. DRust does not require special treatment for `Rc` as it only allows ownership sharing inside a single thread. For `Arc` which shares ownership among multiple threads, DRust handles it in a similar way to immutable references with on-demand local caching and lazy eviction.

Shared-State Concurrency. Rust supports shared-state concurrency, primarily through its atomics and mutexes, where threads commonly share an atomic-typed value or one mutex via ownership sharing (*i.e.*, `Arc`). Unfortunately, the ownership model cannot type check concurrent read/write to shared states. Hence, Rust relies on an *unsafe* implementation in its standard library. §4.1.1 already provides a general discussion on writing unsafe code in DRust, and here we focus on DRust's implementation for distributed shared states.

Shared states create a unique challenge for DRust, as they may be replicated on multiple servers and those states must be synchronized among these servers. For example, an `Arc<AtomicBool>` may be replicated across different servers and used independently, causing multi-version issues if not synchronized properly. DRust addresses this inconsistency by allocating the actual value on the global heap and storing only the `Box` pointer in `atomic` types. This design allows atomics to be freely moved or replicated across servers while keeping a single version of the actual value. To synchronize concurrent operations on atomics, DRust adapts methods of atomic types to forward the operation as a message to the server storing the actual value, which serializes all operations with unsafe logic similar to the original Rust to guarantee atomicity and memory consistency. Similarly, DRust implements `Mutex` by allocating its metadata and owned object on the global heap and leaving only `Box` pointers in the mutex struct. Concurrent operations on mutexes are serialized on the server storing the mutex.

```
1  pub struct Node { val: i32, next: Option<TBox<Node>>, }
2  pub struct List { pub head: Option<Box<Node>>, }
3  impl List {
4    pub fn sum(&self) -> i32 {
5      let mut total: i32 = 0;
6      if let Some(r) = &self.head {
7        let mut node = &**r; // Fetch whole list to local.
8        loop { // Iterate every list node.
9          // Accessing node is guaranteed local.
10         total += (*node).val;
11         if let Some(next) = &node.next {
12           node = &**next;
13         } else { break; }
14       }
15     }
16     total
17   }
18 }
```

Listing 3: A linked list implementation with TBox in DRust. The use of TBox ties list nodes one by one. Iterating a list will fetch all nodes together (if they are on another server), and henceforth accessing any node is guaranteed local.

4.1.3 Affinity Annotations

To further improve performance, DRust allows developers to provide optional data affinity semantics via annotations. These annotations are useful for many datacenter applications that make extensive use of object-oriented data structures that require *pointer-chasing* to access. For instance, Memcached [55] uses a chained hash table where each hash bucket stores its KV pairs with a linked list. To find one KV pair from a bucket, Memcached has to iterate the linked list following the node pointers. However, frequent pointer chasing is unfavorable in a distributed setting, where each pointer dereference incurs additional runtime checks and potential cross-server traffic. It would be beneficial for the runtime to colocate them on the same server and schedule the computation there.

Data-Affinity Pointer. To expose data affinity for clustered placement, DRust includes a new pointer type TBox for developers to “tie” a heap object to its owner. TBox shares the same interfaces as the ordinary Box and can be used as a drop-in replacement for Box. However, TBox enforces that the pointed-to object must reside on the same server as its owner. In other words, when its owner object is copied or moved, the object referenced by TBox will be copied or moved as well. TBox can be used in a nested manner to allow a large union of objects to be co-located. The DRust runtime fetches (*i.e.*, copies or moves) them together in a single batch, leading to fewer network round-trips and higher throughput. TBox can also be assigned to and owned by a stack variable, in which case the referenced object is pinned onto the heap partition of the server that hosts the stack and cannot be moved. Dereferencing a TBox is thus guaranteed to be a local access—DRust optimizes it by skipping the runtime check.

Listing 3 presents a linked list implementation using TBox. Our linked list uses TBox (Line 1) to specify the data affinity between consecutive nodes. As a result, all list nodes are chained with TBox, forming an affinity group. Line 4–17 define a sum function that calculates the total sum of all node values. Assuming the list is non-empty, Line 7 dereferences the pointer to the head node, and the DRust runtime checks the location of the head node and fetches the entire list of nodes together if they are not local. Next, accessing each node inside the loop body (Line 8–14) is guaranteed local and hence skips runtime checks. Compared to using Box directly, TBox makes both data fetching and accessing more efficient.

```
1 fn main() {
2   let val: Box<i32> = Box::new(5);
3   let mut a = Accumulator{val};
4   let remote_add = spawn_to(a.val, move ||
5     a.add(10)).join(); // a.val == 15
6 }
```

Listing 4: A distributed accumulator can leverage spawn_to to offload a thread to the server where a.val locates.

Data-Affinity Thread. To expose the affinity between data and computation for thread scheduling, DRust extends its threading library with a spawn_to interface. spawn_to mirrors the ordinary spawn interface to spawn a new thread but takes an additional Box pointer argument, which indicates where the thread should be created. The runtime checks where the Box points to and creates the new thread on that same server. A common practice to use spawn_to is to pass the mostly-accessed object as the location indicator. Listing 4 presents how the distributed accumulator (shown in Listing 2) can use spawn_to to offload a thread to the same server as a.val resides. Line 5 hence performs local dereference to a.val inside a.add().

4.2 DRust Runtime System

DRust’s runtime system is the core component that manages memory and compute resources. It includes a runtime library (§4.2.1) that is linked to each application and launched on each server and a cluster-wise global controller (§4.2.2).

4.2.1 Application-Integrated Runtime

The runtime library consists of a communication layer to support inter-server coordination and data transfer, a heap allocator to manage the heap partition and the read-only cache, and a thread scheduler to launch and schedule application threads.

Communication Layer. The DRust runtime uses its communication layer to support the cache coherence protocol and cross-server memory accesses. The communication layer consists of a control plane and a data plane, both built with RDMA. The control plane leverages *two-sided verbs* to send and receive small control messages, and the receiver can perform the coherence logic upon receiving the message to minimize the end-to-end latency. The data plane, in contrast, is specialized for bulky data transfer with one-sided verbs. It fetches an object as a whole with a single RDMA message upon pointer dereferencing without interrupting the target server, minimizing both latency and CPU usage.

Heap Allocator. The DRust runtime provides standard memory allocation interfaces and always returns global addresses to the upper-level language abstractions. It prioritizes local memory allocation as long as the local heap partition has sufficient space. This strategy improves data locality by colocating an object with its creating thread.

When the local heap partition is depleted, DRust queries the global controller and allocates memory on the most vacant server. For remote memory allocation, it forwards the request to the target server by sending a message through the communication layer and returns the allocated address to the user. Memory deallocation follows a similar logic but it bypasses the controller and finds the server directly via the object's global address. The allocator does not reserve separate space for the local cache. Instead, it manages the cache as part of the local heap partition and always allocates cached entries in the local heap partition. Under memory pressure, the allocator will scan the local cache and evict entries that are no longer used (*i.e.*, reference count hits zero).

Thread Scheduler. The DRust thread scheduler runs in the user space and schedules threads locally to maximize CPU utilization. It also provides thread migration functionalities, facilitating the global controller to balance load between busy and vacant servers.

The scheduler represents a newly created user thread as a closure, which includes a function pointer and a set of initial arguments (*i.e.*, references). It collaborates with the global controller to allocate a unique stack space for a thread (see Figure 3), and starts the thread by executing the closure.

The scheduler adopts the method of cooperative multitasking and context switches between threads *non-preemptively*. A running thread yields its control flow proactively when developers call `await` or reactively upon long-latency operations. Similarly to other systems [60, 65, 82], our scheduler handles context switches as function calls, which allow DRust to save only a few registers per thread.

The scheduler supports creating/migrating a thread to another server as well. To migrate a thread, DRust sends its function pointer, the saved register state, and its stack to the target server. Because each thread reserves its stack address range globally, DRust can copy the stack across servers without changing its address. Therefore, the thread can be easily resumed by directly calling the function pointer with the saved register state on the target server. DRust generates code for state transmission during the compile time for the scheduler to call upon thread migration.

4.2.2 Global Controller

The controller runs as a daemon process on the machine where the program is launched. It manages cluster resources and coordinates memory allocation and thread migration. It periodically pings each server to probe and record its resource usage (CPU and memory). It controls resource allocation in cooperation with the DRust runtime on each server. When allocating memory or creating a thread, the runtime will first query the controller, which chooses a target server following adaptive policies (discussed later), and then coordinate with the runtime on the target server to perform the actual operation. The controller also maintains a global table to track the location of each thread; the table is queried and updated by the scheduler when migrating a thread.

During program execution, servers may run into imbalanced loads when objects get relocated or new threads are created. DRust balances the load of each server by *migrating* threads from the busy server to less occupied ones, following an adaptive policy to minimize cross-server memory accesses. If a server is about to run out of memory (>90% memory usage), the controller keeps migrating the thread that consumes the most local heap memory until the pressure is resolved. If the server is under compute congestion (>90% CPU usage), the controller migrates threads that frequently access remote objects. The thread is then moved to the server it accesses the most unless the target server is also overloaded, in which case it moves to a vacant server instead.

4.2.3 Fault Tolerance

In DRust, the global heap can be replicated to tolerate failures. Replication creates copies for each heap partition at the same virtual address on backup servers. Threads, in contrast, are not replicated for efficiency and are only executed with the primary global heap. To maintain a synchronized view between the primary heap partition and its backup copy, a thread must additionally write back to the backup partition after each mutable borrow. However, our insight is that the thread can batch modifications to an object and delay the write-back until the object ownership is transferred to another server, which is the time point that the object becomes visible to threads on other servers. When a server with a primary heap partition fails, the controller will automatically promote its backup server to the primary and add a new backup server.

5 Implementation

The majority of DRust was implemented in Rust except for its communication library which is in C. We implemented DRust's core language constructs as three Rust types (*i.e.*, `struct`): `Ref<T>`, `MutRef<T>`, and `DBox<T>`. They serve as the counterpart for the original Rust `&T`, `&mut T`, and `Box<T>`, respectively. We implemented the coherence protocol with traits on these types, including `Copy`, `Clone`, and `Drop`, which are automatically embedded into the program source code and executed when references/pointers are created or destroyed. To support unmodified Rust programs, we changed the Rust compiler and added additional compilation passes to transform Rust references and `Box` pointers into corresponding types in DRust.

Our communication layer links `libibverbs` directly for fast and kernel-bypassing RDMA networking. We implemented a low-level C library that covers basic connection establishment and exposes high-level Rust interfaces for various RDMA verbs, including `RDMA_READ`, `RDMA_WRITE`, `RDMA_SEND`, `RDMA_RECV`, `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP`.

We primarily utilize one-sided `READ` and `WRITE` verbs for data transfers between servers, as they outperform the two-sided `SEND`/`RECV` counterparts—one-sided operations bypass the CPU and OS at the receiver side, whereas two-sided operations require the receiver to pre-post `RECV` verbs and await notification upon message arrival. For instance, when a remote object is accessed via mutable references, DRust copies the object to local memory using the `READ` verb. Upon dropping the reference, DRust updates the original owner `Box` to reflect the new address, a process executed using the `WRITE` verb. Conversely, two-sided `SEND`/`RECV` verbs are utilized for control message exchanges, such as establishing connections across servers. Atomic verbs `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP` are primarily utilized for implementing shared states (e.g., atomic types and mutexes). DRust uses the RC (reliable connection) transport type to ensure reliable transmission and strict message ordering.

Our heap allocator implementation piggybacks Rust's original allocator and aligns its virtual address range with the heap partition range. Our thread scheduler was built upon Tokio [82] for its efficient user thread and cooperative scheduling integration. The global controller is responsible for managing all threads in the cluster and padding their stacks to avoid address overlapping.

6 Limitations

DRust's design has three limitations. First, although DRust permits the use of unsafe code, its consistency guarantees are only applicable to safe Rust code. In unsafe code blocks, developers are responsible for ensuring consistency themselves. Second, DRust's superior performance relies on SWMR semantics exposed by applications. In cases where data is mostly under shared states (such as `Mutex`), DRust degenerates into a traditional DSM system; all concurrent accesses to the same data have to be centralized and serialized by the server responsible for the shared states. However, such scenarios contradict Rust's recommended programming practices. Finally, the current implementation of DRust does not support address space layout randomization (ASLR) yet, and we have temporarily disabled it. However, DRust's design is compatible with ASLR as long as DRust threads share the same randomized address space layout on each server. This can be achieved by delegating the randomization of stack and heap address allocation in DRust to its global controller, a feature that will be supported in future versions of DRust.

7 Evaluation

Setup. We evaluated our system on an 8-node cluster, where each node was equipped with dual Intel Xeon E5-2640 v3 processors (16 cores), 128GB of RAM, and a 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter, connected by a Mellanox 100 Gbps InfiniBand switch. All servers ran Ubuntu 18.04 with kernel 5.14. We disabled hyperthreading, CPU frequency scaling, OS security mitigations in accordance with common practices [69, 72].

Application	**Dataset**	**Memory (GB)**	**Comp. Intensity (cycles/byte)**
DataFrame [67]	h2oai [37]	64	110.13
SocialNet [32]	Socfb-Penn94 [71]	64	86.09
GEMM [11]	LAPACK [2]	96	300.63
KV Store [14]	YCSB [22]	48	48.15

Table 1: Applications used in the evaluation.

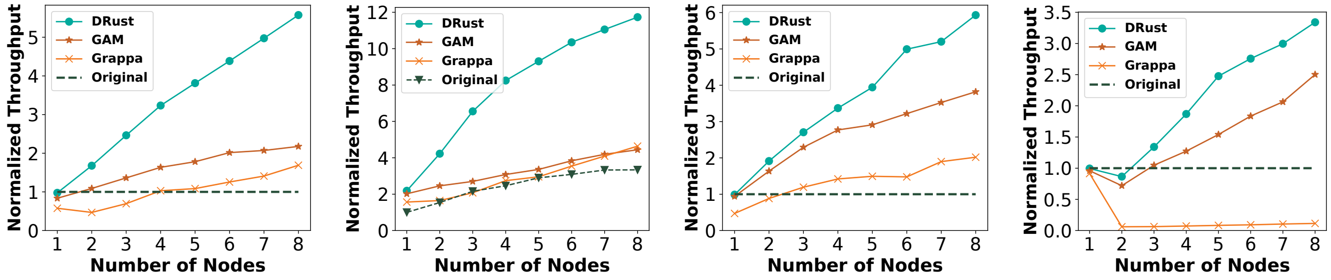
Methodology. We compared DRust with two state-of-the-art DSM systems, GAM [14] and Grappa [60]. For a fair comparison, we ported the evaluated applications to each baseline system and invested extensive effort in tuning parameters to achieve their best possible performance. GAM offers ordinary object read/write interfaces, and we exported it as a library to Rust and hooked pointer dereferencing to use GAM's API without program modification. Grappa, in contrast, offers a drastically different programming abstraction that requires rewriting the program to access shared memory via *delegation*. Therefore, we re-implemented applications in C++ and re-structured them using Grappa's abstractions.

7.1 Applications

We evaluated four representative datacenter applications covering a wide range of use cases and resource demands, including data analytics, microservices, scientific computation, and key-value storage, as shown in Table 1.

DataFrame is an in-memory data analytics framework similar to Spark [91] and Pandas [90]. We built our library atop Polars [67], a native DataFrame engine in Rust offering OLAP query APIs such as `filter`, `groupby`, and `join`. DataFrame organizes the dataset as columnar format tables in shared memory, and user queries will manipulate table columns by reading/writing rows and transforming them into new tables. DataFrame exploits data-level parallelism by internally partitioning columns by row into an array of small chunks where each chunk can be processed independently. We additionally applied `TBox` to annotate chunks from the same table column for co-location and used `spawn_to` to offload columnar operations to the data side to improve data locality and performance. Note that such annotations were not necessary for the application to run; they were added for additional performance optimizations.

SocialNet is a twitter-like web service from the DeathStarBench suite [32]. It is composed of 12 microservices with complicated call dependencies. Each microservice in SocialNet can scale independently with replicas, thereby offering higher throughput with more servers. SocialNet decouples the process of user texts, media resources, and storage into different microservices, and it employs RPCs to pass values (texts, media files, *etc.*) between them. DRust enables



(a) DataFrame (318 s) (b) SocialNet (120 ops/s) (c) GEMM (1039 s) (d) KV Store (2.7 Mops/s)

Figure 5: Application throughput when running with DRust, GAM, and Grappa, normalized to the throughput of their original implementation running on a single node. The number in the parenthesis is the original application's throughput on a single node.

SocialNet to pass only references in RPCs, eliminating the serialization/deserialization overhead and redundant data copies. Because SocialNet was implemented in C++ and deployed with Docker Swarm [28], we ported it into Rust for our evaluation. We followed its original microservice structure but changed the RPC call sites to pass references instead of values, and we followed the original orchestration configuration to spread and scale each microservice in the cluster. We did not use any affinity annotations for SocialNet.

GEMM (general purpose matrix multiplication) is a highly-optimized matrix multiplication routine from the BLAS library [11]. We ported the library using the same divide-and-conquer algorithm by recursively partitioning each matrix into small chunks for parallel processing and reducing the final results. Input and output matrices are stored in the shared memory, where each subroutine will read two input matrix chunks and write the partial results back to the output matrix. Our port strictly followed the original implementation without using additional affinity annotation.

KV Store is an in-memory key-value cache engine similar to Memcached [55]. It uses a hash table to store KV pairs in shared memory and mutexes to synchronize concurrent requests. We used YCSB benchmark [22] to generate zipf load with 90% GET and 10% SET using default skewness parameter 0.99.

7.2 Scaling Performance

In this experiment, we investigated whether DRust can speed up applications by distributing them in a cluster and how well they can scale with the number of servers used. For each application, we first ran it *as is* on a single server without using DSM and measured its throughput. Then, we ran the same application on DSM (subject to modifications when running Grappa) with the same configuration but on varying numbers of servers and measured the throughput normalized to its single-node throughput (*i.e.*, strong scaling). As GAM and Grappa cannot adaptively balance the workload across servers, we evenly distributed the application's working set and threads among all participating nodes. Ideally, an application should scale linearly and enjoy proportionally higher throughput with more nodes. However, this is usually unachievable because of the limited parallelism of real-world applications and the coherence overhead of DSM systems, and a good result for DRust will show that applications' throughput can get close to their ideal throughput.

Figure 5 shows the results for each application respectively. DRust outperforms both GAM and Grappa in all cases. On a single node, it is 1.04–2.10× faster than two baseline DSMs, while only adding a maximum overhead of 2.42% compared to the original program. When running with multiple nodes, DRust scales up applications significantly better than GAM and Grappa. On eight nodes, DRust achieves a throughput that is 1.33–2.64× higher than that of GAM, 2.53–29.16× higher than that of Grappa.

Compared to each program's single-machine performance, using DSM over DRust enables each program to easily leverage the available distributed resources and achieve a throughput that is 3.34–11.73× higher than their single-machine counterparts. Next, we discuss each application to explain the scalability difference between DRust and the baseline DSMs.

DataFrame. As shown in Figure 5a, compared with its original version, DataFrame running on eight nodes with DRust achieves 5.57× higher throughput, whereas with GAM and Grappa, the throughput improvements are 2.18× and 1.69×, respectively. In other words, DataFrame with DRust is 2.56× and 3.29× faster than GAM and Grappa on eight nodes, respectively.

A detailed examination reveals that the performance difference comes from the *shared index table* in each DataFrame operation and the *shared chunks* between dependent DataFrame operations. In each operation, DataFrame constructs an index hash table to track the mapping from each destination chunk in the output column to all its source chunks in the input column. This index table is shared by all index-builder threads and worker threads. During processing, index-builder threads will concurrently insert into the index table using the destination chunk ID as the key and an array of source chunk IDs as the value, and worker threads will look up the shared index table and retrieve source chunks for processing. As a result, the massive writes and reads to the shared table can incur heavy coherence overhead. Further, DataFrame passes chunks as references between dependent operations and relies on the DSM system for actual data movement. However, it only performs lightweight computation over the fetched data (*i.e.*, low compute intensity as shown in Table 1), making the coherence overhead stand out.

DRust outperforms GAM and scales much better because of its light coherence protocol, which incurs negligible object move overhead for writes and no coherence overhead for reads. The use of affinity annotations also helps DataFrame colocate worker threads with their frequently accessed data, bringing 20% additional boost (details in §7.3). GAM, in contrast, has to invalidate each cache block upon each write and read, thereby bottlenecked by the extensive coherence traffic. Grappa performs the worst in all three DSM systems due to its *always-delegation* programming model, which implements every global memory read/write via a delegated function call. The cost for delegation overwhelms the actual memory access latency in this case, ruining the performance of the shared hash table. Grappa's delegation overhead actually causes a 1.23× slowdown when scaling DataFrame from a single node to two nodes.

SocialNet. Since SocialNet is microservice-based and can be deployed distributively, we added another baseline by running the original (non-DSM) code but deploying it on varying numbers of nodes. Figure 5b demonstrates the performance of all systems. SocialNet runs consistently faster with all three DSM systems compared to the original version. DRust, GAM, and Grappa achieve a 2.18×, 2.02×, and 1.57× speedup on a single node and a 3.51×, 1.33×, and 1.39× speedup on eight nodes, respectively. In the conventional setup, SocialNet requires data—such as text and media files—to be serialized into byte streams for network transmission, and then deserialized back into usable formats at the receiving end. This serialization and deserialization process is computationally intensive, particularly for large or complex data objects. In contrast, DSM systems enable SocialNet to pass references instead of the entire data values required by remote procedure calls. This approach eliminates the need for serialization and deserialization, reduces redundant data copies, and significantly enhances performance. DRust scales much better than GAM and Grappa thanks to its lightweight coherence protocol, achieving up to 2.77× and 3.16× higher throughput than GAM and Grappa, respectively.

GEMM. GEMM differs from the previous two applications in its high compute intensity and relatively infrequent shared memory accesses. In this application, matrices are transformed and divided into smaller sub-matrices for parallel processing. Each computing thread, responsible for multiplying sub-matrices, is assigned to a server. These threads cache their respective sub-matrices in the server's local memory and access them repeatedly to compute product results. This process is highly compute-intensive. As depicted in Figure 5c, DRust and GAM scale well for GEMM and achieve 5.93×, 3.82× speedup with eight nodes. In contrast, Grappa only achieves a 2.02× speedup with eight nodes due to its inability to cache sub-matrices locally, necessitating frequent remote accesses. DRust's superior performance over GAM, with a 1.55× higher speedup on eight nodes, is primarily due to its more efficient handling of initial cross-server data accesses required when a sub-matrix is first accessed remotely. Unlike GAM, which incurs significant runtime overhead due to the maintenance of state and location of data copies, DRust directly copies data to local memory, without any complex cross-server synchronization operations, thus enhancing overall efficiency.

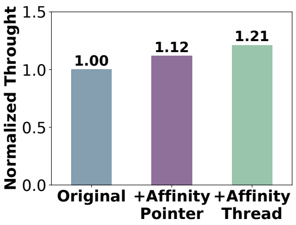


Figure 6: Effectiveness of DRust's affinity annotations.

Latency (cycles)	Average	Median	P90
`DRust`	395	356	536
`Rust`	364	332	496

Table 2: DRust's `Box` pointer only adds a small dereferencing cost compared to Rust's ordinary `Box`.

KV Store. KV Store is the most DSM-unfriendly application in our evaluation because it exposes poor memory locality and low compute intensity, which amplifies the overhead of cross-server memory accesses. In addition, it uses mutexes to synchronize between workers and the structure of the program does not lend itself to ownership-based read/write ordering.

Figure 5d shows the results. KV Store experiences a slowdown on all three DSM systems when scaling from a single node to two nodes (13% for DRust, 25% for GAM, and 93% for Grappa). However, the impact is mitigated when more servers are enlisted—DRust and GAM achieve 3.34× and 2.50× higher throughput on eight nodes compared to the original KV Store implementation, respectively. Due to the limited ownership semantics exposed by mutexes, DRust does not scale as well with KV Store as with other applications. DRust is 1.33× faster than GAM on eight nodes, benefiting from its adaptive load balancing and a more efficient implementation of mutexes utilizing one-sided RDMA atomic verbs, whereas GAM depends on less efficient two-sided RDMA messages for synchronization. Grappa, in contrast, incurs the highest distribution overhead and poorest scalability, primarily because each PUT/GET operation requires remote delegation, and nodes handling popular objects become bottlenecked due to skewed load.

7.3 Drill-Down Experiments

Affinity Annotations. In this experiment, we evaluated the individual contributions of affinity annotations by enabling each of them incrementally for DataFrame on eight nodes. Figure 6 reports the results. Using `TBox` helps DataFrame group chunks from the same column and eliminates the runtime dereference check overhead for single-column operations (*e.g.*, `filter`), bringing a 12% throughput improvement. Adding `spawn_to` further improves the throughput by 9% by

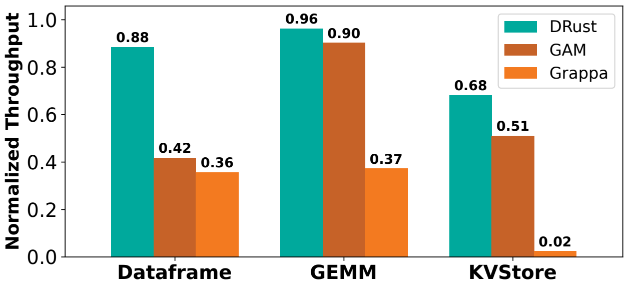


Figure 7: Comparison of cache coherence costs between DRust, GAM, and Grappa on eight nodes.

informing DRust runtime to colocate the worker thread to its input columns, which reduces cross-server memory accesses.

Runtime Dereference Checks. We measured the latency of dereferencing DRust's `Box` pointer and compared it with an ordinary Rust `Box` pointer. Both of them point to an 8-byte object in local memory and not in CPUs cache, which represents the common path for pointer dereferencing. Table 2 reports the results. DRust only adds a small overhead of ∼30 cycles. Note that this microbenchmark is extremely memory-intensive, whereas real-world applications usually employ larger object sizes and are more compute-intensive, further mitigating the runtime check overhead. For our evaluated applications, we observed a 1.02% overhead for DataFrame and a 1.14% overhead for BLAS, when they run with DRust on a single node, respectively.

Thread Migration Latency. To quantify how quickly DRust can resolve the workload imbalance, we measured the latency for the DRust runtime to migrate a thread by running GEMM on eight nodes and repeated the experiment for ten times. On average, DRust migrated 15 threads with an average of 218μs latency for each migration.

Cost of Cache Coherence. In this experiment, we ran each application again on a single node and eight nodes but fixed the total amount of CPU and memory resources. For the eight-node setting, we distributed the resources evenly to each node and measured application throughput. We expect to see a slowdown due to the cost of running the coherence protocol and cross-server memory accesses, but a good result for DRust should show that application performance remains close to its original single-node version. Figure 7 reports the results. SocialNet uses pass-by-value RPCs in its original version and is significantly slower than our DSM-based version, so it is omitted in the evaluation. DRust adds only moderate cache coherence cost with an overhead of 32% in the worst case (KV Store) and 4% in the best case (GEMM). GAM and Grappa, in contrast, incur much higher overheads ranging from 10% to 98% for different applications.

8 Related Work

Software DSM Systems. Distributed cache coherence protocols and their implementations for DSM have been extensively studied since 1980s [16–18, 31, 36, 49, 50, 56, 61–63, 80]. Among them, Munin [10] and TreadMarks [6] proposed relaxed consistency models and simpler protocols trying to alleviate the coherence overhead. Recent DSM systems leveraged today's advanced hardware such as RDMA [14, 45, 60, 77, 81, 92] to improve efficiency.

Disaggregated and Remote Memory. Memory disaggregation and remote memory techniques are another promising approach to scaling applications out of a single machine. Their key idea is to connect a host server with large memory pools [33, 40, 46] via fast datacenter network, which can be accessed by applications via OS kernel [4, 69, 76, 86] or software runtimes [34, 52, 73, 84, 85, 87]. However, they do not provide cache coherence.

Distributed Programming Abstractions. Researchers have studied and proposed new programming languages and abstractions. Munin [10] built a type system that defines types for local and global pointers and tracks whether the pointer is shared via type checking. X10 [20, 39] and UPC [30] introduce function offloading interfaces for distributed computing and additional type annotations to reduce the runtime overhead. Ray [89] and Nu [72] are two recent systems proposing new abstractions for distributed programming. Unlike DRust, they require effort to port applications to avoid fine-grained memory sharing.

Hardware-Accelerated DSM. Specialized datacenter network technologies and emerging hardware designs stand for another trend to accelerate DSM. Clio [35], StRoM [79], and RMC [5] reduce remote memory access latency by offloading tasks into customized hardware. Concordia [88], Kona [15], and CXL [23–26, 47, 48, 92] enable block-level or cache-line-level memory coherence with their hardware-implemented protocols. DRust can benefit from advances in hardware support and achieve better scalability.

9 Conclusion

This paper presents DRust, a practical DSM system based on the ownership model. It automatically turns a single-machine Rust program into its distributed version with a lightweight coherence protocol guided by language semantics. DRust significantly outperforms existing state-of-the-art DSM systems, demonstrating that a language-guided DSM can achieve strong memory consistency, transparency, and efficiency simultaneously.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Daniel S. Berger for his feedback. This work is supported by CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254, CNS-1764077, CNS-1956322, CNS-2106404. This work is also supported by Alibaba Group through Alibaba Research Intern Program, and funding from Amazon and Samsung.

A Artifact Appendix

A.1 Artifact Summary

DRust is an efficient, consistent, and user-friendly DSM system featuring a lightweight coherence protocol guided by language semantics. DRust allows for seamless scaling of single-machine applications to multi-server environments without sacrificing performance. Demonstrating significant improvements over existing DSM systems, DRust combines strong memory consistency, transparency, and efficiency effectively.

A.2 Artifact Check-list

- **Hardware:** Intel servers equipped with InfiniBand

- **Software Environment:** Rust 1.69.0, GCC 5.5, Linux Kernel 5.4, Ubuntu 18.04, MLNX-OFED 4.9

- **Public Link to Repository:** https://github.com/uclasystem/DRust

- **Code License:** GNU General Public License (GPL)

A.3 Description

A.3.1 DRust's Codebase

DRust comprises four main components:

- An RDMA communication library written in C
- The DRust library
- Applications integrated with DRust
- Necessary shell scripts and configuration files

A.3.2 Deploying DRust

The initial step in deploying DRust involves cloning the source code on all involved servers:

```
git clone git@github.com:uclasystem/DRust.git
```

Adjust several configurations according to your server setup and operational requirements:

1. Set the Number of Servers:

    - Define TOTAL_NUM_SERVERS in comm-lib/rdma-common.h based on the total number of available servers.

    - Similarly, adjust NUM_SERVERS in drust/src/conf.rs.

2. Configure the Distributed Heap Size by setting UNIT_HEAP_SIZE_GB in drust/src/conf.rs to the required heap size per server, e.g., 16 for 16GB.

3. Update the InfiniBand IP addresses and ports in comm-lib/rdma-server-lib.c:

    ```
    const char *ip_str[2] = {"10.0.0.1",
    "10.0.0.2"};
    const char *port_str[2] = {"9400", "9401"};
    ```

4. In drust.json, update each server's IP address and specify three available ports.

Following configuration, build DRust as follows:

```
# Compile the communication static library
cd comm-lib
make -j lib
# Copy the static library to the DRust directory
cp libmyrdma.a ../drust/
# Compile the Rust components
cd ../drust
cargo build --release
```

Deploy the compiled binary across all servers post-build, ensuring its correct distribution:

```
scp target/release/drust user@ip:DRust/drust.out
```

A.3.3 Running Applications

DRust is bundled with four example applications: Dataframe, GEMM, KVStore, and SocialNet. Follow these steps to execute them:

1. Launch the DRust executable on all servers, excluding the main server:

    ```
    # Start the DRust process with the specified
    server index and application name.
    # For example, ./../drust.out -s 7 -a gemm
    cd drust
    ./../drust.out -s server_id -a app_name
    ```

2. On the main server:

    ```
    # Start the main DRust process with the
    specified application.
    cd drust
    ./../drust.out -s 0 -a app_name
    ```

More details of DRust's installation and deployment can be found in DRust's code repository.

References

[1] The Z garbage collector. https://wiki.openjdk.java.net/display/zgc/Main.

[2] Lapack benchmark. https://www.netlib.org/lapack/lug/node71.html, 2023.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[4] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[5] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[7] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST*, 2014.

[8] H. G. Baker. Lively linear lisp: look ma, no garbage!. *SIGPLAN Not.*, 27(8):8998, aug 1992.

[9] T. Balabonski, F. Pottier, and J. Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4), aug 2016.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[11] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[12] M. N. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *ISCA*, pages 13–24, 2012.

[13] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, Nov. 2020.

[14] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[15] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, pages 79–92. Association for Computing Machinery, New York, NY, USA, 2021.

[16] R. Campbell, G. Johnston, and V. Russo. Choices (class hierarchical open interface for custom embedded systems). *ACM SIGOPS Operating Systems Review*, 21(3):9–17, 1987.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. *ACM SIGOPS Operating Systems Review*, 25(5):152–164, 1991.

[18] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, 1995.

[19] CCIX. Cache coherent interconnect for accelerators. https://www.ccixconsortium.com/, 2018.

[20] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, 2008.

[21] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, 2005.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[23] Compute express link 3.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf, 2022.

[24] Compute express link 1.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.0-Specification.pdf, 2019.

[25] Compute express link 1.1. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.1-Specification.pdf, 2019.

[26] Compute express link 2.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf, 2020.

[27] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 5969, New York, NY, USA, 2001. Association for Computing Machinery.

[28] Managing a Cluster of Docker Daemons using Swarm Mode. https://docs.docker.com/engine/swarm/, 2023.

[29] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[30] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: distributed shared memory programming*. John Wiley & Sons, 2005.

[31] B. D. Fleisch. Distributed shared memory in a loosely coupled distributed system. *ACM SIGCOMM Computer Communication Review*, 17(5):317–327, 1987.

[32] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 318, New York, NY, USA, 2019. Association for Computing Machinery.

[33] GenZ. Genz consortium. http://genzconsortium.org/, 2019.

[34] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692708, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[36] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE micro*, 12(1):10–22, 1992.

[37] Database-like ops benchmark. https://github.com/h2oai/db-benchmark, 2023.

[38] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[39] R. Haque and J. Palsberg. Type inference for place-oblivious objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[41] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):3749, apr 2007.

[42] Intel. Intel high performance computing fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/, 2019.

[43] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[44] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[45] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2015.

[46] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[47] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574587, New York, NY, USA, 2023. Association for Computing Machinery.

[48] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[49] K. Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[50] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[51] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[52] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107, 2022.

[53] H. Ma, Y. Qiao, S. Liu, S. Yu, Y. Ni, Q. Lu, J. Wu, Y. Zhang, M. Kim, and H. Xu. Drust: Language-guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. *arXiv preprint arXiv:2406.02803*, 2024.

[54] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

[55] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[56] R. G. Minnich and D. J. Farber. The mether system: Distributed shared memory for sunos 4.0. *Technical Reports (CIS)*, page 332, 1993.

[57] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd edition, 2020.

[58] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229240, sep 2008.

[59] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, Nov. 2020.

[60] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[61] J. Nieplocha, R. Harrison, M. Krishnan, B. Palmer, and V. Tipparaju. Combining shared and distributed memory models: Evolution and recent advancements of the global array toolkit. In *proceedings of POHLL'2002 workshop of ICS-2002, NYC*, 2002.

[62] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable" shared-memory" programming model for distributed memory computers. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE, 1994.

[63] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[64] OpenCAPI. Open coherent accelerator processor interface. https://opencapi.org/, 2018.

[65] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[67] Polars: Blazingly Fast DataFrame Library. https://pola-rs.github.io/polars/, 2023.

[68] Y. Qiao, Z. Ruan, H. Ma, A. Belay, M. Kim, and H. Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[69] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, 2023.

[70] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020.

[71] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[72] Z. Ruan, S. J. Park, M. K. Aguilera, A. Belay, and M. Schwarzkopf. Nu: Achieving {Microsecond-Scale} resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, 2023.

[73] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[74] S. M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[75] Rust. https://www.rust-lang.org/, 2023.

[76] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[77] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[78] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[79] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[80] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, 1997.

[81] K. Taranov, S. Di Girolamo, and T. Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1811–1824, 2021.

[82] Tokio Team. Build reliable network applications without compromising speed. https://tokio.rs/.

[83] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, 2015.

[84] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, pages 261–280, 2020.

[85] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, pages 35–53, 2022.

[86] C. Wang, Y. Qiao, H. Ma, S. Liu, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for {Multi-Applications} on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, 2023.

[87] C. Wang, Y. Shan, P. Zuo, and H. Cui. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology*, 38(5):949–969, 2023.

[88] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[89] S. Wang, E. Liang, E. Oakes, B. Hindman, F. S. Luan, A. Cheng, and I. Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, Apr. 2021.

[90] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[91] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, page 10, Berkeley, CA, USA, 2010.

[92] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Taming Throughput-Latency Tradeoff in LLM Inference with *Sarathi-Serve*

Amey Agrawal*[2], Nitin Kedia[1], Ashish Panwar[1], Jayashree Mohan[1], Nipun Kwatra[1],
Bhargav S. Gulavani[1], Alexey Tumanov[2], and Ramachandran Ramjee[1]

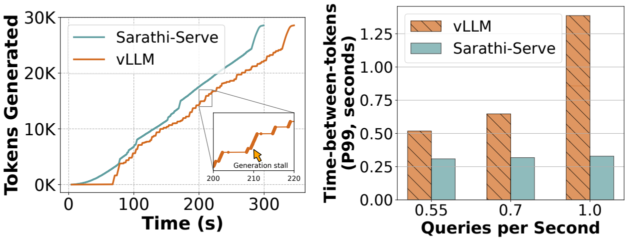
[1]Microsoft Research India
[2]Georgia Institute of Technology

Abstract

Each LLM serving request goes through two phases. The first is *prefill* which processes the entire input prompt and produces the first output token and the second is *decode* which generates the rest of output tokens, one-at-a-time. Prefill iterations have high latency but saturate GPU compute due to parallel processing of the input prompt. In contrast, decode iterations have low latency but also low compute utilization because a decode iteration processes only a single token per request. This makes batching highly effective for decodes and consequently for overall throughput. However, batching multiple requests leads to an interleaving of prefill and decode iterations which makes it challenging to achieve both high throughput and low latency.

We introduce an efficient LLM inference scheduler, Sarathi-Serve, to address this throughput-latency tradeoff. Sarathi-Serve introduces *chunked-prefills* which splits a prefill request into near equal sized chunks and creates *stall-free* schedules that adds new requests in a batch without pausing ongoing decodes. Stall-free scheduling unlocks the opportunity to improve throughput with large batch sizes while minimizing the effect of batching on latency. Furthermore, uniform batches in Sarathi-Serve ameliorate the imbalance between iterations, resulting in minimal pipeline bubbles.

Our techniques yield significant improvements in inference performance across models and hardware under tail latency constraints. For Mistral-7B on single A100 GPUs, we achieve 2.6× higher serving capacity and up to 3.7× higher serving capacity for the Yi-34B model on two A100 GPUs as compared to vLLM. When used with pipeline parallelism on Falcon-180B, Sarathi-Serve provides up to 5.6× gain in the end-to-end serving capacity. The source code for Sarathi-Serve is available at https://github.com/microsoft/sarathi-serve.



(a) Generation stall.

(b) High tail latency.

Figure 1: Yi-34B running on two A100 GPUs serving 128 requests from *arxiv-summarisation* trace. 1a highlights one of the many generation stalls lasting over several seconds in vLLM [53]. 1b shows the impact of increasing load on tail latency. Sarathi-Serve improves throughput while eliminating generation stalls.

1 Introduction

Large language models (LLMs) [34,35,52,57,71] have shown impressive abilities in a wide variety of tasks spanning natural language processing, question answering, code generation, etc. This has led to tremendous increase in their usage across many applications such as chatbots [2, 5, 6, 57], search [4, 9, 11, 19, 25], code assistants [1, 8, 20], etc. The significant GPU compute required for running inference on large models, coupled with significant increase in their usage, has made LLM inference a dominant GPU workload today. Thus, optimizing LLM inference has been a key focus for many recent systems [29, 53, 58, 59, 63, 75, 77].

Optimizing throughput and latency are both important objectives in LLM inference since the former helps keep serving costs tractable while the latter is necessary to meet application requirements. In this paper, we show that current LLM serving systems have to face a tradeoff between throughput and latency. In particular, LLM inference throughput can be increased significantly with batching. However, the way existing systems batch multiple requests leads to a compromise on either throughput or latency. For example, Figure 1b shows

*Part of this work was done during an internship at MSR India.

that increasing load can significantly increase tail latency in a state-of-the-art LLM serving system vLLM [53].

Each LLM inference request goes through two phases – a *prefill* phase followed by a *decode* phase. The *prefill* phase corresponds to the processing of the input prompt and the *decode* phase corresponds to the autoregressive token generation. The prefill phase is compute-bound because it processes all tokens of an input prompt in parallel whereas the decode phase is memory-bound because it processes only one token per-request at a time. Therefore, decodes benefit significantly from batching because larger batches can use GPUs more efficiently whereas prefills do not benefit from batching.

Current LLM inference schedulers can be broadly classified into two categories[1], namely, *prefill-prioritizing* and *decode-prioritizing* depending on how they schedule the prefill and decode phases while batching requests. In this paper, we argue that both strategies have fundamental pitfalls that make them unsuitable for serving online inference (see Figure 2).

Traditional request-level batching systems such as FasterTransformer [7] employ *decode-prioritizing* scheduling. These systems submit a batch of requests to the execution engine that first computes the prefill phase of all requests and then schedules their decode phase. The batch completes only after all requests in it have finished their decode phase i.e., new prefills are not scheduled as long as one or more requests are doing decodes. This strategy optimizes inference for latency metric time-between-tokens or TBT – an important performance metric for LLMs. This is because new requests do not affect the execution of ongoing requests in their decode phase. However, *decode-prioritizing* schedulers severely compromise on throughput because even if some requests in a batch finish early, the execution continues with reduced batch size until the completion of the last request.

Orca [75] introduced iteration-level batching wherein requests can dynamically enter or exit a batch at the granularity of individual iterations. Iteration-level batching improves throughput by avoiding inefficiencies of request-level batching systems. Orca and several other recent systems like vLLM [23] combine iteration-level batching with *prefill-prioritizing* scheduling wherein they eagerly schedule the prefill phase of one or more requests first i.e., whenever GPU memory becomes available. This way, *prefill-prioritizing* schedulers have better throughput because computing prefills first allows subsequent decodes to operate at high batch sizes. However, prioritizing prefills leads to high latency because it interferes with ongoing decodes. Since prefills can take arbitrarily long time depending on the lengths of the given prompts, *prefill-prioritizing* schedulers lead to an undesirable phenomenon that we refer to as *generation stalls* in this paper. For example, Figure 1a shows that a generation stall in vLLM can last over several seconds.

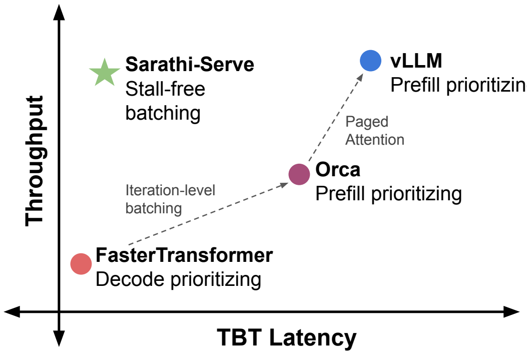


Figure 2: Current LLM serving systems involve a tradeoff between throughput and latency depending on their scheduling policy. Prioritizing prefills optimizes throughput but sacrifices TBT (time-between-tokens) tail latency whereas prioritizing decodes has the opposite effect. Sarathi-Serve serves high throughput with low TBT latency via stall-free batching. (The figure is illustrative and actual values will depend on the model and workload characteristics.)

Another challenge introduced by traditional iteration-level scheduling systems like Orca [75] is pipeline stalls or bubbles [49]. These appear in pipeline-parallelism (PP) deployments that are needed to scale LLM inference across several nodes. In servers with high bandwidth connectivity such as NVIDIA DGX A100 [16], tensor-parallelism (TP) [64] can enable deployment of an LLM on up to 8 GPUs, supporting large batch sizes with low latencies. However, TP can have prohibitively high latencies when hyper-clusters are unavailable [33]. Thus, as an alternative to TP, pipeline-parallelism (PP) [33, 55] is typically used across commodity networks. Existing systems rely on micro-batches to mitigate pipeline stalls or bubbles [49]. However, the standard micro-batch based scheduling can still lead to pipeline bubbles due to the unique characteristics of LLM inference. Specifically, LLM inference consists of a mixture of varying length prefills and decodes. The resulting schedule can thus have wildly varying runtimes across different micro-batches that waste GPU cycles and degrade the overall system throughput.

To address these challenges, we propose Sarathi-Serve, a scheduler to balance the throughput-latency tradeoff for scalable online LLM inference serving. Sarathi-Serve is based on two key ideas: *chunked-prefills* and *stall-free* scheduling. *Chunked-prefills* splits a prefill request into equal compute-sized chunks and computes a prompt's prefill phase over multiple iterations (each with a subset of the prompt tokens). *Stall-free* scheduling allows *new requests to join a running batch without pausing ongoing decodes*. This involves constructing a batch by coalescing all the on-going decodes with one (or more) prefill chunks from new requests such that each batch reaches the pre-configured chunk size. Sarathi-Serve builds upon iteration-level batching but with an important distinction: it throttles the number of prefill tokens in each it-

[1] We classify recent schedulers Splitwise [58] and DistServe [77] under a third category "disaggregated" and discuss them in §6.

eration while admitting new requests in a running batch. This not only bounds the latency of each iteration, but also makes it nearly independent of the total length of input prompts. This way, Sarathi-Serve minimizes the effect of computing new prefills on the TBT of ongoing decodes enabling both high throughput and low TBT latency.

In addition, hybrid batches (consisting of prefill and decode tokens) constructed by Sarathi-Serve have a near-uniform compute requirement. With pipeline-parallelism, this allows us to create balanced micro-batching based schedules that significantly reduce pipeline bubbles and improve GPU utilization, thus allowing efficient and scalable deployments.

We evaluate Sarathi-Serve across different models and hardware — Mistral-7B on a single A100, Yi-34B on 2 A100 GPUs with 2-way tensor parallelism, LLaMA2-70B on 8 A40 GPUs, and Falcon-180B with 2-way pipeline and 4-way tensor parallelism across 8 A100 GPUs connected over commodity ethernet. For Yi-34B, Sarathi-Serve improves system serving capacity by up to 3.7× under different SLO targets. Similarly for Mistral-7B, we achieve up to 2.6× higher serving capacity. Sarathi-Serve also reduces pipeline bubbles, resulting in up to 5.6× gains in end-to-end serving capacity for Falcon-180B deployed with pipeline parallelism.

The main contributions of our paper include:

1. We identify a number of pitfalls in the current LLM serving systems, particularly in the context of navigating the throughput-latency tradeoff.
2. We introduce two simple-yet-effective techniques, *chunked-prefills* and *stall-free batching*, to improve the performance of an LLM serving system.
3. We show generality through extensive evaluation over multiple models, hardware, and parallelism strategies demonstrating that Sarathi-Serve improves model serving capacity by up to an order of magnitude.

2 Background

In this section, we describe the typical LLM model architecture along with their auto-regressive inference process. We also provide an overview of the scheduling policies and important performance metrics.

2.1 The Transformer Architecture

Popular large language models, like, GPT-3 [18], LLaMA [66], Yi [24] etc. are decoder-only transformer models trained on next token prediction tasks. These models consist of a stack of layers identical in structure. Each layer contains two modules – self-attention and feed-forward network (FFN).

Self-attention module: The self-attention module is central to the transformer architecture [67], enabling each part of a sequence to consider all previous parts for generating a contextual representation. During the computation of self-attention, first the Query (Q), Key (K) and Value (V) vectors corresponding to each input token are obtained via a linear transformation. Next, the *attention* operator computes a semantic relationship among all tokens of a sequence. This involves computing a dot-product of each Q vector with K vectors of all preceding tokens of the sequence, followed by a softmax operation to obtain a weight vector, which is then used to compute a weighted average of the V vectors. This attention computation can be performed across multiple *heads*, whose outputs are combined using a linear transformation.

Feed-forward network (FFN): FFN typically consists of two linear transformations with a non-linear activation in between. The first linear layer transforms an input token embedding of dimension h to a higher dimension $h2$. This is followed by an activation function, typically ReLU or GELU [27,46]. Finally, the second linear layer, transforms the token embedding back to the original dimension h.

2.2 LLM Inference Process

Autoregressive decoding: LLM inference consists of two distinct phases – a *prefill* phase followed by a *decode* phase. The prefill phase processes the user's input prompt and produces the first output token. Subsequently, the decode phase generates output tokens one at a time wherein the token generated in the previous step is passed through the model to generate the next token until a special *end-of-sequence* token is generated. Note that the decode phase requires access to all the keys and values associated with all the previously processed tokens to perform the attention operation. To avoid repeated recomputation, contemporary LLM inference systems store activations in KV-cache [7,64,75].

A typical LLM prompt contains 100s-1000s of input tokens Table 2, [76]. During the prefill phase all these prompt tokens are processed in parallel in a single iteration. The parallel processing allows efficient utilization of GPU compute. On the contrary, the decode phase involves a full forward pass of the model over a single token generated in the previous iteration. This leads to low compute utilization making decodes memory-bound.

Batched LLM inference in multi-tenant environment: A production serving system must deal with concurrent requests from multiple users. Naively processing requests in a sequential manner leads to a severe under-utilization of GPU compute. In order to achieve higher GPU utilization, LLM serving systems leverage batching to process multiple requests concurrently. This is particularly effective for the decode phase processing which has lower computational intensity at low batch sizes. Higher batch sizes allows the cost of fetching model parameters to be amortized across multiple requests.

Recently, several complementary techniques have been proposed to optimize throughput by enabling support for larger batch sizes. Kwon et al. propose PagedAttention [53], which allows more requests to concurrently execute, eliminating fragmentation in *KV-cache*. The use of Multi Query Attention

Algorithm 1 Request-level batching. New requests are admitted only if there are no decodes left (line 3). This optimizes TBT but wastes GPU compute in many decode-only iterations (line 10) with potentially small batch sizes.

```
1:  Initialize current batch $B \leftarrow \emptyset$
2:  while True do
3:      if $B = \emptyset$ then
4:          $R_{new} \leftarrow$ get_next_request()
5:          while can_allocate_request($R_{new}$) do
6:              $B \leftarrow B + R_{new}$
7:              $R_{new} \leftarrow$ get_next_request()
8:          prefill($B$)
9:      else
10:         decode($B$)
11:         $B \leftarrow$ filter_finished_requests($B$)
```

Algorithm 2 Iteration-level batching (vLLM). Prefills are executed eagerly (lines 8-9), potentially introducing a generation stall for ongoing decodes (line 12).

```
1:  Initialize current batch $B \leftarrow \emptyset$
2:  while True do
3:      $B_{new} \leftarrow \emptyset$
4:      $R_{new} \leftarrow$ get_next_request()
5:      while can_allocate_request($R_{new}$) do
6:          $B_{new} \leftarrow B_{new} + R_{new}$
7:          $R_{new} \leftarrow$ get_next_request()
8:      if $B_{new} \neq \emptyset$ then
9:          prefill($B_{new}$)
10:         $B \leftarrow B + B_{new}$
11:     else
12:         decode($B$)
13:     $B \leftarrow$ filter_finished_requests($B$)
```

(MQA) [61], Group Query Attention (GQA) [30] in leading edge LLM models like LLaMA2 [66], Falcon [31] and Yi [24] has also significantly helped in alleviating memory bottleneck in LLM inference. For instance, LLaMA2-70B model has a 8× smaller KV-cache footprint compared to LLaMA-65B.

2.3 Multi-GPU LLM Inference

With ever-increasing growth in model sizes, it becomes necessary to scale LLMs to multi-GPU or even multi-node deployments [22, 59]. Furthermore, LLM inference throughput, specifically that of the decode phase is limited by the maximum batch size we can fit on a GPU. Inference efficiency can therefore benefit from model-parallelism which allows larger batch sizes by sharding model weights across multiple GPUs. Prior work has employed both tensor-parallelism (TP) [64] and pipeline-parallelism (PP) [7, 72, 75] for this purpose.

TP shards each layer across the participating GPUs by splitting the model weights and KV-cache equally across GPU workers. This way, TP can linearly scale per-GPU batch size. However, TP involves a high communication cost due to two all-reduce operations per layer – one in attention computation and the other in FFN [64]. Moreover, since these communication operations are in the critical path, TP is preferred only within a single node where GPUs are connected via high bandwidth interconnects like NVLink.

Compared to TP, PP splits a model layer-wise, where each GPU is responsible for a subset of layers. To keep all GPUs in the 'pipeline' busy, *micro-batching* is employed. These micro-batches move along the pipeline from one stage to the next at each iteration. PP has much better compute-communication ratio compared to TP, as it only needs to send activations once for multiple layers of compute. Furthermore, PP requires communication only via point-to-point communication operations, compared to the more expensive allreduces in TP. Thus, PP is more efficient than TP when high-bandwidth interconnects are unavailable *e.g.,* in cross-node deployments.

2.4 Performance Metrics

There are two primary latency metrics of interest for LLM serving: TTFT (time-to-first-token) and TBT (time-between-tokens). For a given request, TTFT measures the latency of generating the first output token from the moment a request arrives in the system. This metric reflects the initial responsiveness of the model. TBT on the other hand measures the interval between the generation of consecutive output tokens of a request, and affects the overall perceived fluidity of the response. When system is under load, low throughput can lead to large scheduling delays and consequently higher TTFT.

In addition, we use a throughput metric, *Capacity*, defined as the maximum request load (queries-per-second) a system can sustain while meeting certain latency targets. Higher capacity is desirable because it reduces the cost of serving.

2.5 Scheduling Policies for LLM Inference

The scheduler is responsible for admission control and batching policy. For the ease of exposition, we investigate existing LLM inference schedulers by broadly classifying them under two categories – *prefill-prioritizing* and *decode-prioritizing*.

Conventional inference engines like FasterTransformer [7], Triton Inference Server [17] use *decode-prioritizing* schedules with request-level batching *i.e.,* they pick a batch of requests and execute it until *all* requests in the batch complete (Algorithm 1). This approach reduces the operational complexity of the scheduling framework but at the expense of inefficient resource utilization. Different requests in a batch typically have a large variation in the number of input and output tokens. Request-level schedulers pad shorter requests with zeros to match their length with the longest request in the batch which results in wasteful compute and longer wait times for pending requests [75].

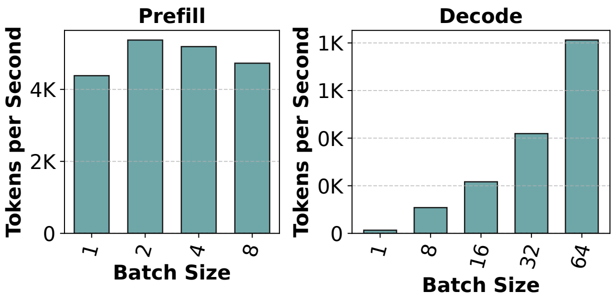


Figure 3: Throughput of the prefill and decode phases with different batch sizes for Mistral-7B running on a single A100 GPU. We use prompt length of 1024 for both prefill and decode experiments. Note that different y-axis, showing prefills are much more efficient than decode. Further, note that *batching boosts decode throughput almost linearly but has a marginal effect on prefill throughput.*

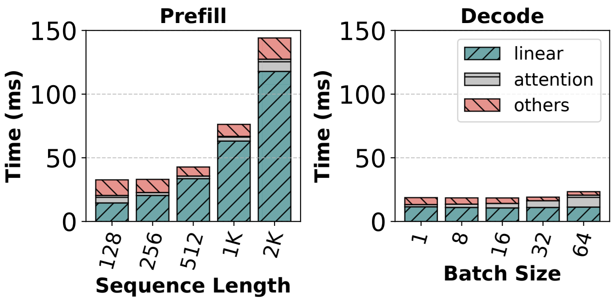


Figure 4: Prefill and decode time with different input sizes for Mistral-7B running on single A100 GPU. Linear layers contribute to the majority of runtime in both prefill and decode phases. Due to the low arithmetic intensity in decode batches, the cost of linear operation for 1 decode token is nearly same as 128 prefill tokens.

To avoid wasted compute of request-level batching, Orca [75] introduced a fine-grained iteration-level batching mechanism where requests can dynamically enter and exit a batch after each model iteration.(Algorithm 2). This approach can significantly increase system throughput and is being used in many LLM inference serving systems today *e.g.,* vLLM [23], TensorRT-LLM [21], and LightLLM [12].

Current iteration-level batching systems such as vLLM [23] and Orca [75] use *prefill-prioritizing* schedules that eagerly admit new requests in a running batch at the first available opportunity, e.g., whenever GPU memory becomes available. Prioritizing prefills can improve throughput because it increases the batch size of subsequent decode iterations.

3 Motivation

In this section, we first analyse the cost of prefill and decode operations. We then highlight the throughput-latency trade-off and pipeline bubbles that appear in serving LLMs.

3.1 Cost Analysis of Prefill and Decode

As discussed in §2.2, while the *prefill* phase processes all input tokens in parallel and effectively saturates GPU compute, the *decode* phase processes only a single token at a time and is very inefficient. Figure 3 illustrates throughput as a function of batch size, and we can observe that while for decode iterations throughput increases roughly linearly with batch size, prefill throughput almost saturates even with a single request.

Takeaway-1: *The two phases of LLM inference – prefill and decode – demonstrate contrasting behaviors wherein batching boosts decode phase throughput immensely but has little effect on prefill throughput.*

Figure 4 breaks down the prefill and decode compute times into linear, attention and others, and shows their individual contributions. From the figure, we see that linear operators contribute to the majority of the runtime cost. While attention cost grows quadratically with sequence length, linear operators still contribute more than 80% to the total time even at high sequence lengths. Therefore, optimizing linear operators is important for improving LLM inference.

Low Compute Utilization during Decodes: Low compute utilization during the decode phase is a waste of GPU's processing capacity. To understand this further, we analyze the arithmetic intensity of prefill and decode iterations. Since the majority of the time in LLM inference is spent in linear operators, we focus our analysis on them.

Matrix multiplication kernels overlap memory accesses along with computation of math operations. The total execution time of an operation can be approximated to $T = \max(T_{\text{math}}, T_{\text{mem}})$, where T_{math} and T_{mem} represent the time spent on math and memory fetch operations respectively. An operation is considered memory-bound if $T_{\text{math}} < T_{\text{mem}}$. Memory-bound operations have low Model FLOPs Utilization (MFU) [35]. On the other hand, compute-bound operations have low Model Bandwidth Utilization (MBU). When $T_{\text{math}} = T_{\text{mem}}$, both compute and memory bandwidth utilization are maximized. Arithmetic intensity quantifies the number of math operations performed per byte of data fetched from the memory. At the optimal point, the arithmetic intensity of operation matches the FLOPS-to-Bandwidth ratio of the device. Figure 5 shows arithmetic intensity as a function of the number of tokens in the batch for linear layers in LLaMA2-70B running on four A100 GPUs. Prefill batches amortize the cost of fetching weights of the linear operators from HBM memory to GPU cache over a large number of tokens, allowing it to have high arithmetic intensity. In contrast, decode batches have very low computation intensity. Figure 6 shows the total execution time of linear operators in

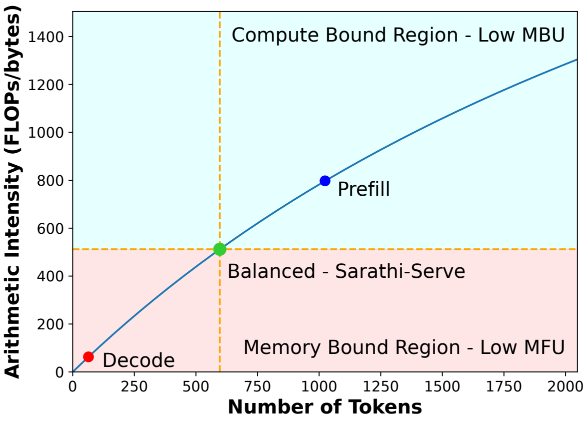


Figure 5: Arithmetic intensity trend for LLaMA2-70B linear operations with different number of token running on four A100s. Decode batches have low arithmetic intensity *i.e.,* they are bottlenecked by memory fetch time, leading to low compute utilization. Prefill batches are compute bound with sub-optimal bandwidth utilization. Sarathi-Serve forms balanced batches by combining decodes and prefill chunks to maximize both compute and bandwidth utilization.

an iteration for LLaMA2-70B as a function of the number of tokens. Note that execution time increases only marginally in the beginning *i.e.,* as long as the batch is in a memory-bound regime, but linearly afterwards *i.e.,* when the batch becomes compute-bound.[2]

Takeaway-2: *Decode batches operate in memory-bound regime leaving compute underutilized. This implies that more tokens can be processed along with a decode batch without significantly increasing its latency.*

3.2 Throughput-Latency Trade-off

Iteration-level batching improves system throughput but we show that it comes at the cost of high TBT latency due to a phenomenon we call *generation stalls*.

Figure 7 compares different scheduling policies. The example shows a timeline (left to right) of requests A, B, C and D. Requests A and B are in decode phase at the start of the interval and after one iteration, requests C and D enter the system. Orca and vLLM both use FCFS iteration-level batching with eager admission of prefill requests but differ in their batch composition policy. Orca supports hybrid batches composed of both prefill and decode requests whereas vLLM only supports batches that contain either all prefill or all decode requests. Irrespective of this difference, both Orca and vLLM can improve throughput by maximizing the batch size

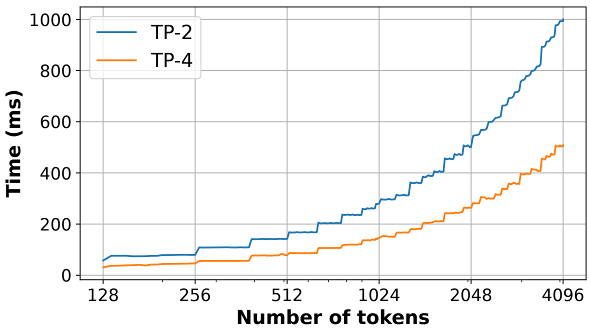


Figure 6: Linear layer execution time as function of number of tokens in a batch for LLaMA2-70B on A100(s) with different tensor parallel degrees. When the number of tokens is small, execution time is dictated by the cost of fetching weights from HBM memory. Hence, execution time is largely stagnant in the 128-512 tokens range, especially for higher tensor parallel degrees. Once the number of tokens in the batch cross a critical threshold, the operation become compute bound and the runtime increases linearly with number of tokens.

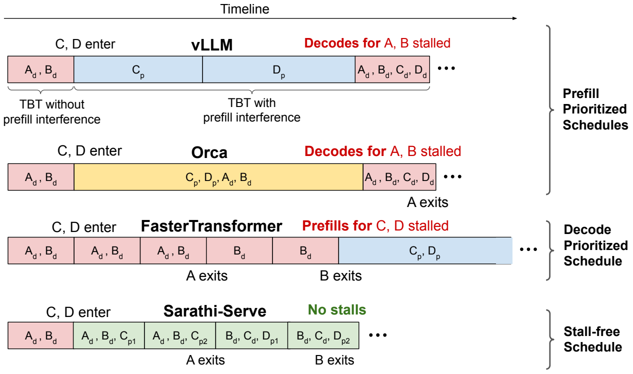


Figure 7: A generation stall occurs when one or more prefills are scheduled in between consecutive decode iterations of a request. A, B, C and D represent different requests. Subscript d represents a decode iteration, p represents a full prefill and $p0$, $p1$ represent two chunked prefills of a given prompt. vLLM induces generation stalls by scheduling as many prefills as possible before resuming ongoing decodes. Despite supporting hybrid batches, Orca cannot mitigate generation stalls because the execution time of batches containing long prompts remains high. FasterTransformer is free of generation stalls as it finishes all ongoing decodes before scheduling a new prefill but compromises on throughput due to low decode batch size. In contrast, Sarathi-Serve generates a schedule that eliminates generation stalls yet delivers high throughput.

in subsequent decode iterations. However, eagerly scheduling prefills of requests C and D delays the decodes of already running requests A and B because an iteration that computes one or more prefills can take several seconds depending on the

[2] Theoretically, we expect the operators to become compute-bound at ∼200 tokens on A100 GPUs, however, in practice we observe that it happens at ∼500-600 tokens for higher tensor parallel dimensions due to fixed overheads.

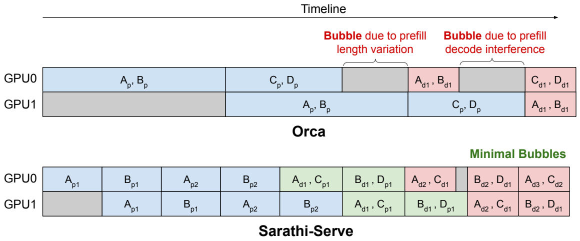


Figure 8: A 2-way pipeline parallel iteration-level schedule in Orca across 4 requests (A,B,C,D) shows the existence of pipeline bubbles due to non-uniform batch execution times. Sarathi-Serve is able to minimize these stalls by creating uniform-compute batches.

lengths of input prompts. Therefore, *prefill-prioritizing* schedulers can introduce *generation stalls* for ongoing decodes resulting in latency spikes caused by high TBT.

In contrast to iteration-level batching, request-level batching systems such as FasterTransformer [7] do not schedule new requests until *all* the already running requests complete their decode phase (line 3 in Algorithm 1). In Figure 7, the prefills for requests C and D get stalled until requests A and B both exit the system. Therefore, *decode-prioritizing* systems provide low TBT latency albeit at the cost of low system throughput. For example, Kwon et al. [53] show that iteration-level batching with PagedAttention can achieve an order of magnitude higher throughput compared to FasterTransformer.

One way to reduce latency spikes in iteration-level batching systems is to use smaller batch sizes as recommended in Orca [75]. However, lowering batch size adversely impacts throughput as shown in §2.2. Therefore, existing systems are forced to trade-off between throughput and latency depending on the desired SLOs.

Takeaway-3: *The interleaving of prefills and decodes involves a trade-off between throughput and latency for current LLM inference schedulers. State-of-the-art systems today use prefill-prioritizing schedules that trade TBT latency for high throughput.*

3.3 Pipeline Bubbles waste GPU Cycles

Pipeline-parallelism (PP) is a popular strategy for cross-node deployment of large models, owing to its lower communication overheads compared to Tensor Parallelism (TP). A challenge with PP, however, is that it introduces *pipeline bubbles* or periods of GPU inactivity as subsequent pipeline stages have to wait for the completion of the corresponding micro-batch in the prior stages. Pipeline bubbles is a known problem in training jobs, where they arise between the forward and backward passes due to prior stages needing to wait for the backward pass to arrive. Micro-batching is a common technique used in PP training jobs to mitigate pipeline bubbles [33, 49, 55].

Inference jobs only require forward computation and therefore one might expect that micro-batching can eliminate pipeline bubbles during inference. In fact, prior work on transformer inference, such as, FasterTransformer [7] and FastServe [72] use micro-batches but do not mention pipeline-bubbles. Recently proposed Orca [75] also suggests that iteration-level scheduling eliminates bubbles in pipeline scheduling (see Figure 8 in [75]). However, our experiments show that even with iteration-level scheduling, pipeline bubbles can waste significant GPU cycles with PP (§5.3).

Each micro-batch (or iteration) in LLM inference can require a different amount of compute (and consequently has varying execution time), depending on the composition of prefill and decode tokens in the micro-batch (see Figure 8). We identify three types of bubbles during inference: (1) bubbles like PB_1 that occur due to the varying number of prefill tokens in two consecutive micro-batches (2) bubbles like PB_2 that occur due to different compute times of prefill and decode stages when one is followed by the other, and (3) bubbles like PB_3 that occur due to difference in decode compute times between micro-batches since the attention cost depends on the accumulated context length (size of the KV-cache) and varies across requests. For Falcon-180B, a single prompt of 4k tokens takes $\approx$ 1150 ms to execute compared to a decode only iteration with batch size 32 which would take about $\approx$ 200 ms to execute. Interleaving of these iteration could result in a bubble of $\approx$ 950 ms. These pipeline bubbles are wasted GPU cycles and directly correspond to a loss in serving throughput and increased latency. This problem is aggravated with increase in prompt lengths and batch size, due to longer and more frequent prefill iterations respectively. If we can ensure that each micro-batch performs uniform computation, we can mitigate these pipeline bubbles.

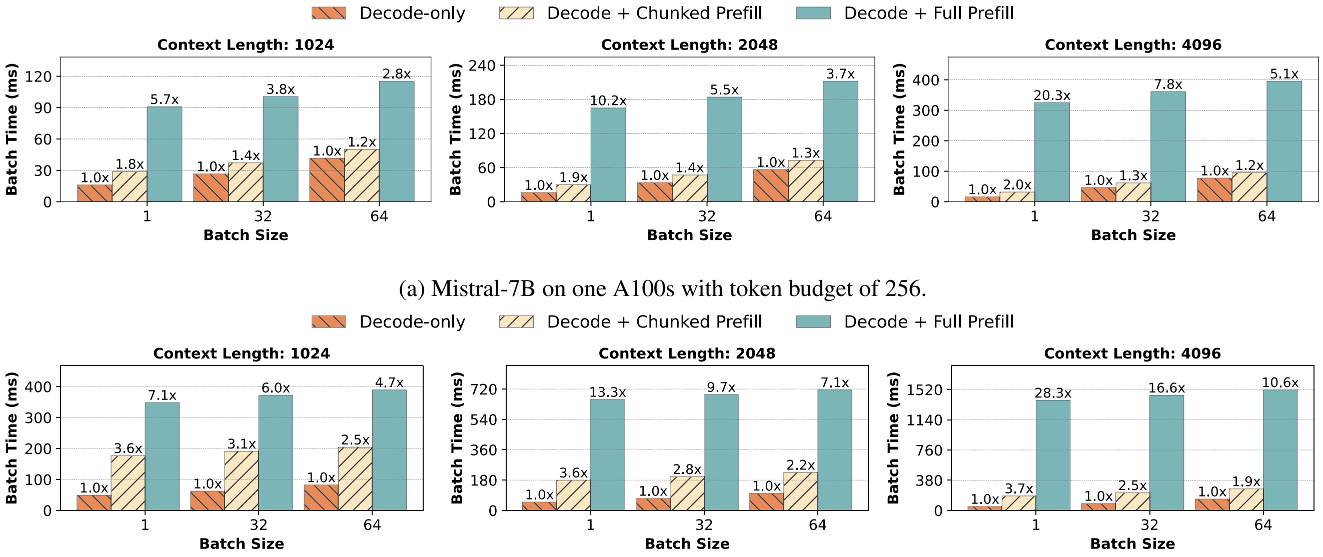
Takeaway-4: *There can be a large variance in compute time of LLM iterations depending on composition of prefill- and decode-tokens in the batch. This can lead to significant bubbles when using pipeline-parallelism.*

4 Sarathi-Serve: Design and Implementation

We now discuss the design and implementation of Sarathi-Serve — a system that provides high throughput with predictable tail latency via two key techniques – *chunked-prefills* and *stall-free batching*.

4.1 Chunked-prefills

As we show in §3.1, decode batches are heavily memory bound with low arithmetic intensity. This slack in arithmetic intensity presents an opportunity to piggyback additional computation in decode batches. Naively, this can be done by creating hybrid batches which combine the memory bound decodes along with compute bound prefills. However, in many practical scenarios, input prompts contain several thousand tokens



(a) Mistral-7B on one A100s with token budget of 256.

(b) LLaMA2-70B on four A100s with token budget of 512.

Figure 9: The incremental cost of coalescing prefills with decode batches. We consider two batching schemes – (i) Decode + Full Prefill represents the hybrid batching of Orca wherein the entire prefill is executed in a single iteration along with ongoing decodes. (ii) Decode + Chunked Prefill represents Sarathi-Serve wherein prefills are chunked before being coalesced with ongoing decodes with a fixed token budget. Sarathi-Serve processes prefill tokens with much lower impact on the latency of decodes. Further, the relative impact of Sarathi-Serve on latency reduces with higher decode batch size and context lengths.

on average *e.g.,* Table 2 shows that the median prompt size in *openchat_sharegpt4* and *arxiv_summarization* datasets is 1730 and 7059 respectively. Combining these long prefills with decode iterations would lead to high TBT latency.

To tackle this challenge, we present a technique called *chunked-prefills* which allows computing large prefills in small chunks across several iterations. *Chunked-prefills* is a prefill splitting mechanism hinged on two key insights. First, as discussed in §3.1, a prefill request with modest sequence length can effectively saturate GPU compute. For example, in Figure 4, prefill throughput starts saturating around sequence length of 512 tokens. Second, in many practical scenarios, input prompts contain several thousand tokens on average (Table 2). This provides an opportunity to break large prefill requests into smaller units of compute which are still large enough to saturate GPU compute. In Sarathi-Serve, we leverage this mechanism to form batches with appropriate number of tokens such that we can utilize the compute potential in decode batches without violating the TBT SLO.

4.2 *Stall-free batching*

The Sarathi-Serve scheduler is an iteration-level scheduler that leverages *chunked-prefills* and coalescing of prefills and decodes to improve throughput while minimizing latency.

Unlike Orca and vLLM which stall existing decodes to execute prefills, Sarathi-Serve leverages the arithmetic intensity slack in decode iterations to execute prefills without delaying the execution of decode requests in the system. We call this approach *stall-free batching* (Algorithm 3). Sarathi-Serve first calculates the budget of maximum number of tokens that can be executed in a batch based on user specified SLO. We describe the considerations involved in determining this token budget in depth in §4.3. In every scheduling iteration, we first pack all the running decodes in the next batch (lines 6-8 in Algorithm 3). After that, we include any partially completed prefill (lines 9-12). Only after all the running requests have been accommodated, we admit new requests (lines 13-20). When adding prefill requests to the batch, we compute the maximum chunk size that can be accommodated within the leftover token budget for that batch (lines 11, 15). By restricting the computational load in every iteration, *stall-free batching* ensures that decodes never experience a generation stall due to a co-running prefill chunk. We compare the latency for hybrid batches with and without chunked prefills in Figure 9. Naive hybrid batching leads to dramatic increase of up to 28.3× in the TBT latency compared to a decode-only batch. In contrast, Sarathi-Serve provides a much tighter bound on latency with chunking.

Figure 7 shows the scheduling policy of Sarathi-Serve in action, for the same example used in §3.2. The first iteration is decode-only as there are no prefills to be computed. However, after a new request C enters the system, Sarathi-Serve first splits the prefill of C into two chunks and schedules them in

Algorithm 3 *Stall-free batching* with Sarathi-Serve. First the batch is filled with with ongoing decode tokens (lines 6-8) and optionally one prefill chunk from ongoing (lines 10-12). Finally, new requests are added (lines 13-20) within the token budget so as to maximize throughput with minimal latency impact on the TBT of delaying the ongoing decodes.

```
 1: Input: $T_{max}$, Application TBT SLO.
 2: Initialize $token\_budget$, $\tau \leftarrow$ compute_token_buget($T_{max}$)
 3: Initialize $batch\_num\_tokens$, $n_t \leftarrow 0$
 4: Initialize current batch $B \leftarrow \emptyset$
 5: while True do
 6:     for $R$ in $B$ do
 7:         if is_prefill_complete($R$) then
 8:             $n_t \leftarrow n_t + 1$
 9:     for $R$ in $B$ do
10:         if not is_prefill_complete($R$) then
11:             $c \leftarrow$ get_next_chunk_size($R$, $\tau$, $n_t$)
12:             $n_t \leftarrow n_t + c$
13:     $R_{new} \leftarrow$ get_next_request()
14:     while can_allocate_request($R_{new}$) $\wedge$ $n_t < \tau$ do
15:         $c \leftarrow$ get_next_chunk_size($R_{new}$, $\tau$, $n_t$)
16:         if $c > 0$ then
17:             $n_t \leftarrow n_t + c$
18:             $B \leftarrow R_{new}$
19:         else
20:             break
21:
22:     process_hybrid_batch($B$)
23:     $B \leftarrow$ filter_finished_requests($B$)
24:     $n_t \leftarrow 0$
```

subsequent iterations. At the same time, with *stall-free batching*, it coalesces the chunked prefills with ongoing decodes of A and B. This way, Sarathi-Serve stalls neither decodes nor prefills unlike existing systems, allowing Sarathi-Serve to be largely free of latency spikes in TBT without compromising throughput. Furthermore, *stall-free batching* combined with *chunked-prefills* also ensures uniform compute hybrid batches in most cases, which helps reduce bubbles when using pipeline parallelism, thereby enabling efficient and scalable deployments.

4.3 Determining Token Budget

The token budget is determined based on two competing factors — TBT SLO requirement and *chunked-prefills* overhead. From a TBT minimization point of view, a smaller token budget is preferable because iterations with fewer prefill tokens have lower latency. However, smaller token budget can result in excessive chunking of prefills resulting in overheads due to 1) lower GPU utilization and 2) repeated KV-cache access in the attention operation which we discuss below.

During the computation of *chunked-prefills*, the attention operation for every chunk of a prompt needs to access the KV-cache of *all* prior chunks of the same prompt. This results in increased memory reads from the GPU HBM even though the computational cost is unchanged. For example, if a prefill sequence is split into N chunks, then the first chunk's KV-cache is loaded $N-1$ times, the second chunk's KV-cache is loaded $N-2$ times, and so on. However, we find that even at small chunk sizes attention prefill operation is compute bound operation. In practice, there can be small overhead associated with chunking due to fixed overheads of kernel launch, etc. We present a detailed study of the overheads of *chunked-prefills* in §5.4.

Thus, one needs to take into account the trade-offs between prefill overhead and decode latency while determining the token budget. This can be handled with a one-time profiling of batches with different number of tokens and setting the token budget to maximum number of tokens that can be packed in a batch without violating TBT SLO.

Another factor that influences the choice of token budget is the *tile-quantization* effect [13]. GPUs compute matmuls by partitioning the given matrices into tiles and assigning them to different thread blocks for parallel computation. Here, each thread block refers to a group of GPU threads and computes the same number of arithmetic operations. Therefore, matmuls achieve maximum GPU utilization when the matrix dimensions are divisible by the tile size. Otherwise, due to *tile-quantization*, some thread blocks perform extraneous computation [13]. We observe that tile-quantization can dramatically increase prefill computation time *e.g.,* in some cases, using chunk size of 257 can increase prefill time by 32% compared to that with chunk size 256.

Finally, when using pipeline parallelism the effect of token budget on pipeline bubbles should also be taken into account. Larger chunks lead to higher inter-batch runtime variations that result in pipeline bubbles which results in lower overall system throughput. On the other hand, picking a very small token budget can lead to higher overhead due to lower arithmetic intensity and other fixed overheads.

Therefore, selecting a suitable token budget is a complex decision which depends on the desired TBT SLO, parallelism configuration, and specific hardware properties. We leverage Vidur [28], a LLM inference profiler and simulator to determine the token budget that maximizes system capacity under specific deployment scenario.

4.4 Implementation

We implement Sarathi-Serve on top of the open-source implementation of vLLM [23, 53]. We added support for paged chunk prefill using FlashAttention v2 [38] and FlashInfer [74] kernels. We use FlashAttention backend for all the evaluations in this paper due to its support for wider set of models. We also extend the base vLLM codebase to support various

Model	Attention Mechanism	GPU Configuration	Memory Total (per-GPU)
Mistral-7B	GQA-SW	1 A100	80GB (80GB)
Yi-34B	GQA	2 A100s (TP2)	160GB (80GB)
LLaMA2-70B	GQA	8 A40s (TP4-PP2)	384GB (48GB)
Falcon-180B	GQA	4 A100s×2 nodes (TP4-PP2)	640GB (80GB)

Table 1: Models and GPU configurations (GQA: grouped-query attention, SW: sliding window).

Dataset	Prompt Tokens			Output Tokens		
	Median	P90	Std.	Median	P90	Std.
openchat_sharegpt4	1730	5696	2088	415	834	101
arxiv_summarization	7059	12985	3638	208	371	265

Table 2: Datasets used for evaluation.

scheduling policies, chunked prefills, pipeline parallelism and an extensive telemetry system. We use NCCL [15] for both pipeline and tensor parallel communication. Source code for the project is available at https://github.com/microsoft/sarathi-serve.

5 Evaluation

We evaluate Sarathi-Serve on a variety of popular models and GPU configurations (see Table 1) and two datasets (see Table 2). We consider vLLM and Orca as baseline because they represent the state-of-the-art for LLM inference. Our evaluation seeks to answer the following questions:

1. What is the maximum load a model replica can serve under specific Service Level Objective (SLO) constraints with different inference serving systems (§5.1) and how does this load vary with varying SLO constraints (§5.2)?
2. How does Sarathi-Serve perform under various deployments such as TP and PP? (§5.3)
3. What is the overhead of *chunked-prefills*? (§5.4.1)
4. What is the effect of each of *chunked-prefills* and *stall-free batching* in isolation as opposed to using them in tandem? (§5.4.2)

Models and Environment: We evaluate Sarathi-Serve across four different models Mistral-7B [51], Yi-34B [24], LLaMA2-70B [66] and Falcon-180B [31] – these models are among the best in their model size categories. We use two different server configurations. For all models except LLaMA2-70B we use Azure NC96ads v4 VMs, each equipped with 4 NVIDIA 80GB A100 GPUs, connected with pairwise NVLINK. The machines are connected with a 100 Gbps ethernet connection. For LLaMA2-70B, we use a server with eight pairwise connected NVIDIA 48GB A40 GPUs. We run Yi-34B in a 2-way tensor parallel configuration (TP-2), and LLaMA2-70B and Falcon-180B in a hybrid parallel configuration with four tensor parallel workers and two pipeline stages for (TP4-PP2).

Model	*relaxed* SLO P99 TBT (s)	*strict* SLO P99 TBT (s)
Mistral-7B	0.5	0.1
Yi-34B	1	0.2
LLaMA2-70B	5	1
Falcon-180B	5	1

Table 3: SLOs for different model configurations.

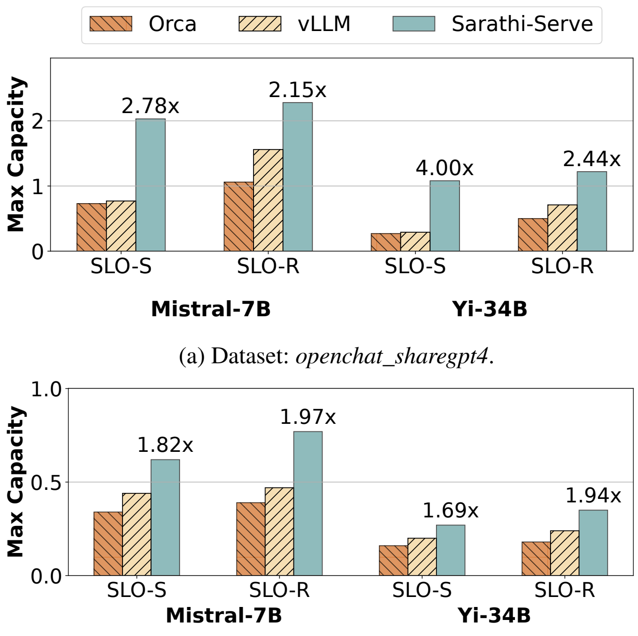
Workloads: In order to emulate the real-world serving scenarios, we generate traces by using the request length characteristics from the *openchat_sharegpt4* [68] and *arxiv_summarization* [36] datasets (Table 2). The *openchat_sharegpt4* trace contains user-shared conversations with ChatGPT-4 [6]. A conversation may contain multiple rounds of interactions between the user and chatbot. Each such interaction round is performed as a separate request to the system. This multi-round nature leads to high relative variance in the prompt lengths. In contrast, *arxiv_summarization* is a collection of scientific publications and their summaries (abstracts) on arXiv.org [3]. This dataset contains longer prompts and lower variance in the number of output tokens, and is representative of LLM workloads such as Microsoft M365 Copilot [14] and Google Duet AI [10] etc. The request arrival times are generated using Poisson distribution. We filter outliers of these datasets by removing requests with total length more than 8192 and 16384 tokens, respectively.

Metrics: We focus on the median value for the TTFT since this metric is obtained only once per user request and on the 99th percentile (P99) for TBT values since every decode token results in a TBT latency value.

5.1 Capacity Evaluation

We evaluate Sarathi-Serve, Orca and vLLM on all four models and both datasets under two different latency configurations: *relaxed* and *strict*. Similar to Patel et al. [58], to account for the intrinsic performance limitations of a model and hardware pair, we define the SLO on P99 TBT to be equal to 5× and 25× the execution time of a decode iteration for a request (with prefill length of 4k and 32 batch size) running without any prefill interference for the *strict* and *relaxed* settings, respectively. Table 3 shows a summary of the absolute SLO thresholds. Note that the *strict* SLO represents the latency target desired for interactive applications like chatbots. On the other hand, the *relaxed* configuration is exemplary of systems where the complete sequence of output tokens should be generated within a predictable time limit but the TBT constraints on individual tokens is not very strict. For all load experiments, we ensure that the maximum load is sustainable, i.e., the queuing delay does not blow up (we use a limit of 2 seconds on median scheduling delay).

Figure 10 and Figure 11 show the results of our capacity



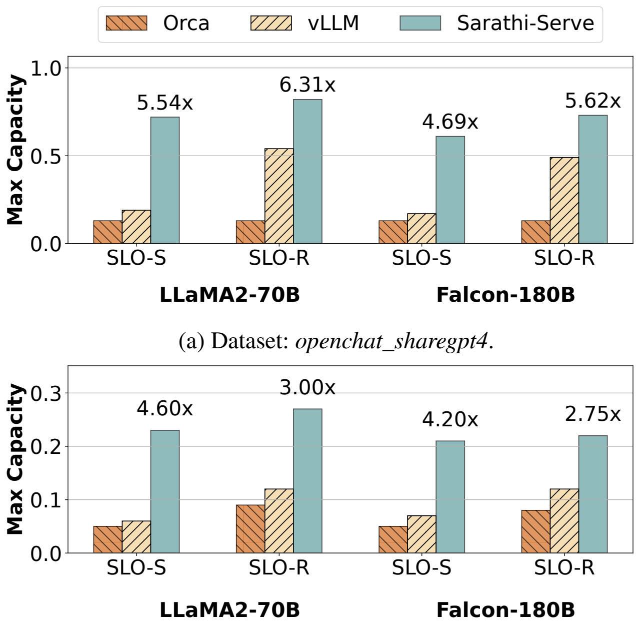
(a) Dataset: *openchat_sharegpt4*.

(b) Dataset: *arxiv_summarization*.

Figure 10: Capacity (in queries per second) of Mistral-7B and Yi-34B with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.

experiments. Sarathi-Serve consistently outperforms Orca and vLLM in all cases across models and workloads. Under *strict* SLO, Sarathi-Serve can sustain up to 4.0× higher load compared to Orca and 3.7× higher load than vLLM under *strict* SLO (Yi-34B, *openchat_sharegpt4*). For larger models using pipeline parallelism, Sarathi-Serve achieves gains of up to 6.3× and 4.3× compared to Orca and vLLM respectively (LLaMA2-70B, *openchat_sharegpt4*) due to few pipeline bubbles.

We observe that in most scenarios, Orca and vLLM violate the P99 TBT latency SLO before they can reach their maximum serviceable throughput. Thus, we observe relaxing the latency target leads to considerable increase in their model serving capacity. In Sarathi-Serve, one can adjust the chunk size based on the desired SLO. Therefore, we use a strict token budget and split prompts into smaller chunks when operating under *strict* latency SLO. This reduces system efficiency marginally but allows us to achieve lower tail latency. On the other hand, when the latency constraint is relaxed, we increase the token budget to allow more efficient prefills. We use token budget of 2048 and 512 for all models under the *relaxed* and *strict* settings, respectively, except for the LLaMA2-70B *relaxed* configuration where we use token budget of 1536 to reduce the impact of pipeline bubbles. The system performance can be further enhanced by dynamically varying the token budget based on workload characteristics. We leave this exploration for future work.



(a) Dataset: *openchat_sharegpt4*.

(b) Dataset: *arxiv_summarization*.

Figure 11: Capacity of LLaMA2-70B and Falcon-180B (models with pipeline parallelism) with different schedulers under strict (SLO-S) and relaxed (SLO-R) latency SLOs.

We further notice that vLLM significantly outperforms Orca under relaxed setting. The reason for this is two-fold. First, Orca batches prompts for multiple requests together (*max sequence length * batch size* compared to *max sequence length* in vLLM), which can lead to even higher tail latency in some cases. Second, vLLM supports a much larger batch size compared to Orca. The lower batch size in Orca is due to the lack of PagedAttention and the large activation memory footprint associated with processing batches with excessively large number of tokens.

Finally, note that the capacity of each system is higher for *openchat_sharegpt4* dataset compared to the *arxiv_summarization* dataset. This is expected because prompts in the *arxiv_summarization* datasets are much longer - 7059 vs 1730 median tokens as shown in Table 2. The larger prompts makes Orca and vLLM more susceptible to latency violations due to higher processing time of these longer prefills.

5.2 Throughput-Latency Tradeoff

To fully understand the throughput-latency tradeoff in LLM serving systems, we vary the P99 TBT latency SLO and observe the impact on system capacity for vLLM and Sarathi-Serve. Figure 12 shows the results for Mistral-7B and Yi-34B models with five different SLO values for the *openchat_sharegpt4* dataset.

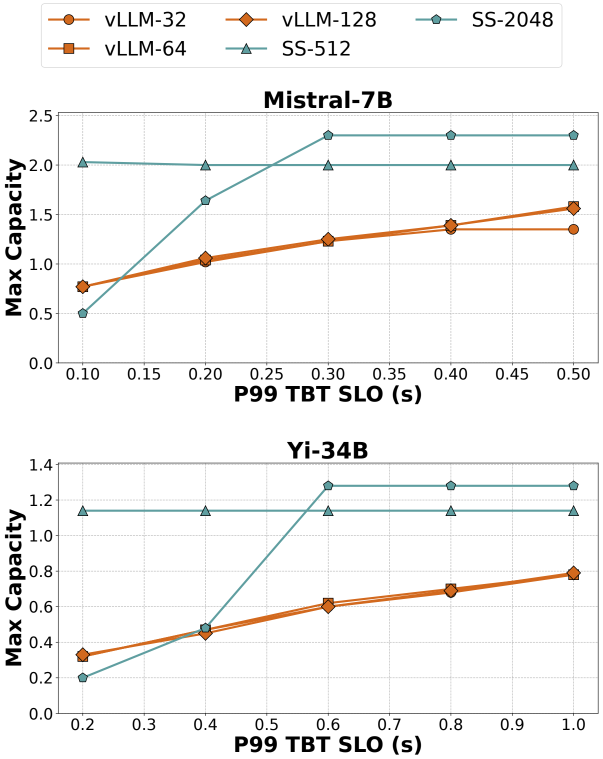
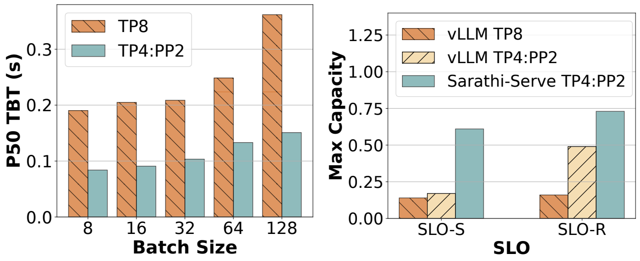


Figure 12: Latency – Throughput tradeoff in vLLM and Sarathi-Serve for Mistral-7B and Yi-34B models on *openchat_sharegpt4* dataset. We evaluate vLLM with three different max batch sizes of 32, 64 and 128. For Sarathi-Serve, we consider token budget of 512 and 2048 with max batch size of 128. Sarathi-Serve delivers 3.5× higher capacity under stringent SLOs for Yi-34B using *Stall-free batching*.

We evaluate vLLM with three different batch sizes in an attempt to navigate the latency-throughput trade-off as prescribed by Yu et al. [75]. The maximum capacity of vLLM gets capped due to generation stalls under stringent TBT SLOs. Notably, the capacity of vLLM remains largely identical for all the three batch size settings. This implies that even though PagedAttention enables large batch sizes with efficient memory management – in practical situations with latency constraints, vLLM cannot leverage the large batch size due to the steep latency-throughput tradeoff made by it's *prefill-prioritizing* scheduler.

On the other hand, the latency-throughput tradeoff in Sarathi-Serve can be precisely controlled by varying the token budget. Sarathi-Serve achieves 3.5× higher capacity compared to vLLM under strict SLO (100ms, Mistral-7B) using a small token budget of 512. For scenarios with more relaxed SLO constraints, picking a larger token budget of 2048 allows Sarathi-Serve to operate more efficiently resulting in 1.65× higher capacity compared to vLLM (1s, Yi-34B).



(a) TBT (Falcon-180B).

(b) Capacity (Falcon-180B).

Figure 13: TP scales poorly across nodes. (a) Median TBT for decode-only batches: cross node TP increases median TBT by more than 2× compared to a 4-way TP within node and PP across nodes. (b) Capacity under strict (SLO-S) and relaxed (SLO-R) latency SLOs: Sarathi-Serve increases Falcon-180B's serving capacity by 4.3× and 3.6× over vLLM's TP-only and hybrid-parallel configurations under strict SLOs.

5.3 Making Pipeline Parallel Viable

We now show that Sarathi-Serve makes it feasible to efficiently serve LLM inference across commodity networks with efficient pipeline parallelism. For these experiments, we run Falcon-180B over two nodes, each with four A100 GPUs, connected over 100 Gbps Ethernet. We evaluate model capacity under three configurations: vLLM with 8-way TP, vLLM with our pipeline-parallel implementation and Sarathi-Serve with pipeline-parallel. For PP configurations, we do 4-way TP within node and 2-way PP across nodes.

Figure 13a shows the latency for decode-only batches for Falcon-180B with purely tensor parallel TP-8 deployment compared to a TP-4 PP-2 hybrid parallel configuration. We observe that the median latency for tensor parallelism is ∼ 2× higher than pipeline parallelism. This is because TP incurs high communication overhead due to cross-node all-reduces.

Figure 13b shows the capacity for tensor and hybrid parallel configurations for Falcon-180B on *openchat_sharegpt4* dataset. Note that unlike the hybrid parallel configuration, TP achieves low capacity even under the *relaxed* SLO due to high latency. Even though vLLM can support a fairly high load with hybrid parallelism under *relaxed* SLO, it's performance drops sharply under the *strict* regime due to pipeline bubbles. Sarathi-Serve on the other hand, leverages *chunked-prefills* to reduce the variation in the execution time between microbatches to avoid pipeline bubbles, resulting in a 1.48× increase in capacity under *relaxed* SLOs and 3.6× increase in capacity under *strict* SLOs.

5.4 Ablation Study

In this subsection, we conduct an ablation study on different aspects on Sarathi-Serve. In particular, we are interested in answering the following two questions: 1) what is the effect of

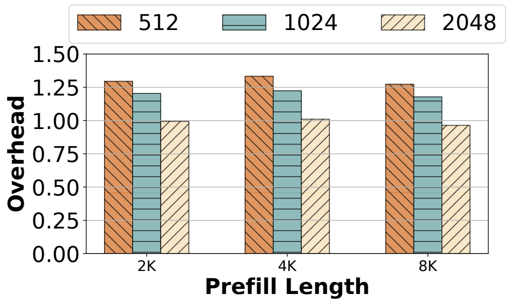


Figure 14: Overhead of *chunked-prefills* in prefill computation for Yi-34B (TP-2) normalized to the cost of no-chunking, shown for various prompt lengths using chunk lengths of 512, 1024 and 2048.

Scheduler	*openchat_sharegpt4*		*arxiv_summarization*	
	P50 TTFT	P99 TBT	P50 TTFT	P99 TBT
hybrid-batching-only	0.53	0.68	3.78	1.38
chunked-prefills-only	1.04	0.17	5.38	0.20
Sarathi-Serve (combined)	0.76	0.14	3.90	0.17

Table 4: TTFT and TBT latency measured in seconds for *hybrid-batching* and *chunked-prefills* used in isolation as well as when they are used in tandem, evaluated over 128 requests for Yi-34B running on two A100s with a token budget of 1024. By using both *hybrid-batching* and *chunked-prefills*, Sarathi-Serve is able to lower both TTFT and TBT.

chunking on prefill throughput, and 2) analyzing the impact of hybrid-batching and chunking on latency. While we provide results only for a few experiments in this section, all the trends discussed below are consistent across various model-hardware combinations.

5.4.1 Overhead of *chunked-prefills*

Figure 14 shows how much overhead chunking adds in Yi-34B – on overall prefill runtime. As expected, smaller chunks introduce higher overhead as shown by the gradually decreasing bar heights in Figure 14. However, even with the smallest chunk size of 512, we observe a moderate overhead of at most ∼25%. Whereas with the larger token budget of 2048, chunked prefills have almost negligible overhead.

5.4.2 Impact of individual techniques

Finally, Table 4 shows the TTFT and TBT latency with each component of Sarathi-Serve evaluated in isolation *i.e., chunked-prefills*-only, *hybrid-batching*-only (mixed batches with both prefill and decode requests) and when they are used in tandem. These results show that the two techniques work best together: *chunked-prefills*-only increases TTFT as prefill chunks are slightly inefficient whereas *hybrid-batching*-only increases TBT because long prefills can still create generation stalls. When used together, Sarathi-Serve improves performance along both dimensions.

6 Related Work

Model serving systems: Systems such as Clipper [37], TensorFlow-Serving [56], Clockwork [45] and BatchMaker [44] study various placement, caching and batching strategies for model serving. However, these systems fail to address the challenges of auto-regressive transformer inference. More recently, systems such as Orca [75], vLLM [53], FlexGen [63], FasterTransformers [7], LightSeq [70], and TurboTransformers [42] propose domain-specific optimizations for transformer inference. FlexGen [63] optimizes LLM inference for throughput in resource-constrained offline scenarios i.e., it is not suitable for online serving. FastServe [72] proposed a preemptive scheduling framework for LLM inference to minimize the job completion times. We present a detailed comparison with Orca and vLLM as they represent the state-of-the-art in LLM inference.

Another approach that has emerged recently is to disaggregate the prefill and decode phases on different replicas as proposed in SplitWise, DistServe and TetriInfer [47, 58, 77]. These solutions can entirely eliminate the interference between prefills and decodes. However, disaggregation requires migrating the KV cache of *each* request upon the completion of its prefill phase which could be challenging in the absence of high-bandwidth interconnects between different replicas. In addition, this approach also under-utilizes the GPU memory capacity of the prefill replicas i.e., only the decode replicas are responsible for storing the KV cache. On the positive side, disaggregated approaches can execute prefills with maximum efficiency (and therefore yield better TTFT) unlike chunked prefills that are somewhat slower than full prefills. We leave a quantitative comparison between Sarathi-Serve and disaggregation-based solutions for future work.

Recently, Sheng et al. [62] proposed modification to iteration-level batching algorithm to ensure fairness among clients in a multi-tenant environment. FastServe [72] uses a preemption based scheduling mechanism to mitigate head-of-the-line blocking. Such algorithmic optimizations are complimentary to our approach and can benefit from lower prefill-decode interference enabled by Sarathi-Serve. Another recent system, APIServe [26] adopted chunked prefills from Sarathi to utilize wasted compute in decode batches for ahead-of-time prefill recomputation for multi-turn API serving.

Improving GPU utilization for transformers: Recent works have proposed various optimizations to improve the hardware utilization for transformers. FasterTransformer uses model-specific GPU kernel implementations. CocoNet [50] and [69] aim to overlap compute with communication to improve GPU utilization: these techniques are specially useful while using a high degree of tensor-parallel for distributed models where communication time can dominate compute. Further, the cost of computing self-attention grows quadratically with sequence length and hence can become significant for long contexts. [38,39,60] have proposed various techniques to minimize the memory bottlenecks of self-attention with careful tiling and work partitioning. In addition, various parallelization strategies have been explore to optimize model placement. These techniques are orthogonal to Sarathi-Serve.

Model optimizations: A significant body of work around model innovations has attempted to address the shortcomings of transformer-based language models or to take the next leap forward in model architectures, beyond transformers. For example, multi-query attention [61] shares the same keys and values across all the attention heads to reduce the size of the KV-cache, allowing to fit a larger batch size on the GPUs. Several recent works have also shown that the model sizes can be compressed significantly using quantization [40, 41, 43, 73]. Mixture-of-expert models are aimed primarily at reducing the number of model parameters that get activated in an iteration [32, 48, 54]. More recently, retentive networks have been proposed as a successor to transformers [65]. In contrast, we focus on addressing the performance issues of popular transformer models from a GPU's perspective.

7 Conclusion

Optimizing LLM inference for high throughput and low latency is desirable but challenging. We presented a broad characterization of existing LLM inference schedulers by dividing them into two categories – *prefill-prioritizing* and *decode-prioritizing*. In general, we argue that the former category is better at optimizing throughput whereas the latter is better at optimizing TBT latency. However, none of them is ideal when optimizing throughput and latency are both important.

To address this tradeoff, we introduce Sarathi-Serve—a system that instantiates a novel approach comprised of *chunked-prefills* and *stall-free batching*. Sarathi-Serve chunks input prompts into smaller units of work to create stall-free schedules. This way, Sarathi-Serve can add new requests in a running batch without pausing ongoing decodes. Our evaluation shows that Sarathi-Serve improves the serving capacity of Mistral-7B by up to 2.6× on a single A100 GPU and up to 5.6× for Falcon-180B on 8 A100 GPUs.

8 Acknowledgement

We would like to thank OSDI reviewers and our shepherd for their insightful feedback. This research is partly supported by GT Cloud Hub, under the auspices of the Institute for Data Engineering and Science (IDEaS), with funding from Microsoft, and the Center for Research into Novel Compute Hierarchies (CRNCH) at Georgia Tech.

References

[1] Amazon codewhisperer. https://aws.amazon.com/codewhisperer/.

[2] Anthropic claude. https://claude.ai.

[3] arxiv.org e-print archive. https://arxiv.org/.

[4] Bing ai. https://www.bing.com/chat.

[5] Character ai. https://character.ai.

[6] Chatgpt. https://chat.openai.com.

[7] Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

[8] Github copilot. https://github.com/features/copilot.

[9] Google bard. https://bard.google.com.

[10] Google duet ai. https://workspace.google.com/solutions/ai/.

[11] Komo. https://komo.ai/.

[12] Lightllm: A light and fast inference service for llm. https://github.com/ModelTC/lightllm.

[13] Matrix multiplication background user's guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[14] Microsoft copilot. https://www.microsoft.com/en-us/microsoft-copilot.

[15] Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl.

[16] Nvidia dgx platform. https://www.nvidia.com/en-us/data-center/dgx-platform/.

[17] NVIDIA Triton Dynamic Batching. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher.

[18] Openai gpt-3: Understanding the architecture. https://www.theaidream.com/post/openai-gpt-3-understanding-the-architecture.

[19] Perplexity ai. https://www.perplexity.ai/.

[20] Replit ghostwriter. https://replit.com/site/ghostwriter.

[21] Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM.

[22] Using NVIDIA's AI/ML Frameworks for Generative AI on VMware vSphere. https://core.vmware.com/blog/using-nvidias-aiml-frameworks-generative-ai-vmware-vsphere.

[23] vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm.

[24] Yi series of large language models trained from scratch by developers at 01.AI. https://huggingface.co/01-ai/Yi-34B-200K.

[25] You.com. https://you.com/.

[26] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Apiserve: Efficient api support for large-language model inferencing. *arXiv preprint arXiv:2402.01869*, 2024.

[27] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[28] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara*, 2024.

[29] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[30] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[31] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models, 2023.

[32] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2022.

[33] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[34] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[35] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[36] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[37] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[38] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[39] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[40] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[41] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[42] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient GPU serving system for transformer models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 389–402. ACM, 2021.

[43] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[44] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[46] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[47] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[48] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin S. Lee, Anjali Sridhar, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference, 2023.

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[50] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[52] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[53] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA, July 2023. USENIX Association.

[55] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[56] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[57] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[58] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[59] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[60] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[61] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[62] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588*, 2023.

[63] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[64] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[65] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023.

[66] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[68] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data, 2023.

[69] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[70] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*, pages 113–120. Association for Computational Linguistics, June 2021.

[71] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[72] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[73] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

[74] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. Accelerating self-attentions for llm serving with flashinfer, February 2024.

[75] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[76] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

[77] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

A Artifact Appendix

Abstract

Our open source artifact is available on GitHub. This repository contains our implementation of Sarathi-Serve as well as the harnesses and scripts for running and plotting the experiments described in this paper.

This repository originally started as a fork of the vLLM project. Sarathi-Serve is a lightweight high-performance research prototype and doesn't have complete feature parity with open-source vLLM. We have only retained the most critical features and adopted the codebase for faster research iterations.

Scope

This artifact allows the readers to validate the claims made in the Sarathi-Serve paper (the figures) and provides a means to replicate the experiments described. The artifact can be used to set up the necessary environment, execute the main results, and perform microbenchmarks, thus providing a comprehensive understanding of the key claims in Sarathi-Serve.

Contents

The repository is structured as follows, the primary source code for the system is contained in directory */sarathi*. The implementations for custom CUDA kernels are within the */csrc* directory. All the scripts to reproduce the experiments are in */osdi-experiments* and finally, the trace files used for the experiments are stored in */data*.

Hosting

You can obtain our artifacts from GitHub: GitHub. The main branch of the Github repository is actively updated, but we will maintain clear and accessible instructions about our artifacts in an easily identifiable README file. All the detailed instructions and README files to reproduce the experiments in the OSDI paper are available in the branch *osdi-sarathi-serve*.

Requirements

Sarathi-Serve has been tested with CUDA 12.1 on A100 and A40 GPUs. The specific GPU SKUs on which the experiments were performed and the parallelism strategies used are clearly explained in the README corresponding to the figures in the artifact, for ease of reproducibility.

ServerlessLLM: Low-Latency Serverless Inference for Large Language Models

Yao Fu[1] Leyang Xue[1] Yeqi Huang[1] Andrei-Octavian Brabete[1] Dmitrii Ustiugov[2] Yuvraj Patel[1] Luo Mai[1]

[1]*University of Edinburgh* [2]*NTU Singapore*

Abstract

This paper presents ServerlessLLM, a distributed system designed to support low-latency serverless inference for Large Language Models (LLMs). By harnessing the substantial near-GPU storage and memory capacities of inference servers, ServerlessLLM achieves effective local checkpoint storage, minimizing the need for remote checkpoint downloads and ensuring efficient checkpoint loading. The design of ServerlessLLM features three core contributions: (i) *fast multi-tier checkpoint loading*, featuring a new loading-optimized checkpoint format and a multi-tier loading system, fully utilizing the bandwidth of complex storage hierarchies on GPU servers; (ii) *efficient live migration of LLM inference*, which enables newly initiated inferences to capitalize on local checkpoint storage while ensuring minimal user interruption; and (iii) *startup-time-optimized model scheduling*, which assesses the locality statuses of checkpoints on each server and schedules the model onto servers that minimize the time to start the inference. Comprehensive evaluations, including microbenchmarks and real-world scenarios, demonstrate that ServerlessLLM dramatically outperforms state-of-the-art serverless systems, reducing latency by 10 - 200X across various LLM inference workloads.

1 Introduction

Large Language Models (LLMs) have recently been integrated into various online applications, such as programming assistants [26], search engines [21], and conversational bots [54]. These applications process user inputs, such as questions, by breaking them down into tokens (e.g., words). LLMs generate responses in an autoregressive manner, predicting each subsequent token based on the combination of input tokens and those already generated, until a sentence-ending token (EoS) is reached. To optimize this process, LLMs utilize key-value caches to store intermediate results, thereby minimizing redundant computations.

Serving LLMs at scale presents significant challenges due to the extensive GPU resources required and the stringent low response time constraints demanded by interactive services. Additionally, LLM inference latency is unpredictable as it depends on the output length, which varies significantly due to iterative token generation [27, 42, 81].

To achieve low latency, processing an LLM request often requires multiple GPUs for durations ranging from seconds to minutes. In practice, service providers hosting LLMs need to cater to a diverse range of developers, leading to substantial GPU consumption [20] and impacting the sustainability of LLM services [23]. Consequently, LLM inference services are compelled to impose strict limits on the number of requests users can send (e.g., 40 messages per 3 hours for ChatGPT [54]), highlighting the providers' current challenges in meeting demand. Researchers predict that LLM inference costs could escalate by more than 50 times as it approaches the popularity of Google Search [23].

To reduce GPU consumption, LLM service operators are turning to serverless inference, as demonstrated in platforms such as Amazon SageMaker [63], Azure [50], KServe [16], and HuggingFace [35]. In this model, developers upload their LLM checkpoints, which include both model execution and parameter files, to a checkpoint storage system. When a request is received, a model loading scheduler selects available GPUs to initiate these checkpoints. A request router then directs the inference request to the selected GPUs. This serverless approach allows infrastructure providers to efficiently multiplex LLMs on GPUs, improving resource utilization. Additionally, it offers economic advantages to infrastructure users, who incur costs only for each request's duration, thereby avoiding expensive long-term GPU reservations.

While serverless inference offers cost savings for deploying LLMs, it also introduces significant latency overheads. These overheads are commonly attributed to inference cold starts, a frequent issue in serverless workloads, as demonstrated by public traces [64, 92]. Cold starts are especially prolonged for LLM checkpoints, whose sizes can range from gigabytes [11, 76, 91] to terabytes [29]. The vast size is due to the immense number of parameters in such models, leading to notable delays when downloading from remote storage.

Moreover, these checkpoints consist of numerous tensors, each with unique structures and sizes. The complex process of loading these tensors onto GPUs, which involves file deserialization, memory allocation, and tensor shape parsing, further compounds these delays.

We aim to explore system designs that support low-latency serverless inference for LLMs. We note that GPU-based inference servers typically feature a sophisticated yet underutilized storage hierarchy, equipped with extensive host memory and storage capacities. Current serverless inference systems, such as KServe [16] and Ray Serve [73], often only utilize a fraction of the available host memory and minimally employ SSDs for caching checkpoints from the model repository. This observation has led us to propose a novel system design: leveraging the multi-tier storage hierarchy for local checkpoint storage and harnessing their significant storage bandwidth for efficient checkpoint loading.

However, several open concerns arise when implementing local checkpoint storage: (i) Given the complex storage architecture of a GPU server, which includes multiple GPUs, DRAM, SSDs, and remote storage, all interconnected through various links such as PCIe, NVMe, and network connections, how can we optimize the loading of LLM checkpoints to fully exploit the available bandwidth? (ii) Assigning requests to servers with pre-loaded checkpoints can avoid the need for remote checkpoint downloads, but this strategy might lead to prolonged queuing delays or high preemption costs. This is particularly challenging as LLMs typically have long, unpredictable inference durations, which differ markedly from traditional deep neural network inference. (iii) In a distributed cluster where model requests are concurrently served and checkpoints are preloaded onto various layers of local storage, which servers should be strategically selected to minimize the time required to start a model inference?

To address the above, we have designed and implemented ServerlessLLM, which includes three core contributions:

(1) Fast multi-tier checkpoint loading. ServerlessLLM can maximize the storage bandwidth usage of GPU servers for LLM checkpoint loading. It introduces (i) a new *loading-optimized checkpoint* that supports sequential, chunk-based reading and efficient tensor in-memory addressing, and (ii) an *efficient multi-tier checkpoint storage system* that can harness the substantial capacity and bandwidth on a multi-tier storage hierarchy, through an in-memory data chunk pool, memory-copy efficient data path, and a multi-stage data loading pipeline.

(2) Efficient live migration of LLM inference. We motivate the need for live migration of LLM inference and are the first to implement LLM live migration in serverless inference systems to enhance the performance when supporting locality-driven inference. To achieve high efficiency when migrating LLM inference, we have implemented two strategic designs: (i) the source server migrates only the tokens, rather than the large kv-cache, which significantly reduces network traffic during the migration; and (ii) it triggers an efficient re-computation of the kv-cache at the destination server, ensuring migration can complete in a timely manner.

(3) Startup-time-optimized model scheduling. ServerlessLLM aids serverless inference systems by enabling latency-preserving, locality-aware model scheduling. It integrates cost models for accurately estimating the time of loading checkpoints from different tiers in the storage hierarchy and the time of migrating an LLM inference to another server. Based on the estimation results, Phantom can choose the best server to minimize model startup latency.

We have conducted comprehensive evaluation to compare ServerlessLLM against various baseline methods in a GPU cluster. Micro-benchmark results revealed that ServerlessLLM's LLM checkpoint loading significantly outperforms existing systems such as Safetensors [36], and PyTorch [58], achieving loading times that are 3.6 - 8.2X faster. This performance enhancement is particularly notable with large LLMs like OPT [91], LLaMA-2 [76], and Falcon [11]. ServerlessLLM also supports emerging LoRA adaptors [34], achieving 4.4X speed-ups in checkpoint loading.

Furthermore, we evaluated ServerlessLLM with real-world serverless workloads, modeled on the public Azure Trace [64], and benchmarked it against KServe, Ray Serve, and a Ray Serve variant with local checkpoint caching. In these scenarios, ServerlessLLM demonstrated a 10 to 200 times improvement in latency for running OPT model inferences across datasets (i.e., GSM8K [25] and ShareGPT [83]). These results underscore ServerlessLLM's effectiveness in combining fast checkpoint loading, efficient inference migration, and optimized scheduling for model loading. The source code for ServerlessLLM is released at https://github.com/ServerlessLLM/ServerlessLLM.

2 Background and Motivation

2.1 Why Serverless Inference for LLMs

Numerous companies, including Amazon, Azure, Google, HuggingFace, Together AI [6], Deepinfra [2], Replicate [5], Databricks [7], Fireworks-AI [3], and Cohere [4], have introduced serverless inference services (also known as serverless model entrypoints). These services enable users to deploy standard open-source LLMs either in their original form or by modifying them through fine-tuning or by running custom-built models.

Serverless inference can significantly reduce costs for LLM users by charging only for the duration of inference and the volume of processed data. These serverless platforms also offer functionalities such as auto-scaling and auto-failure-recovery to keep instances in an "always-on" state. For the providing companies, serverless inference allows effective

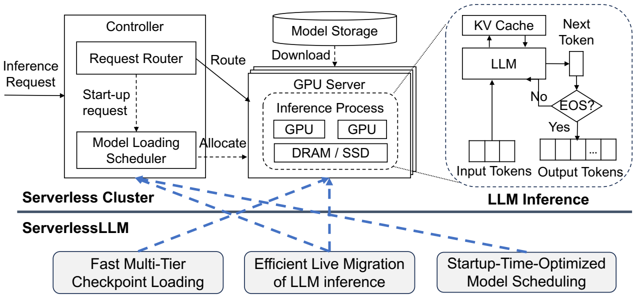


Figure 1: Overview of GPU serverless clusters, LLM inference and new designs introduced by ServerlessLLM.

multiplexing of models within a GPU cluster, improving resource utilization, and generating a software premium for managing infrastructure on behalf of users.

Serverless inference systems are especially advantageous for LLM applications with dynamic and unpredictable workloads. These may include newly launched products without clear predictions of user engagement (e.g., the launch of the ChatGPT service) or those facing spontaneous and unpredictable demands, which are typical in sectors such as healthcare, education, legal, and sales. Unlike global-scale LLM services, these applications are activated only when users access the LLM service.

2.2 Serverless Cluster and LLM Inference

We introduce the key components in GPU serverless clusters in Figure 1. Upon receiving a new inference request, the controller dispatches it to GPU-equipped nodes in a cluster running LLM inference service instances, and to cloud storage hosting model checkpoints. The controller typically consists of two main components: the *request router* and the *model loading scheduler*. The request router directs incoming requests to nodes already running LLM inference processes, or instructs the model loading scheduler to activate LLM inference processes on unallocated GPUs. The selected GPU node initiates a GPU process/container, setting up an inference library (e.g., HuggingFace Accelerate [32] and vLLM [42]). This inference process involves downloading the requested model's checkpoint from a remote model storage and loading it into the GPU, passing through SSD and DRAM.

The LLM inference process often handle requests that include user-specified input prompts, i.e., a list of tokens, as shown in Figure 1. This process iteratively generates tokens based on the prompt and all previously generated tokens, continuing until an end-of-sentence token (denoted as EoS) is produced, resulting in non-deterministic total inference time [55]. During each iteration, the LLM caches intermediate computations in a KV-cache to accelerate subsequent token generation [42, 56]. The tokens generated by each iteration are continuously streamed back to the requesting client, making LLM applications interactive by nature. Their performance is thus measured by both first-token latency (i.e., the time to return the first token) and per-token latency (i.e., the average time to generate a token).

2.3 Challenges with Serverless LLM Inference

The deployment of LLMs on serverless systems, although promising, often incurs significant latency overheads. This is largely due to the substantial proportions of cold-start in serverless clusters, as demonstrated by public data: the Azure Trace [64] shows that over 40% of functions exhibit a cold-start rate exceeding 25%, and approximately 25% of functions experience a cold-start rate greater than 60%, within a 5-minute keep-alive interval. These figures align with the findings from our experiments, underscoring the impact of cold-starts in real-world settings. Consequently, many serverless providers, including Bloomberg, have publicly acknowledged experiencing extremely high latencies, often reaching tens of seconds, when initializing state-of-the-art LLMs for inference on their platforms.

We observe several primary reasons for the prolonged LLM cold-start latency:

(1) LLM checkpoints are large, prolonging downloads. LLM checkpoints are significantly larger than conventional DNN checkpoints, which leads to longer download times. For instance, Grok-1 [82] checkpoints are over 600 GB, DBRX [72] are 250GB, and Mixtral-8x22B [71] are about 280GB [1] Downloading such large checkpoints from remote storage becomes costly. For example, acquiring an LLM checkpoint with a size of 130GB (e.g., LLaMA-2-70B [76]) from S3 or blob storage takes a minimum of 26 seconds using a fast commodity network capable of 5GB/s [19].

(2) Loading LLM checkpoints incurs a lengthy process. Even when model checkpoints are stored locally on NVMe SSDs, loading these checkpoints into GPUs remains a complex process including model initialization, GPU memory allocation, tensor creation, and tensor data copy, typically taking tens of seconds (as detailed in §7.2). For instance, loading the OPT-30B model into 4 GPUs requires 34 seconds using PyTorch, and loading LLaMA-2-70B into 8 GPUs takes 84 seconds. This loading latency far exceeds the time required for generating a token during the inference process, which is usually less than 100ms [55]. Consequently, the prolonged first-token latency can significantly disrupt user experience.

2.4 Existing Solutions and Associated Issues

To improve the latency performance when supporting LLMs, existing solutions show a variety of issues:

(1) Over-subscribing GPUs. The prevalent solutions [13, 84], aimed at circumventing model download and loading times

[1] Model size calculated in float16 precision.

in serverless inference clusters, frequently involve oversubscribing GPUs to accommodate peak demand scenarios. For instance, AWS Serverless Inference [13] maintains a certain number of GPU instances in a warmed state to alleviate the impacts of slow cold starts. While this strategy is effective for managing conventional smaller models, such as ResNet and BERT, it proves challenging for LLMs, which require substantially greater resources from costly GPUs.

(2) Caching checkpoints in host memory. Several solutions [33, 39] have been developed that cache model checkpoints in the host memory of GPU servers to eliminate the need for model downloads. This approach is typically effective for smaller conventional models (e.g., up to a few GBs [39]). However, solely relying on host-memory-based caching proves inadequate for LLMs. LLMs can easily exceed hundreds of GBs in size, challenging the capacity of host memory to store a sufficient number of their checkpoints adequately. The limited size of host memory leads to significant cache misses, resulting in frequent model downloads, as further discussed in §7.4.

(3) Deploying additional storage servers. Various strategies [19] recommend the deployment of additional storage servers within a local cluster to cache model checkpoints. Despite these enhancements, recent trace studies [19] indicate that model downloads can exceed 20 seconds through an optimized pipeline, even when connected to local commodity storage servers equipped with a 100 Gbps NIC. Although the integration of faster networks (e.g., 200 Gbps Ethernet or InfiniBand) could reduce this latency, the associated costs of implementing additional storage servers and high-bandwidth networks are substantial [18, 31]. For instance, utilizing network-optimized AWS ElasticCache servers [1] to support a 70B model can lead to a 100% increase in costs. Specifically, cache.c7gn.16xlarge servers, which provide 210GB of memory and 200 Gbps of network performance, are priced at $16.3/h, equivalent to the cost of an 8-GPU g5.48xlarge server.

3 Exploiting In-Server Multi-Tier Storage

ServerlessLLM addresses the challenges highlighted in the previous sections—namely, high model download times and lengthy model loading—using a design approach that is cost-effective, scalable, and long-term viable.

3.1 Design Intuitions

Our design is inspired by the simple observation that GPU servers used for inference feature a multi-tier storage hierarchy with substantial capacity and bandwidth. From a capacity standpoint, these servers are equipped with extensive memory capabilities. For example, a contemporary 8-GPU server can support up to 4 TBs of main memory, 64 TBs on NVMe SSDs, and 192 TBs on SATA SSDs [52]. Additionally, we observe that in the serverless inference context, a significant portion of the host memory and storage devices in GPU servers remains underutilized.

Regarding bandwidth, GPU servers typically house multiple GPUs, each connected to the host memory via a dedicated PCIe connection, providing significant aggregated bandwidth between the memory and GPU. NVMe and SATA SSDs also connect through their respective links and can be configured in RAID to enhance throughput. For instance, an 8-GPU server utilizing PCIe 5.0 technology can achieve an aggregated bandwidth of 512 GB/s between the host memory and GPUs, and around 60 GB/s from NVMe SSDs (RAID 0) to host memory.

Building on these observations, we propose a design approach that leverages the unused in-server multi-tier storage capacity to store models locally and load them more rapidly, thus reducing latency. This approach is (i) *cost-effective*, as it reutilizes existing, underutilized storage resources in GPU servers; (ii) *scalable*, given that the available local storage capacities and bandwidth can naturally increase with the addition of more inference servers; and (iii) *long-term viable*, as upcoming GPU servers will include even greater capacities and bandwidth (*e.g.*, each Grace-Hopper GPU features 1 TB on-chip DRAM and a 900GB/s C2C link between on-chip DRAM and HBM).

3.2 Design Concerns and Overview

In implementing our design, we identify three crucial concerns that must be addressed.

(1) Support complex multi-tiered storage hierarchy. Current checkpoint and model loading tools such as PyTorch [58], TensorFlow [75], and ONNX Runtime [62] are primarily designed to enhance the training and debugging phases of model development. However, these tools are not optimized for read performance, which becomes critically important in a serverless inference environment. In these settings, model checkpoints are stored once but need to be frequently loaded and accessed across multiple GPUs. This insufficient optimization for read operations results in significant loading delays. While solutions like Safetensors [36] can enhance loading performance, as demonstrated in Section 7, they still fail to fully leverage the capabilities of a multi-tiered storage hierarchy.

(2) Strong locality-driven inference. Supporting efficient model loading alone is insufficient; we also need approaches that can effectively schedule requests onto GPU servers with locally stored checkpoints. Implementing locality-driven LLM inference, however, presents challenges. Current ML model serving systems such as ClockWork [33] and Shepherd [90] take checkpoint locality into account. Yet, they either depend on accurate predictions of model inference time, which is problematic with LLMs, or they preempt ongoing model inferences, causing significant downtime and redundant computations. Therefore, ServerlessLLM must adopt a new approach that is tailored to the unique characteristics of LLM inference (*i.e.*, this workload is interactive and features long, unpredictable durations), necessitating the support for inference live migration, which is further detailed in Section 5.

(3) Scheduling models for optimized startup time. ServerlessLLM is designed to minimize the model startup latency. The cluster scheduler (or controller) plays a crucial role in scheduling models onto GPU resources to answer incoming inference requests. However, the scheduler needs to carefully consider the checkpoint's locality in the entire cluster. Many factors may influence the overall startup latency, such as the difference in the bandwidth offered by each layer in the memory hierarchy. There may be instances where it is beneficial to move the current inference execution to a new GPU than to allocate the request to a GPU where the model may have to be loaded from the storage media. Hence, ServerlessLLM needs to accurately estimate the startup times considering the cluster's checkpoint locality status and accordingly allocate resources to minimize startup time.

Overview. ServerlessLLM addresses these concerns with three novel designs, as depicted in Figure 1. Firstly, it facilitates fast multi-tier checkpoint loading (Section 4) to fully utilize the storage capacity and bandwidth of each GPU server. It also coordinates GPU servers and the cluster controller for efficient live migration of LLM inference (Section 5), ensuring locality-driven inference with minimal resource overhead and user disruption. Lastly, ServerlessLLM features a startup-time-optimized model scheduling policy (Section 6) implemented in its controller, effectively analyzing the checkpoint storage status of each server within a cluster, and it chooses a server for initiating a model, minimizing its startup time.

4 Fast Multi-Tier Checkpoint Loading

In this section, we introduce the design of fast multi-tier checkpoint loading in ServerlessLLM, with several key objectives: (i) to fully utilize the bandwidth and capacity of multi-tier local storage on GPU servers, (ii) to ensure predictable loading performance, critical for ServerlessLLM's readiness in low-latency inference clusters, and (iii) to maintain a generic design that supports checkpoints from various deep learning frameworks.

4.1 Loading-Optimized Checkpoints

Our design is motivated by the observation that LLM checkpoints are often written frequently during training and debugging but loaded infrequently. Conversely, in serverless inference environments, checkpoints are uploaded once and loaded multiple times. This discrepancy has inspired us to convert these checkpoints into a loading-optimized format.

To ensure our design is generic for different frameworks,

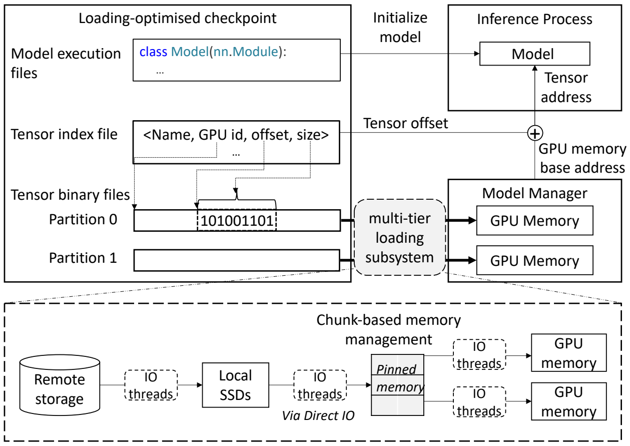


Figure 2: Components in fast multi-tier checkpoint loading.

we operate under a set of assumptions that are common in checkpoints. The checkpoints have: (i) *Model execution files* which define the model architecture. Depending on the framework, the format varies; TensorFlow typically uses protobuf files [74], while PyTorch employs Python scripts [57]. Beyond architecture, these files detail the size and shape of each tensor and include a model parallelism plan. This plan specifies the target GPU for each tensor during checkpoint loading. (ii) *Model parameter files* which stores the binary data of parameters in an LLM. Tensors within these files can be arranged in any sequence. Runtimes such as PyTorch may also store tensor shapes as indices to calculate the offset and size for each tensor.

To ensure fast loading performance, we implement two main features for the converted checkpoints: (i) *Sequential chunk-based reading*: To ensure efficient sequential reading, tensors for each GPU are grouped in partitions (shown in Figure 2). These files contain only the binary data of model parameters and exclude metadata such as tensor shapes, facilitating large chunk reading. (ii) *Direct tensor addressing*: We create a tensor index file (shown in Figure 2) that maps tensor names to a tuple of GPU id, offset, and size, facilitating the efficient restoration of tensors. The tensors are aligned with memory word sizes, facilitating direct computation of memory address.

We observe that decoupling the loading and inference processes can further enhance loading performance. This separation allows checkpoint loading to be pre-scheduled and overlapped with the initialization of the inference process. For this, ServerlessLLM uses a model manager to load tensor data, while allowing the inference process to focus on initializing the model by setting the data pointers for each tensor. More specifically, the model manager allocates memory on GPUs and loads the binary data of the checkpoint via a fast multi-tier loading subsystem (see details in 4.2). The inference process initializes the model object and sets the GPU memory address for each tensor. It acquires the base addresses for each GPU (i.e., CUDA IPC handles) from the model manager and reads the tensor offset from the tensor index file, facilitating the computation of the tensor GPU memory address (i.e., *base* + *offset*). To ensure the model is fully initialized before inference, the inference process and the model manager perform a synchronization.

4.2 Multi-Tier Loading Subsystem

To achieve fast and predictable checkpoint loading performance, we design a multi-tier loading subsystem, integrated within the model manager. This subsystem incorporates several techniques:

Chunk-based data management. For fast loading performance, we have implemented chunk-based data management with three main features: (i) *Utilizing parallel PCIe links*. To mitigate the bottleneck caused by a single PCIe link from storage when loading multiple models into GPUs, we employ parallel DRAM-to-GPU PCIe links to facilitate concurrent checkpoint loading across GPUs. (ii) *Supporting application-specific controls*. Our memory pool surpasses simple caching by providing APIs for the allocation and deallocation of memory. This enables fine-grained management of cached or evicted data chunks, based on specific requirements of the application. (iii) *Mitigating memory fragmentation*. We address latency and space inefficiencies caused by memory fragmentation by using fixed-size memory chunks.

Predictable data path. We have created an efficient data path in our model manager with two main strategies: (i) *Exploiting direct file access*. We use direct file access (e.g., ‘O_DIRECT’ in Linux) to avoid excessive data copying by directly reading data into user space. This method outperforms memory-mapped files (mmap), currently adopted in high-speed loaders such as Safetensors [36], which rely on system cache and lack consistent performance guarantees (critical for predictable performance). (ii) *Exploiting pinned memory*. We utilize pinned memory to eliminate redundant data copying between DRAM and GPU. This approach allows direct copying to the GPU with minimal CPU involvement, ensuring efficient use of PCIe bandwidth with a single thread.

Multi-tier loading pipeline. We have developed a multi-tier loading pipeline to support various storage interfaces and improve loading throughput. This pipeline has three features: (i) *Support for multiple storage interfaces*. ServerlessLLM offers dedicated function calls for various storage interfaces, including local storage (e.g., NVMe, SATA), remote storage (e.g., S3 object store [12]), and in-memory storage (pinned memory). It utilizes appropriate methods for efficient data access in each case. (ii) *Support for intra-tier concurrency*. To leverage modern storage devices’ high concurrency, ServerlessLLM employs multiple I/O threads for reading data within each storage tier, improving bandwidth utilization. (iii) *Flexible pipeline structure*. We use a flexible task queue-based pipeline design, supporting new storage tiers to be efficiently integrated. I/O threads read storage chunks and enqueue their indices (offset and size) for the I/O threads in the next tier.

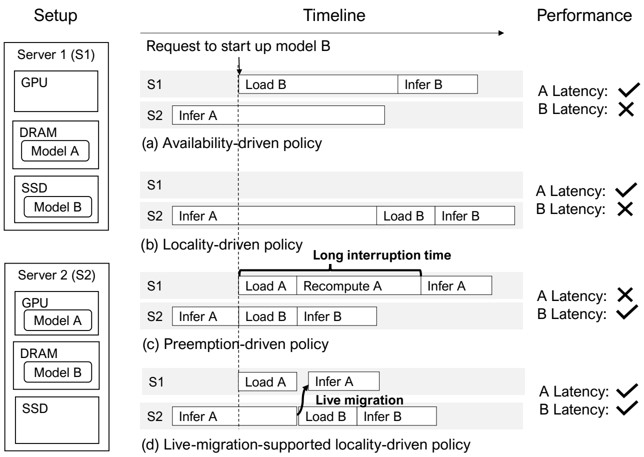


Figure 3: Analysis of different locality-driven policies

5 Efficient Live Migration of LLM Inference

In this section, we describe why live migration is the key to effective locality-driven LLM inference, and how to make such a live migration process particularly efficient.

5.1 Need for Live Migration

We consider a simple example to analyze the performance of different current approaches in supporting the checkpoint locality. In this example, we have two servers (named Server 1 and Server 2) and two models (named Model A and Model B), as illustrated in Figure 3. Server 1 currently has Model A in DRAM and Model B in SSD and its GPU is idle, while Server 2 currently has Model B in DRAM, and its GPU is running the inference of Model A.

In Figure 3, we analyze the performance of potential policies for starting up Model B. Our analysis is based on their impact on the latency performance of both Model A and B:

- *Availability-driven policy* chooses Server 1 currently with an available GPU, and it is agnostic to the location of Model B. As a result, the Model B’s startup latency suffers while the Model A remains unaffected.
- *Locality-driven policy* opts for the locality in choosing the server and thus launching Model B on Server 2. However, it waits for Model A to complete, making Model B suffer from a long queuing delay. Furthermore, the locality policy leaves Server 1 under-utilized, preventing all servers from being fully utilized.

- *Preemption-driven policy* preempts Model A on Server 2 and startups Model B. It identifies that Server 1 is free and reinitiates Model A there. This policy reduces Model B's latency but results in significant downtime for Model A when it performs reloading and recomputation.
- *Live-migration-supported locality-driven policy* prioritizes locality without disrupting Model A. It initially preloads Model A on Server 1, maintaining inference operations. When Model A is set on Server 1, its intermediate state is transferred there, continuing the inference seamlessly. Following this, Model B commences on Server 2, taking advantage of locality. This policy optimizes latency for both Models A and B.

According to the examples above, live migration stands out in improving latency for both Model A and Model B among all locality-driven policies.

5.2 Making Live Migration Efficient

We aim to achieve efficient live migration of LLM inference, incurring minimal resource overhead and minimal user interruption. We initially considered using the snapshot method from Singularity [68], which involves snapshotting the LLM inference. However, this method is slow due to lengthy snapshot creation and transfer times (e.g., typically 10s seconds or even minutes). Dirty-page-based migration might be considered to accelerate virtual machine migration, but this approach is currently not supported in GPU-enabled containers and virtual machines. Hence, we decided to explore live migration methods that can be easily implemented in applications.

To make the live migration method effective for LLM inference, we aim to achieve two objectives: (i) the migrated inference state must be minimal to reduce network traffic, and (ii) the destination server must quickly synchronize with the source server's progress to minimize migration times.

For (i), we propose to migrate tokens (typically 10-100s KB) instead of the large KV-Cache (typically 1-10s GB), as recomputing the KV-Cache based on the migrated tokens on the destination GPU is generally much faster than transferring the dirty state over the network. In certain conditions (e.g., given high-bandwidth network and short input sequences), migrating KV-Cache might also be fast yet it still increases cluster network traffic compared to migrating tokens.

For (ii), we leverage an insight from LLM inference: recomputing the KV-Cache for current tokens on the destination GPU is significantly faster (usually an order of magnitude shorter) than generating an equivalent number of new tokens on the source GPU. This approach facilitates efficient convergence of multi-round token-based migration, with the quantity of tokens generated on the source diminishing with each round. For example, time to recompute the KV-Cache for 1000 tokens equals to the time to generate about 100 new tokens according to [70].

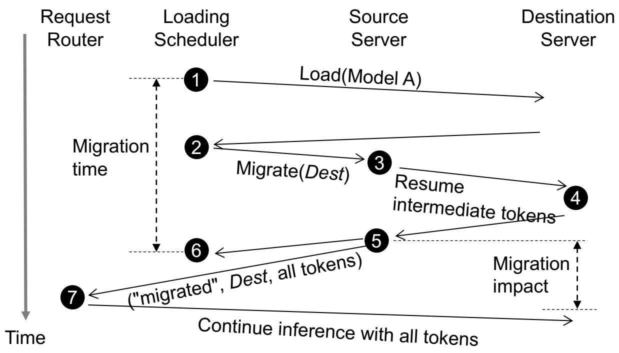


Figure 4: Live migration process for LLM inference

5.3 Multi-Round Live Migration Process

We implement the above proposal as a multi-round live migration process. In each migration round (step ❸, ❹ and ❺), the destination server (referred to as the *dest* server) recomputes the KV cache using the intermediate tokens sent by the source server (referred to as the *src* server). When the gap (i.e., the tokens generated after the last round) between the source server and the destination server is close enough, the *src* server stops generating and sends all tokens to the *dest* via the request router, ensuring minimal interruption on ongoing inference during migration. This migration process is depicted in Figure 4 with its steps defined below:

1. The model loading scheduler sends a model loading request to *dest* server to load model A into GPUs. If there is an idle instance of model A on *dest* server, the scheduler skips this step.
2. After loading, the scheduler sends a migration request carrying the address of *dest* server to *src* server.
3. Upon receiving a migrate request, *src* server sets itself as "migrating", sends a resume request with intermediate tokens (i.e., input tokens and the output tokens produced before step 3) to *dest* server if the inference is not completed. Otherwise, it immediately returns to the scheduler.
4. *dest* server recomputes KV cache given the tokens in the resume request.
5. Once *resume* request is done, *src* server stops inference, returns to the scheduler, and replies to the request router with all tokens (i.e., the intermediate tokens together with the remaining tokens produced between step 3 and step 5) and a flag "migrated".
6. The scheduler finishes the migration, unloads model A at *src* server and starts loading model B.
7. The request router checks the flag in the inference response. If it is "migrated", the request router replaces *src* server with *dest* server in its route table and sends all tokens to *dest* server to continue inference.

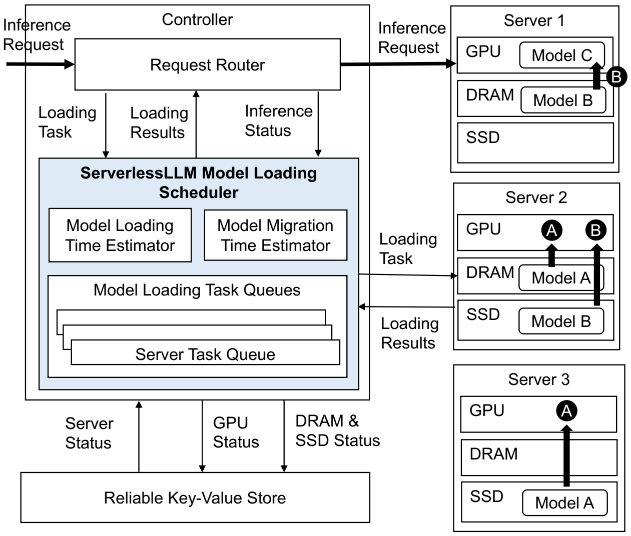


Figure 5: Overview of the model loading scheduler design

5.4 Practical Concerns

Handling inference completion. The autoregressive nature of LLM inference may lead to task completion at *src* server between steps ❸ and ❺. In such cases, *src* server informs the request router of the inference completion as usual. Additionally, it notifies the loading scheduler, which then instructs *dest* server to cease resuming, terminating the migration.

Handling server failures. ServerlessLLM can manage server failures during LLM inference migration. In scenarios where *src* server fails, if the failure happens during loading (i.e., before step ❷ in Figure 4), the scheduler aborts the migration and unloads the model from the destination. If the failure occurs during migration (i.e., between steps ❷ and ❸), the scheduler directs the destination to clear any resumed KV cache and unload the model.

In cases where *src* server fails, if the failure takes place during loading, the migration is canceled by the scheduler. Should the failure occur while resuming, the source notifies the scheduler of the failure and continues with the inference.

6 Startup-Time-Optimized Model Scheduling

In this section, we describe the design of the startup-time-optimized model scheduling implemented in ServerlessLLM's cluster scheduler (denoted as controller), as shown in Figure 5. This scheduler processes loading tasks from the request router and employs two key components: a model loading time estimator and a model migration time estimator. The former assesses loading times from various storage media, while the latter estimates times for necessary model migrations. For example, as shown in Figure 5, the scheduler calculates the time to load Model A (indicated by 🅐) from different servers' DRAM and SSD, aiding in server selection. Similarly, for Model B (🅑), it assesses whether to migrate Model C to another server or load Model B from Server 2's SSD.

To ensure robust time estimation, the ServerlessLLM scheduler employs distinct loading task queues for each server, effectively mitigating the impact of contentions caused by concurrent loading activities. Upon assigning a task, it promptly updates the server status—including GPU and DRAM/SSD states—in a reliable key-value store (e.g., , etcd [30] and ZooKeeper [38]). This mechanism enables ServerlessLLM to maintain continuity and recover efficiently from failures.

6.1 Estimating Model Loading Time

To estimate the time needed to load models from different storage tiers, we consider three primary factors: (i) *queuing time* (q), which is the wait time for a model in the server's loading task queue. This occurs when other models are pending load on the same server; (ii) *model size* (n), the size of the model in bytes, or its model partition in multi-GPU inference scenarios; (iii) *bandwidth* (b), the available speed for transferring the model from storage to GPUs. ServerlessLLM tracks bandwidth for network, SSD, and DRAM, allowing us to calculate loading time as $q + n/b$. Here, q accumulates from previous estimations for the models already in the queue.

For precise estimations, we have implemented: (i) Sequential model loading per server, with single I/O queues for both Remote-SSD and SSD-DRAM paths (since these paths are shared by multiple GPUs on a server), reducing bandwidth contention which complicates estimation; (ii) In multi-tier storage, ServerlessLLM uses the slowest bandwidth for estimation because of ServerlessLLM's pipeline loading design. For example, when SSD and DRAM are both involved, SSD bandwidth is the critical bottleneck since it is orders of magnitude slower than DRAM; (iii) The scheduler monitors the loading latency returned by the servers. It leverages the monitoring metrics to continuously improve its estimation of the bandwidth through different storage media.

6.2 Estimating Model Migration Time

For live migration time estimation, our focus is on model resuming time (as shown in step ❹ in Figure 4), as this is significantly slower (seconds) than token transfer over the network (milliseconds). We calculate model resuming time considering: (i) *input tokens* (t_{in}), the number of tokens in the LLM's input prompt; (ii) *output tokens* (t_{out}), the tokens generated so far; and (iii) *model-specific parameters* (a and b), which vary with each LLM's batch sizes and other factors, based on LLM system studies like vLLM [42]. With all the above factors, we can compute the model resuming time as $a \times (t_{in} + t_{out}) + b$.

However, obtaining real-time output tokens from servers for the scheduler can lead to bottlenecks due to excessive server interactions. To circumvent this, we developed a method where the scheduler queries the local request router for the inference status of a model, as illustrated in Figure 5. With the inference duration (d) and the average time to produce a token (t), we calculate $t_{out} = d/t$.

For selecting the optimal server for model migration, ServerlessLLM employs a dynamic programming approach to minimize migration time.

6.3 Practical Concerns

Selecting best servers. Utilizing our time estimations, ServerlessLLM evaluates all servers for loading the forthcoming model, selecting the one offering the lowest estimated startup time. The selection includes the server ID and GPU slots to assign. If no GPUs are available, even after considering migration, the loading task is held pending and retried once the request router informs the scheduler to release GPUs.

Handling scheduler failures. ServerlessLLM is built to withstand failures, utilizing a reliable key-value store to track server statuses. On receiving a server loading task, its GPU status is promptly updated in this store. Post server's confirmation of task completion, the scheduler updates the server's storage status in the store. Once recorded, the scheduler notifies the request router of the completion, enabling request routing to the server. In the event of a scheduler failure, recovery involves retrieving the latest server status from the key-value store and synchronizing it across all servers.

Scaling schedulers. The performance of the loading scheduler has been significantly enhanced by implementing asynchronous operations for server status reads, writes, and estimations. Current benchmarks demonstrate its capability to handle thousands of loading tasks per second on a standard server. Plans for its distributed scaling are earmarked for future development.

Resource fairness. ServerlessLLM treats all models with equal importance and it ensures migrations do not impact latency. While we currently adopt sequential model loading on the I/O path, exploring concurrent loading on servers with a fairness guarantee is planned for future work.

Estimator accuracy. Our estimator can continuously improve their estimation based on the monitored loading metrics returned by the servers. They offer sufficient accuracy for server selection, as shown in Section 7.

7 Evaluation

This section offers a comprehensive evaluation of ServerlessLLM, covering three key aspects: (i) assessing the performance of our loading-optimized checkpoints and model manager, (ii) examining the efficiency and overheads associated with live migration for LLM inference, and (iii) evaluating ServerlessLLM against a large-scale serverless workload, modelled on real-world serverless trace data.

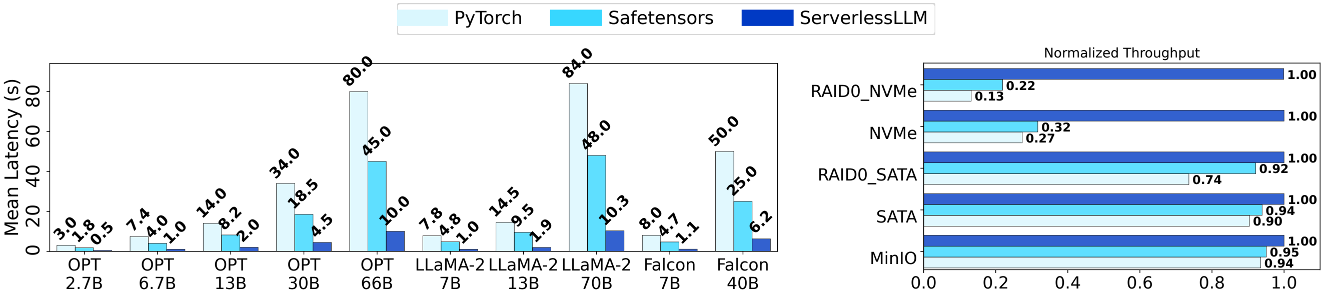
7.1 Evaluation Setup

Setup. We have two test beds: (i) a GPU server has 8 NVIDIA A5000 GPUs, 1TB DDR4 memory and 2 AMD EPYC 7453 CPUs, two PCIe 4.0-capable NVMe 4TB SSDs (in RAID 0) and two SATA 3.0 4TB SSDs (in RAID 0). This server is connected to a storage server via 1 Gbps networks on which we have deployed MinIO [51], an S3 compatible object store; (ii) a GPU cluster with 4 servers connected with 10 Gbps Ethernet connections. Each server has 4 A40 GPUs, 512 GB DDR4 memory, 2 Intel Xeon Silver 4314 CPUs and one PCIe 4.0 NVMe 2TB SSD.

Models. We use state-of-the-art LLMs, including OPT [91], LLaMA-2 [76] and Falcon [11] in different sizes. For cluster evaluation (§7.3 and §7.4) on test bed (ii), following prior work [44], we replicate OPT-6.7B/OPT-13B/OPT-30B models for 32/16/8 instances respectively (unless otherwise indicated) that are treated as different models during evaluation.

Datasets. We use real-world LLM datasets as the input to models. This includes GSM8K [25] that contains problems created by human problem writers, and ShareGPT [83] that contains multilanguage chat from GPT4. Since the models we used can handle at most 2048 context lengths, we truncate the input number of tokens to the max length. We also randomly sample 4K samples from each dataset to create a mixed workload, emulating real-world inference workloads.

Workloads. Since there are no publicly available LLM serverless inference workloads, we use Azure Serverless Trace [64] which is a representative serverless workload used in recent serverless studies [61] and model-serving studies [44, 90]. We designate functions to models and creates bursty request traces (CV=8 using Gamma distribution), following the workload generation method used in AlpaServe [44]. We then scale this trace to the desired requests per second (RPS). For cluster evaluation, we replicate each model based on its popularity and distribute them across nodes' SSDs using round-robin placement until the total cluster-wide storage limit is reached. Optimization of checkpoint placement is considered a separate issue and is not addressed in this paper. For all experiments (unless we indicate otherwise), we report the model startup latency, a critical metric for serverless inference scenarios. When migration or preemption is enabled, this latency is added with pause latency, accounting for the impacts of delays.



(a) Checkpoint loading speed.

(b) Normalized bandwidth utilization.

Figure 6: Checkpoint loading performance.

7.2 ServerlessLLM Checkpoint Loading

We now evaluate the model manager's effectiveness in reducing the model loading latency. For our experiments, we test the checkpoint read on test bed (i). We record reads from 20 copies of each model checkpoint to get a statistically significant performance report. We clear the page and inode caches after checkpoint copies are made to ensure a cold start. For each type of model, we randomly access the 20 copies to simulate real-world access patterns.

Loading performance. We aim to quantify the performance gains achieved by the ServerlessLLM checkpoint manager. We compare PyTorch [58] and Safetensors [36], representing the read-by-tensor checkpoint loading and mmap-based checkpoint loading, respectively. We use all types of models with all checkpoints in FP16 and run the test on RAID0-NVMe SSD having a throughput of 12 GB/s.

Figure 6a shows the performance comparison in terms of mean latency for all the models[2]. We observe that ServerlessLLM is 6X and 3.6X faster than PyTorch and Safetensors, respectively, for our smallest model (OPT-2.7B). We observe similar results with the largest model (LLaMA-2-70B) where ServerlessLLM is faster than PyTorch and Safetensors by 8.2X and 4.7X respectively. Safetensors is slower than ServerlessLLM due to a lot of page faults (112K for LLaMA-2-7B) on cold start. In contrast, ServerlessLLM's checkpoint manager leverages direct I/O and realizes chunk-based parallel loading, all contributing to the significant improvement in loading throughput. PyTorch is about 2X slower than Safetensors in our evaluation, consistent with the results in a public benchmark [37] reported by Safetensors. The primary reason is that PyTorch first copies data into host memory and then into GPU memory.

Furthermore, we observe that the loading performance of ServerlessLLM is agnostic to the type of the model. For example, the performance of both OPT-13B and LLaMA-2-13B is similar signifying the fact that the performance is only dependent on the checkpoint size.

Loading performance with LoRA adapters. ServerlessLLM also supports loading LoRA adapters [34] in PEFT format [49]. We conducted experiments using the same setting in [65]. For an adapter (rank=32, size=1GB) of LLaMA-70B model, ServerlessLLM achieves 83.5ms loading latency which is 4.4X faster than Safetensors whose loading latency is 370ms. This demonstrates ServerlessLLM's loader design efficiency in small checkpoint loading.

Harness full bandwidth of the storage devices. We now move to understand if ServerlessLLM can utilize the entire bandwidth that a storage medium offers to achieve low latency. We use the same setup as described above. We choose LLaMA-2-7B to represent the SOTA LLM model. We use FIO [17] with the configuration of asynchronous 4M direct sequential read with the depth of 32 as the optimal baseline and optimized throughput using the result in all storage media. We test various settings of FIO to make sure the configuration chosen has the highest bandwidth on each storage media. For object storage over the network, we use the official MinIO benchmark to get the maximum throughput.

Figure 6b shows the bandwidth utilization across different storage devices, normalized relative to the measurements obtained using FIO and MinIO. The storage device from bottom to top is ascending in maximum bandwidth. We observe that ServerlessLLM's model manager is capable of harnessing different storage mediums and saturating their entire bandwidth to get maximum performance. Interestingly, we observe that ServerlessLLM is well suited for faster storage devices such as RAID0-NVMe compared to Pytorch and Safetensors. It shows that existing mechanisms are not adaptive to newer and faster storage technology. Despite the loading process passing through the entire memory hierarchy, ServerlessLLM is capable of saturating the bandwidth highlighting the effectiveness of pipelining the loading process.

Performance breakdown. We now move to highlight how each optimization within the model manager contributes towards the overall performance. We run an experiment using RAID0-NVMe with various OPT models. We start from the basic implementation (ReadByTensor) and incrementally add

[2] The number after the model name represents the number of parameters in the figure and B stands for Billion.

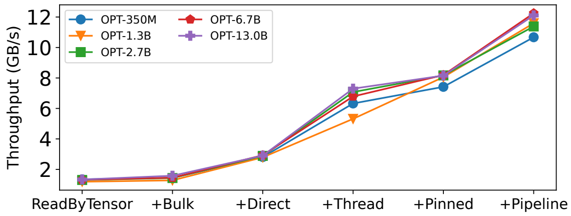


Figure 7: Performance breakdown of checkpoint loaders.

optimizations until the Pipeline implementation. Figure 7 shows the performance breakdown for each model. We observe similar contributions by different optimizations for all the models despite having different checkpoint size.

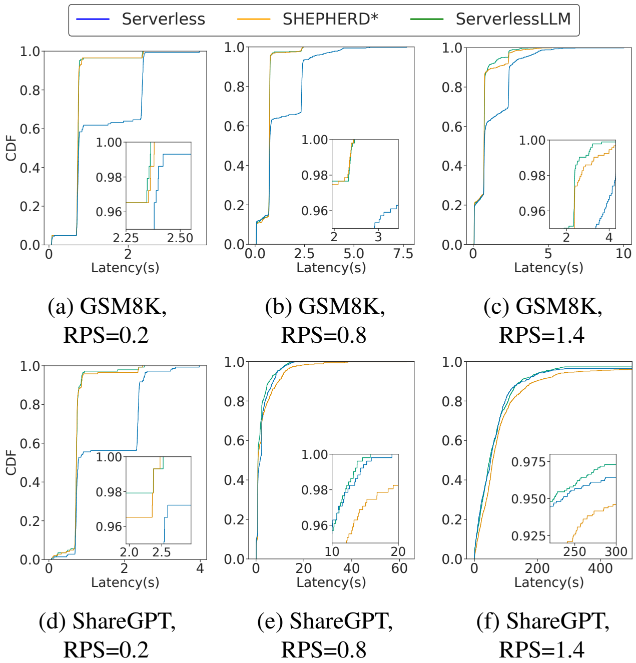
Bulk reading improves 1.2x throughput, mitigating the throughput degradation from reading small tensors one after another (on average one-third of the tensors in the model are less than 1MB). Direct IO improves 2.1x throughput, bypassing cache and data copy in the kernel. Multi-thread improves 2.3x throughput, as multiple channels within the SSD can be concurrently accessed. Pinned memory provides a further 1.4x throughput, bypassing the CPU with GPU DMA. Pipeline provides a final 1.5x improvement in throughput, helping to avoid synchronization for all data on each storage tier.

We run ServerlessLLM in a container to limit the CPU cores it can use. We find that with 4 CPU cores, ServerlessLLM can achieve maximum bandwidth utilization. We set a sufficiently large chunk size in bulk reading (16MB) to involve less number of reads and also pinned memory-based chunk pool does not need extra CPU cycles for data copy.

7.3 ServerlessLLM Model Scheduler

In this section, we evaluate the performance of the ServerlessLLM's cluster scheduler on test bed (ii). We compare ServerlessLLM against two schedulers – the de-facto serverless scheduler and Shepherd [90] scheduler. The serverless scheduler randomly chooses any GPU available and does not comprise any optimization for loading time. We implement Shepherd scheduler and use ServerlessLLM's loading time estimation strategy to identify the correct GPU. We call the modified scheduler as Shepherd*. Therefore, in principle, Shepherd* and ServerlessLLM will choose the same GPU. However, Shepherd* will continue to rely on preemption, while ServerlessLLM will rely on live migration to ensure lower latency times.

Figure 8a shows the result of a scenario where we run all three schedulers against OPT-6.7B model and GSM8K and ShareGPT dataset while increasing the requests per second. ShareGPT dataset's average inference time is 3.7X longer than GSM8K. Figure 8a and Figure 8d show the case where there is no locality contention for both datasets. The serverless scheduler cannot take advantage of locality-aware scheduling unlike ServerlessLLM and Shepherd* leading to longer latency. For 40% of the time, the model is loaded from SSD due to random allocation of the GPUs. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar.



(a) GSM8K, RPS=0.2

(b) GSM8K, RPS=0.8

(c) GSM8K, RPS=1.4

(d) ShareGPT, RPS=0.2

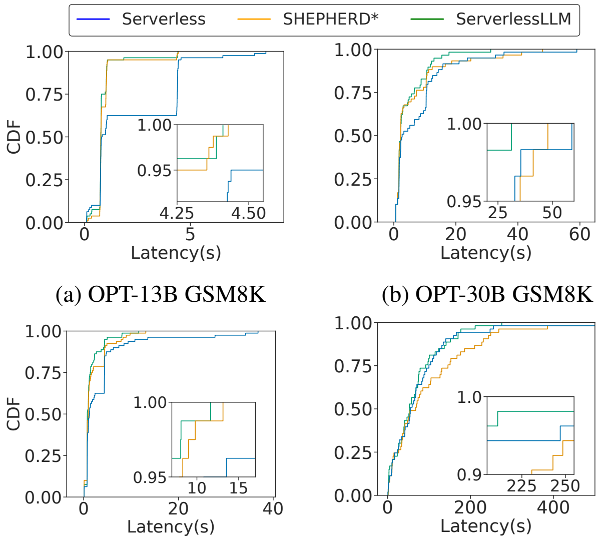
(e) ShareGPT, RPS=0.8

(f) ShareGPT, RPS=1.4

Figure 8: Impacts of RPS on model loading schedulers.

When the schedulers are subjected to medium requests per second, for GSM8K (Figure 8b, without locality-aware scheduling, the loading times start causing queueing latency leading with Serverless scheduler resulting in increasing the P99 latency by 1.86X. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar. With a longer inference time with ShareGPT (Figure 8e, we even observe 2X higher P99 latency with Shepherd* compared to ServerlessLLM due to preemption. As ServerlessLLM relies on live migration in case of locality contention, ServerlessLLM performs better than the other schedulers despite the number of migrations is higher (114 out of 513 total requests) than the number of preemptions (40 out of 513 total requests).

On further stressing the system by increasing the requests per second to 1.4, for GSM8K, one can clearly observe the impact of live migration and preemption. ServerlessLLM outperforms Shepherd* and Serverless schedulers by 1.27X and 1.95X on P99 latency respectively. There are 9 preemptions and 53 migrations respectively for a total of 925 requests. As discussed in Section 5.1, preemptions lead to longer latency compared to migrations. We also observe that with Shepherd*, model checkpoints are read from SSD 2X times more than with ServerlessLLM. With ShareGPT (figure 8f, we observe that the GPU occupancy reaches 100% leading to requests



(a) OPT-13B GSM8K

(b) OPT-30B GSM8K

(c) OPT-13B ShareGPT

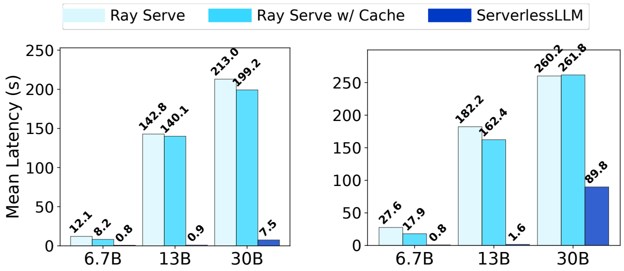
(d) OPT-30B ShareGPT

Figure 9: Impacts of datasets and models on model loading schedulers.

timeouts with all the three schedulers[3]. Shepherd behaves the worst compared to Serverless and ServerlessLLM schedulers, i.e., 1.43X and 1.5X higher P95 latency respectively. ServerlessLLM and Shepherd* issue 64 migrations and 166 preemptions, respectively for a total of 925 requests. In this scenario, ServerlessLLM's effectiveness is constrained by resource limitations.

We further stress the system by running even larger models (OPT-13B and OPT-30B) with GSM8K and ShareGPT datasets. Figure 9 shows the results for those experiments. locality-aware scheduling is more important for larger models as caching them in the main memory can reap better performance. As ServerlessLLM and Shepherd* are both locality-aware, they can make better decisions while scheduling the requests leading to better performance. As Serverless scheduler makes decisions randomly, for GSM8K, we observe that for 35-40% times, the model is loaded from SSD leading to poor performance. We see similar behavior for ShareGPT, OPT-13B experiment too. For the OPT-30B ShareGPT case, the model size is 66 GB. Hence, only two models can be stored in the main memory at any given time reducing the impact of locality-aware scheduling. Even in this extreme case, ServerlessLLM still achieves 35% and 45% lower P99 latency compared to Serverless and Shepherd* respectively.

Time Estimation. The GPU time estimation error is bounded at 5ms, while the SSD loading error is bounded at 40ms. However, we do observe instability in CUDA driver calls. For instance, when migrating a model, we noted that cleaning up GPU states (e.g., KV cache) using `torch.cuda.empty_cache()` can lead to inaccurate estimations, resulting in an average underestimation of 25.78 ms. While infrequent, we observed a maximum underestimation of 623 ms during GPU state cleanup in one out of 119 migrations (as depicted in Figure 8e).



(a) GSM8K

(b) ShareGPT

Figure 10: Impacts of datasets and models on overall serving systems.

7.4 Entire ServerlessLLM in Action

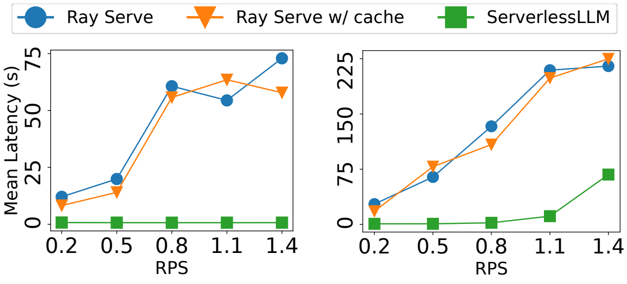
We aimed to deploy the entire ServerlessLLM with a serverless workload on test bed (ii). Here, we compare ServerlessLLM against state-of-the-art distributed model serving systems: (i) Ray Serve (Version 2.7.0), a version we have extended to support serverless inference scenarios with performance that can match SOTA serverless solutions such as KServe; (ii) Ray Serve with Cache, a version we improved to adopt a local SSD cache on each server (utilizing the LRU policy as in ServerlessLLM) to avoid costly model downloads; and (iii) KServe (Version 0.10.2), the SOTA serverless inference system designed for Kubernetes clusters.

For best performance, Ray Serve and its cache variant are both enhanced by storing model checkpoints on local SSDs and estimating download latency by assuming an exclusively occupied 10 Gbps network. For each system, we set the maximum concurrency to one and set the keep-alive period equal to its loading latency, following prior work [60]. We launch parallel LLM inference clients to generate various workloads, where each request has a timeout threshold of 300 seconds.

Effectiveness of loading-optimized checkpoints. We aimed to assess the effectiveness of loading-optimized checkpoints within a complete serverless workload, employing various model sizes and datasets to diversely test the checkpoint loaders.

In this experiment, as depicted in Figure 10, Ray Serve and Ray Serve with Cache utilize Safetensors. Owing to the large sizes of the models, the SSD cache cannot accommodate all models, necessitating some to be downloaded from the storage server. With OPT-6.7B and GSM 8K, ServerlessLLM starts models in an average of 0.8 seconds, whereas Ray Serve takes 12.1 seconds and Ray Serve with Cache 8.2 seconds, demonstrating an improvement of over 10X. Even

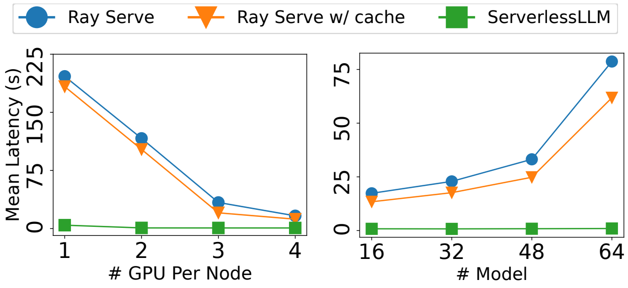
[3] Based on the average inference time of OPT-6.7B on ShareGPT dataset, the maximum theoretically RPS is 1.79.



(a) GSM8k

(b) ShareGPT

Figure 11: Impacts of RPS on overall serving systems.



(a) Impacts of # GPUs per node

(b) Impacts of # models

Figure 12: System scalability and resource efficiency.

with a faster network (i.e., 100 Gbps), the average latency of Ray Serve could drop to 3.8 seconds, making it still 4.7 times slower than ServerlessLLM. The significance of the model loader becomes more pronounced with larger models, as ServerlessLLM can utilize parallel PCIe links when loading large models partitioned on multiple GPUs from pinned memory pool. For instance, with OPT-30B, ServerlessLLM still initiates the model in 7.5 seconds, while Ray Serve's time escalates to 213 seconds and Ray Serve with Cache to 199.2 seconds, marking a 28X improvement.

This considerable difference in latency substantially affects the user experience in LLM services. Our observations indicate that ServerlessLLM can fulfill 89% of requests within a 300-second timeout with OPT-30B, whereas Ray Serve with Cache manages only 26%.

With the ShareGPT dataset (Figure 10b), which incurs a 3.7X longer inference time than GSM 8K, the challenge for model loaders becomes even more intense. For models like 6.7B and 13B, ServerlessLLM achieves latencies of 0.8 and 1.6 seconds on average, respectively, compared to Ray Serve and Ray Serve with Cache, which soar to 182.2 and 162.4 seconds. When utilizing OPT-30B, ServerlessLLM begins to confront GPU limitations (with all GPUs occupied and migration unable to free up more resources), leading to an increased latency of 89.9 seconds. However, this is still a significant improvement over Ray Serve with Cache, which reaches a latency of 261.8 seconds

Effectiveness of live migration and loading scheduler. In evaluating the effectiveness of LLM live migration and the loading scheduler, we created workloads with varying RPS levels. Scenarios with higher RPS highlight the importance of achieving load balancing and locality-aware scheduling since simply speeding up model loading is insufficient to address the resource contention common at large RPS levels.

From Figure 11a, it is evident that ServerlessLLM, equipped with GSM8K, consistently maintains low latency, approximately 1 second, even as RPS increases. In contrast, both Ray Serve and Ray Serve with Cache experience rising latency once the RPS exceeds 0.5, which can be attributed to GPU resource shortages. Their inability to migrate LLM inference for locality release or to achieve load balancing, unlike ServerlessLLM, results in performance degradation.

With the more demanding ShareGPT workload, as shown in Figure 11b, ServerlessLLM maintains significant performance improvements — up to 212 times better — over Ray Serve and Ray Serve with Cache across RPS ranging from 0.2 to 1.1. However, at an RPS of 1.4, ServerlessLLM's latency begins to rise, indicating that despite live migration and optimized server scheduling, the limited GPU resources eventually impact ServerlessLLM's performance.

Resource efficiency. A major advantage of the low model startup latency in ServerlessLLM is its contribution to resource savings when serving LLMs. We vary the number of GPUs available on each server to represent different levels of resource provisioning. As shown in Figure 12a, ServerlessLLM scales well with elastic resources. With just one GPU per server, ServerlessLLM already achieves a 4-second latency by efficient migrations and swaps. In contrast, Ray Serve with Cache requires at least four GPUs per server to attain a 12-second latency, which is still higher than ServerlessLLM's performance with only one GPU per node. With larger clusters, the resource-saving efficiency of ServerlessLLM is expected to become even more pronounced, as larger clusters offer more options for live migration and server scheduling.

The resource efficiency of ServerlessLLM is further evident when maintaining a fixed number of GPUs while increasing the number of LLMs in the cluster. In Figure 12b, with a limited number of models, Ray Serve with Cache can match ServerlessLLM in latency performance. However, as the number of models grows, the performance gap widens, showcasing ServerlessLLM's potential suitability for large-scale serverless platforms.

KServe comparison. In our study, we assess KServe and ServerlessLLM within a Kubernetes cluster. Given that our four-server cluster is unsuitable for a Kubernetes deployment, we instead utilize an eight-GPU server, simulating four nodes with two GPUs each. Since KServe performs slower than the other baselines considered in our evaluation, we only briefly mention KServe's results without delving into details.

With KServe, the GPU nodes initially exhibited a first token latency of 128 seconds. This latency was primarily due to KServe taking 114 seconds to download an OPT-6.7B model checkpoint from the local S3 storage over a 1 Gbps network. However, after applying the same enhancement as those for Ray Serve, we reduced the first token latency to 28 seconds. Despite this improvement, KServe's best latency was significantly higher than those achieved by ServerlessLLM. Notably, ServerlessLLM was the only system able to reduce the latency to within one second.

8 Related Work

Serverless inference systems. Extensive research has focused on optimizing ML model serving in serverless architectures, targeting batching [10, 84, 89], scheduling [60, 87], and resource efficiency [24, 43]. Industry solutions like AWS SageMaker and Azure ML [50], along with the open-source KServe [16], demonstrate practical implementations. Despite these advancements, serverless inference systems still perform suboptimally with LLMs, as our paper demonstrates.

Serverless cold-start optimizations. Cold-start latency is a significant issue in serverless systems, addressed through various strategies including fast image pulling [79], lightweight isolation [45, 53], snapshot and restore [15, 22, 28, 67, 77], resource pre-provision [64], elastic resource provisioning [48, 78], and fork [9, 80]. These approaches mainly focus on reducing startup times for containers or VMs without loading large external states. Recent research has explored optimizing cold-starts by facilitating faster model swaps between GPUs and host memory [39, 88], though scalability with LLMs is still challenging. In contrast, ServerlessLLM effectively minimizes cold-start latency through LLM-specific innovations, such as optimized checkpoint formats and loading pipelines, live migration, and a cluster scheduler tailored to LLM inference characteristics.

Exploiting locality in serverless systems. Locality plays a crucial role in various optimization strategies for serverless systems. This includes leveraging host memory and local storage for data cache [41, 59, 69], optimizing the reading of shared logs [40], and enhancing communication efficiency in serverless Directed Acyclic Graphs (DAGs) [46, 47]. ServerlessLLM, distinct from existing methods, introduces a high-performance checkpoint cache for GPUs, markedly improving checkpoint loading from multi-tier local storage to GPU memory. Recent studies [8, 86] have also recognized the need for leveraging locality in orchestrating serverless functions. Beyond these studies, ServerlessLLM leverages LLM-specific characteristics in improving the locality-based server's selection and launching locality-driven inference.

LLM serving systems. Recent advancements in LLM serving have improved inference latency and throughput. Orca [85] uses continuous batching for better GPU utilization during inference. AlpaServe [44] shows that model parallelism can enhance throughput while meeting SLO constraints, though it has yet to be tested on generative models. vLLM [42] introduces PagedAttention for efficient KV cache management. SplitWise [55] improves throughput by distributing prompt and token generation phases across different machines. Some approaches [14, 66] also use storage devices to offload parameters from GPUs to manage large LLM sizes. However, these systems often overlook model loading challenges, leading to increased first token latencies when multiple models share GPUs. ServerlessLLM addresses this by focusing on minimizing loading latency to complement these throughput and latency optimizations.

9 Conclusion

This paper describes ServerlessLLM, a low-latency serverless inference system purposefully designed for LLMs. The design of ServerlessLLM uncovers significant opportunities for system research, including designing new loading-optimized checkpoints, discovering the need to support live migration when conducting locality-driven LLM inference, and enabling a serverless cluster scheduler to be aware of the locality of checkpoints in a cluster when optimizing its model scheduling decision. We believe our work can be extended to ensure fairness of resources across the cluster and explore the possibility of smart checkpoint placement. We look forward to addressing these issues in the future. We consider ServerlessLLM as the first step towards unlocking the potential of serverless computing for LLMs. We will continue to develop the open-source version of ServerlessLLM. Given its versatility, we envision it as a platform to test new research ideas.

10 Acknowledgments

We sincerely thank our shepherd, Amar Phanishayee, and the OSDI reviewers for their insightful feedback, which helped improve the quality of this paper. We also extend our thanks to Boris Grot for his feedback. We acknowledge Zhaonan Zhang, Mingyu Dai, and Yusen Fei for their work in the early stages of this project. We are grateful for the GPU resources provided by the School of Informatics and the Edinburgh International Data Facility. This work is supported by UK EPSRC and gifts from Tencent.

References

[1] Amazon elasticache pricing. https://aws.amazon.com/elasticache/pricing/?nc=sn&loc=5. Accessed: 2024-05-31.

[2] Custom LLMs. https://deepinfra.com/docs/advanced/custom_llms, 2024. Accessed on 2024-06-02.

[3] Generative AI tailored to you. https://fireworks.ai/, 2024. Accessed on 2024-06-02.

[4] The leading enterprise AI platform. https://cohere.com/, 2024. Accessed on 2024-06-02.

[5] Run AI with an API. https://replicate.com/, 2024. Accessed on 2024-06-02.

[6] Serverless endpoints for leading open-source models. https://www.together.ai/products#inference, 2024. Accessed on 2024-06-02.

[7] Your data. your AI. your future. https://www.databricks.com/, 2024. Accessed on 2024-06-02.

[8] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 365–380, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[11] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.

[12] Amazon. AWS S3. https://aws.amazon.com/s3/, 2023. Accessed on 2024-01-22.

[13] Amazon. Serverless inference - minimizing cold starts. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, 2023. Accessed on 2024-01-22.

[14] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[15] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[16] The KServe Authors. Kserve. https://github.com/kserve/kserve, 2023. Accessed on 2024-01-22.

[17] Jens Axboe. Flexible I/O Tester, 2022. Accessed on 2024-01-22.

[18] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[19] Anyscale Technical Blog. https://www.anyscale.com/blog/loading-llama-2-70b-20x-faster-with-anyscale-endpoints, 2023. Accessed on 2024-01-22.

[20] Banana.dev Technical Blog. https://www.banana.dev/blog/turboboot, 2023. Accessed on 2024-01-22.

[21] Microsoft Official Blog. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-a

i-powered-microsoft-bing-and-edge-your-copilot-for-the-web/, 2023. Accessed on 2024-01-22.

[22] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Andrew A. Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. Reducing the carbon impact of generative AI inference (today and in 2035). In *HotCarbon*, pages 11:1–11:7. ACM, 2023.

[24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[25] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

[26] GitHub Copilot. https://github.com/features/copilot, 2023. Accessed on 2024-01-22.

[27] DataBricks. LLM Inference Performance Engineering: Best Practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023. Accessed on 2024-01-22.

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[29] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23(1), jan 2022.

[30] Cloud Native Computing Foundation. etcd. https://etcd.io, 2023. Accessed on 2024-01-22.

[31] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {rdma}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[32] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022.

[33] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*, pages 443–462. USENIX Association, 2020.

[34] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[35] HuggingFace. https://huggingface.co, 2023. Accessed on 2024-01-22.

[36] HuggingFace. Safetensors: ML Safer for All. https://github.com/huggingface/safetensors, 2023. Accessed on 2024-01-22.

[37] HuggingFace. Speed Comparison. https://huggingface.co/docs/safetensors/speed, 2023. Accessed on 2024-01-22.

[38] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[39] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *EuroSys*, pages 249–265. ACM, 2023.

[40] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*, pages 611–626. ACM, 2023.

[43] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[44] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, pages 663–679. USENIX Association, 2023.

[45] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *USENIX Annual Technical Conference*, pages 53–68. USENIX Association, 2022.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[47] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[48] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. {KungFu}: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.

[49] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[50] Microsoft. Azure ML. https://learn.microsoft.com/en-us/azure/machine-learning, 2023. Accessed on 2024-01-22.

[51] MinIO. MinIO. https://min.io, 2023. Accessed on 2024-01-22.

[52] NVIDIA. NVIDIA DGX H100. https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx, 2023. Accessed on 2024-01-22.

[53] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[54] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/, 2022. Accessed on 2024-01-22.

[55] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting, 2023.

[56] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[57] PyTorch. PyTorch Documentation: Saving and Loading Models. https://pytorch.org/tutorials/beginner/saving_loading_models.html, 2023. Accessed on 2024-01-22.

[58] PyTorch. torch.load. https://pytorch.org/docs/stable/generated/torch.load.html, 2023. Accessed on 2024-01-22.

[59] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[61] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[62] ONNX Runtime. API Reference, onnxruntime.backend.prepare. https://onnxruntime.ai/docs/api/python/api_summary.html, 2023. Accessed on 2024-01-22.

[63] AWS SageMaker. Machine Learning Service - Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/, 2023.

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[65] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[66] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[67] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[68] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads, 2022.

[69] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[70] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative LLM serving, 2024.

[71] Mistral AI Team. Better, faster, stronger. Mistral AI News, April 2024. Accessed: 2024-05-31.

[72] The Mosaic Research Team. Introducing DBRX: A new state-of-the-art open LLM. *Mosaic AI Research*, March 2024. Accessed: 2024-05-31.

[73] The Ray Team. Ray serve. https://docs.ray.io/en/latest/serve/index.html, 2023. Accessed on 2024-01-22.

[74] TensorFlow. TensorFlow Documentation: Using the Saved Model Format. https://www.tensorflow.org/guide/saved_model, 2023. Accessed on 2024-01-22.

[75] TensorFlow. tf.saved_model.load. https://www.tensorflow.org/api_docs/python/tf/saved_model/load, 2023. Accessed on 2024-01-22.

[76] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[77] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572,

New York, NY, USA, 2021. Association for Computing Machinery.

[78] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[79] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[80] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[81] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[82] xAI. Grok-1, 2024. Accessed: 2024-05-31.

[83] Ming Xu. https://huggingface.co/datasets/shibing624/sharegpt_gpt4, 2023.

[84] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

[85] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[86] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[87] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148. IEEE, 2021.

[88] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *CoRR*, abs/2306.03622, 2023.

[89] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[90] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the Wild. In *NSDI*, pages 787–808. USENIX Association, 2023.

[91] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *CoRR*, abs/2205.01068, 2022.

[92] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management

Wonbeom Lee[†] Jungi Lee[†] Junghwan Seo Jaewoong Sim

Seoul National University

Abstract

Transformer-based large language models (LLMs) demonstrate impressive performance across various natural language processing tasks. Serving LLM inference for generating long contents, however, poses a challenge due to the enormous memory footprint of the transient state, known as the key-value (KV) cache, which scales with the sequence length and batch size. In this paper, we present InfiniGen, a novel KV cache management framework tailored for long-text generation, which synergistically works with modern offloading-based inference systems. InfiniGen leverages the key insight that a few important tokens that are essential for computing the subsequent attention layer in the Transformer can be speculated by performing a minimal rehearsal with the inputs of the current layer and part of the query weight and key cache of the subsequent layer. This allows us to prefetch only the essential KV cache entries (without fetching them all), thereby mitigating the fetch overhead from the host memory in offloading-based LLM serving systems. Our evaluation on several representative LLMs shows that InfiniGen improves the overall performance of a modern offloading-based system by up to 3.00× compared to prior KV cache management methods while offering substantially better model accuracy.

1 Introduction

Large language models (LLMs) have opened a new era across a wide range of real-world applications such as chatbots [40, 76], coding assistants [11, 43], language translations [1, 68], and document summarization [64, 74]. The remarkable success of LLMs can largely be attributed to the enormous model size, which enables effective processing and generation of long contents. For instance, while the maximum sequence length of the first version of GPT was restricted to 512 tokens [51], the latest version, GPT-4, can handle up to 32K tokens, which is equivalent to approximately 50 pages of text [3]. Some recently announced models such as Claude 3 [6] and Gemini 1.5 [53] can even process up to 1 million tokens, significantly expanding the context window by several orders of magnitude.

In addition to the well-studied challenge of the model size, deploying LLMs now encounters a new challenge due to the substantial footprint of the transient state, referred to as the *key-value (KV) cache*, during long context processing and generation. For generative LLM inference, the keys and values of all preceding tokens are *stored* in memory to avoid redundant and repeated computation. Unlike the model weights, however, the KV cache scales with the output sequence length, often consuming even more memory capacity than the model weights. As the demand for longer sequence lengths (along with larger batch sizes) continues to grow, the issue of the KV cache size will become more pronounced in the future.

Meanwhile, modern LLM serving systems support offloading data to the CPU memory to efficiently serve LLMs within the hardware budget [5, 57]. These offloading-based inference systems begin to support even offloading the KV cache to the CPU memory, thereby allowing users to generate much longer contexts beyond the GPU memory capacity. However, transferring the massive size of the KV cache from the CPU memory to the GPU becomes a new performance bottleneck in LLM inference.

In this work, we propose InfiniGen, a KV cache management framework designed to synergistically work with modern offloading-based inference systems. InfiniGen builds on two key design principles. First, it speculates and chooses the KV cache entries that are critical to produce the next output token, dropping the non-critical ones, by conducting a minimal *rehearsal* of attention computation for Layer i at Layer $i-1$. Second, it leverages the CPU memory capacity and maintains the KV cache pool on the CPU, rather than on the GPU, to ensure that the critical KV cache values can be identified for all outputs and layers with a large *window size* while alleviating the concerns about limited GPU memory capacity for long content generation.

In particular, InfiniGen manipulates the model weights *offline* to make the speculation far more efficient and precise, by

[†] Equal contribution

skewing the Transformer architecture query and key matrices to emphasize certain important columns. During the prefill stage, while the prompt and input of an inference request are initially processed, InfiniGen generates *partial* weights for use in the subsequent decoding (i.e., output generation) stage. At Layer $i-1$ of the decoding stage, InfiniGen speculates on the attention pattern of the next layer (Layer i) using the attention input of Layer $i-1$, a partial query weight, and a partial key cache of Layer i. Based on the speculated attention pattern, InfiniGen prefetches the essential KV cache entries from the CPU memory for attention computation at Layer i. By dynamically adjusting the number of KV entries to prefetch, InfiniGen brings only the necessary amount of the KV cache to the GPU, thereby greatly reducing the overhead of the KV cache transfer. In addition, InfiniGen manages the KV cache pool by dynamically removing the KV cache entries of infrequently used tokens.

We implement InfiniGen on a modern offloading-based inference system [57] and evaluate it on two representative LLMs with varying model sizes, batch sizes, and sequence lengths. Our evaluation shows that InfiniGen achieves up to a 3.00× speedup over the existing KV cache management methods while offering up to a 32.6 percentage point increase in accuracy. In addition, InfiniGen consistently provides performance improvements with larger models, longer sequence lengths, and larger batch sizes, while prior compression-based methods lead to saturating speedups.

In summary, this paper makes the following contributions:

- We present InfiniGen, a dynamic KV cache management framework that synergistically works with modern offloading-based LLM serving systems by intelligently managing the KV cache in the CPU memory.

- We propose a novel KV cache prefetching technique with ephemeral pruning, which speculates on the attention pattern of the subsequent attention layer and brings only the essential portion of the KV cache to the GPU while retaining the rest in the CPU memory.

- We implement InfiniGen on a modern offloading-based inference system and demonstrate that it greatly outperforms the existing KV cache management methods, achieving up to 3.00× faster performance while also providing better model accuracy.

2 Background

This section briefly explains the operational flow and the KV caching technique of large language models and introduces the singular value decomposition (SVD) as a method of skewing matrices for a better understanding of our proposed framework, which we discuss in Section 4.

2.1 Large Language Models

Large language models (LLMs) are composed of a stack of Transformer blocks, each of which contains an attention layer followed by a feed-forward layer [61]. The input tensor (X) of the Transformer block has a dimension of $N \times D$, where N is the number of query tokens, and D is the model dimension. This input tensor (X) is first layer-normalized (LayerNorm), and the layer-normalized tensor (X_a) is fed into the attention layer as input. The attention input (X_a) is multiplied by three different *weight* matrices (W_Q, W_K, W_V) to generate Query (Q), Key (K), and Value (V) matrices. Each weight matrix has a dimension of $D \times D$. Thus, Query, Key, and Value have a dimension of $N \times D$. These matrices are reshaped to have a dimension of $H \times N \times d$, where H is the number of attention heads and d is the head dimension; note that $D = H \times d$.

Each head individually performs attention computation, which can be formulated as follows: $\mathrm{softmax}(QK^T)V$.[1] The attention output, after a residual add (adding to the input tensor X) and layer normalization, is fed into the feed-forward layer. The feed-forward network (FFN) consists of two consecutive linear projections and a non-linear activation operation between them. The output of FFN after a residual add becomes the output of a Transformer block, which has the *same* dimensionality as the input of the Transformer block (i.e., $N \times D$). This allows us to easily scale LLMs by adjusting the number of Transformer blocks.

2.2 Generative Inference and KV Caching

Generative LLM inference normally involves two key stages: the *prefill* stage and the *decoding* stage. In the prefill stage, LLMs summarize the context of the input sequence (i.e., input prompt) and produce a *new* token that serves as the initial input for the decoding stage. Subsequently, using this new token, LLMs run the decoding stage to generate the next token. The newly generated token is then fed back into the decoding stage as input, creating an *autoregressive* process for token generation. In this work, we refer to each token generation in the decoding stage as an *iteration*.

To generate a new token that aligns well with the context, LLMs need to compute the relationship between the last token and all the previous ones, including the tokens from the input sequence, in the attention layer. A naïve approach to this is to recompute the keys and values of all the previous tokens at every iteration. However, this incurs a significant overhead due to redundant and repeated computation. Furthermore, the computation overhead linearly grows with the number of the previous tokens; i.e., the overhead becomes larger for longer sequences.

To avoid such overhead, the keys (K) and values (V) of all the previous tokens are typically *memoized* in memory,

[1] In this work, we refer to the results of QK^T and $\mathrm{softmax}(QK^T)$ as *attention scores* and *attention weights*, respectively.

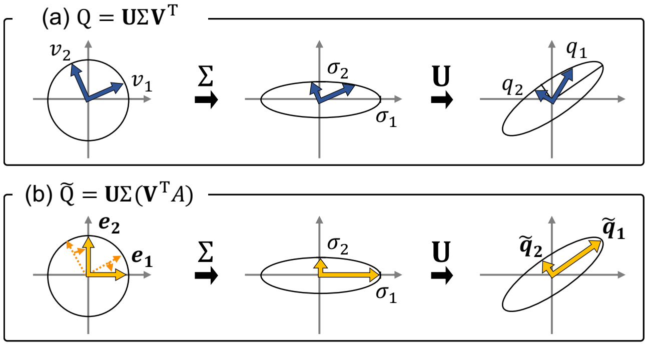


Figure 1: Transformation from matrix $\mathbf{V}^T$ to matrix Q in terms of SVD. The orthogonal matrix A maximizes the difference in magnitude between the column vectors of Q.

which is known as the *KV cache*. The KV cache then keeps updated with the key and value of the generated token at each iteration. As such, the dimension of the KV cache at the i-th iteration can be expressed as $H \times (N+i) \times d$. If batched inference is employed, the size of the KV cache also grows linearly to the batch size. By employing the KV cache, we can avoid repeated computation and produce the key and value of only one token at each iteration. Note that in the decoding stage, the input to the Transformer block (X) has a dimension of $1 \times D$, and the dimension of the attention score matrix becomes $H \times 1 \times (N+i)$ at the i-th iteration.

2.3 Outliers in Large Language Models

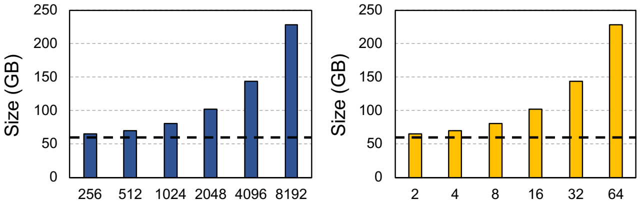
Large language models have outliers in the Transformer block input tensors. The outliers refer to the elements with substantially larger magnitudes than the other elements. The outliers in LLMs appear in a few fixed channels (i.e., columns in a 2D matrix) across the layers. Prior work has shown that outliers are due to the intrinsic property of the model (e.g., large magnitudes in a few fixed channels of layer normalization weights) [19, 65].

2.4 Singular Value Decomposition

We observe that skewing the query and key matrices to make a small number of channels much larger than others and using only those channels to compute the attention score matrix can effectively predict which tokens are important. In essence, we multiply the Q and K matrices with an orthogonal matrix A to make it align with the direction that Q stretches the most, to produce the respective skewed matrices $\tilde{Q}$ and $\tilde{K}$. We explain in detail why we use an orthogonal matrix in Section 4.2.

To find such an orthogonal matrix A, we employ the singular value decomposition (SVD), which is a widely used matrix factorization technique in linear algebra. For a real matrix Q of size $m \times n$, its SVD factorization can be expressed as follows:

$$Q = \mathbf{U}\Sigma\mathbf{V}^T,$$



(a) Sequence Length

(b) Batch Size

Figure 2: Total size of the KV cache and model weights of OPT-30B for different sequence lengths and batch sizes. The batch size of (a) is 16, and the sequence length of (b) is 2048. The dotted line represents the size of the model weights.

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of size $m \times m$ and $n \times n$, respectively.[2] Σ is an $m \times n$ diagonal matrix, which has nonzero values $(\sigma_1, \sigma_2, ..., \sigma_k)$ on the diagonal, where $k = \min(m,n)$. In terms of linear transformation, it is well known that a transformation of a vector $v \in \mathbb{R}^n$ by a real matrix B (i.e., the product of B and v) is a rotation and/or reflection in $\mathbb{R}^n$ if the B matrix is orthogonal. If B is an $m \times n$ diagonal matrix, each dimension of v is stretched by the corresponding diagonal entry of B and is projected to $\mathbb{R}^m$.

For example, Figure 1 shows how the column vectors v_1 and v_2 of $\mathbf{V}^T$ would transform to column vectors q_1 and q_2 of Q, when m and n are 2. In Figure 1(a), the orthogonal unit vectors v_1 and v_2 are first stretched to the points on an ellipse whose semi-axis lengths correspond to the diagonal entries in Σ. The vectors are then rotated and/or reflected to q_1 and q_2 by matrix $\mathbf{U}$. On the other hand, Figure 1(b) shows how orthogonal matrix A performs rotation to make the resulting $\tilde{q}_1$ much larger than $\tilde{q}_2$. Specifically, A rotates vectors v_1 and v_2 to e_1 and e_2, which map to the semi-axes of the ellipse. In this way, the vectors are stretched to the maximum and minimum by the matrix Σ. This process emphasizes the magnitude of $\tilde{q}_1$ over $\tilde{q}_2$, which allows us to effectively predict the attention score using only $\tilde{q}_1$ while omitting $\tilde{q}_2$.

3 Motivation

In this section, we first explain that the KV cache size becomes a critical issue for long-text generation in LLM inference, and it becomes more problematic when deploying modern offloading-based inference systems (Section 3.1). We then discuss why the existing KV cache management methods cannot fundamentally address the problem in the offloading-based inference system (Section 3.2).

3.1 KV Cache in LLM Inference Systems

As discussed in Section 2.2, today's LLM serving systems exploit KV caching to avoid redundant computation of key and

[2] Note that this $\mathbf{V}$, typeset with a different font, is one of the resulting matrices of SVD and is distinct from the V of the Value matrix in the Transformer attention layer.

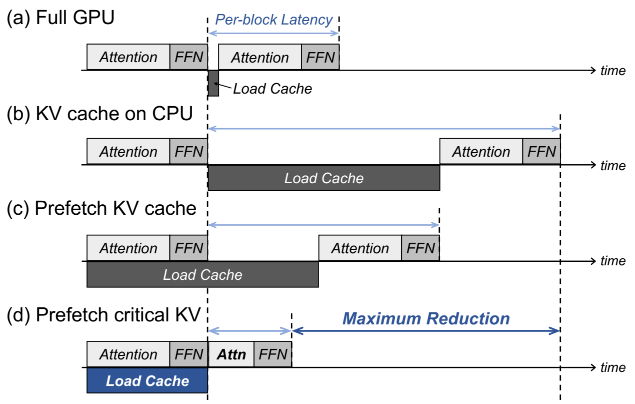


Figure 3: Comparison between different execution styles of Transformer blocks.

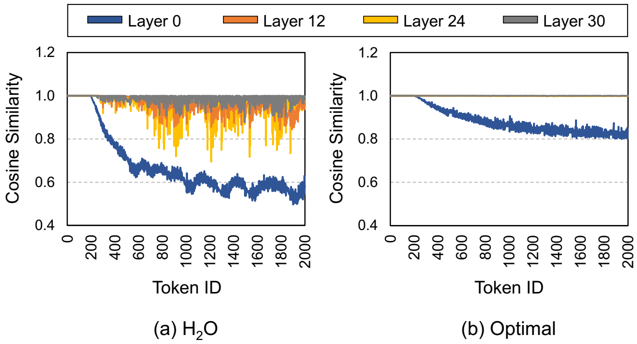


Figure 4: Cosine similarity between the attention weights of the base model with full cache and (a) H_2O or (b) Optimal. H_2O and Optimal use 200 tokens for attention computation. We use OPT-6.7B and a random sentence with 2000 tokens from the PG-19 dataset [52].

value projections during the decoding stage. While this is an effective solution for short sequence generation with a single client request, the KV cache quickly becomes a key memory consumer when we generate long sequences or employ modern request batching techniques [57, 71].

Figure 2 shows the combined size of LLM weights and the KV cache across different sequence lengths and batch sizes. As depicted in the figure, the model size remains constant regardless of sequence lengths or batch sizes, whereas the KV cache size linearly scales with them. Note that modern LLM serving systems, such as NVIDIA Triton Inference Server [45] and TensorFlow Serving [47], already support *batched* inference for better compute utilization and higher throughput in serving client requests. When individual requests are batched, each request retains its own KV cache, thereby increasing the overall KV cache size for the inference. Even for a single client request, beam search [59] and parallel sampling [20] are widely used to generate better outputs or to offer clients a selection of candidates [11, 24]. The techniques also increase the size of the KV cache like batched inference as multiple sequences are processed together. Consequently, the KV cache size can easily exceed the model size for many real-world use cases, as also observed in prior work [37, 49, 57, 78]. This can put substantial pressure on GPU memory capacity, which is relatively scarce and expensive.

LLM Inference Systems with Offloading. Modern LLM serving systems such as DeepSpeed [5] and FlexGen [57] already support offloading the model weights or the KV cache to the CPU memory. When it comes to offloading-based inference systems, the KV cache size becomes more problematic due to the low PCIe bandwidth between the CPU and GPU, which becomes a new and critical bottleneck.

Figure 3 depicts a high-level timing diagram between different execution styles of Transformer blocks. Figure 3(a) represents the case when the KV cache entirely resides in the GPU memory (Full GPU). In this case, the load latency of the KV cache (Load Cache) involves a simple read operation from the GPU memory, which is negligible due to the high bandwidth of GPU memory. However, the maximum batch size or sequence length is limited by the GPU memory capacity, which is relatively smaller than the CPU memory.

To enable a larger batch size or a longer sequence length, we can offload the KV cache to CPU memory (KV cache on CPU), as shown in Figure 3(b). While offloading-based inference systems alleviate the limitation on the batch size and sequence length, transferring hundreds of gigabytes of the KV cache to the GPU for attention computation significantly increases the overall execution time of Transformer blocks due to the limited PCIe bandwidth.

Even when we apply a conventional prefetching technique (Prefetch KV cache), as shown in Figure 3(c), only part of the load latency can be hidden by the computation of the preceding Transformer block. Note that although compressing the KV cache via quantization could potentially reduce the data transfer overhead in offloading-based systems [57], it does not serve as a fundamental solution as quantization does not address the root cause of the KV cache problem, which is the linear scaling of KV entries with the sequence length. This necessitates intelligent KV cache management to mitigate the performance overhead while preserving its benefits.

3.2 Challenges in KV Cache Management

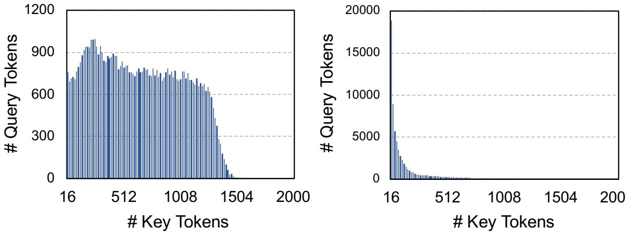
The fundamental approach to mitigating the transfer overhead of the KV cache from the CPU to GPU is to reduce the volume of the KV cache to load by identifying the *critical* keys and values for computing attention scores, as shown in Figure 3(d). It is widely recognized that the keys and values of certain tokens are more important than others in attention computation [9, 10, 14, 33, 63]. As explained in Section 2.1, after computing the attention score, the softmax operation is applied, which emphasizes a few large values of tokens. Therefore, skipping attention computation for some less critical tokens does not significantly degrade the model accuracy, provided the token selection is appropriate.

In this context, several recent works propose to reduce the KV cache size through key/value evictions at runtime within a constrained KV cache budget [37, 78]. However, all the prior works assume the *persistence* of attention patterns across iterations; that is, if a token is deemed unimportant in the current iteration (i.e., having a low attention weight), it is likely to remain unimportant in the generation of future tokens. Under the assumption, they evict the tokens with a low attention weight from the KV cache at each iteration when the KV cache size exceeds its budget. The keys and values of the evicted tokens are *permanently* excluded from the subsequent iterations while being removed from the memory. Although the recent works on managing the KV cache can be applied to offloading-based inference systems, we observe that they do not effectively address the challenges in KV cache management below and thus have subpar performance with offloading-based inference systems.

C1: Dynamic nature of attention patterns across iterations. Figure 4 shows the cosine similarity between the attention weights of the baseline model, which uses the KV cache of all prior tokens for computing attention weights (i.e., a maximum of 2000 tokens in the experiment), and two different KV cache management methods (H_2O and Optimal) with a KV cache budget of 200 tokens.[3] H_2O [78] is a state-of-the-art technique that retains only a small percentage of important tokens in the KV cache to reduce its size. It *assesses* the importance of each token in every iteration and *removes* unimportant ones before the next iteration to keep the KV cache size in check (i.e., using a narrow assessment window). In contrast, Optimal represents the scenario where we choose the same number of tokens as H_2O from the KV cache at each iteration but retain all prior keys and values (i.e., using a wider assessment window). In other words, Optimal selects 200 tokens out of the entire sequence of previous tokens at each iteration.

The figure indicates that despite H_2O-like approaches assuming that the attention pattern does not change across iterations, this is not the case in practice. The tokens deemed unimportant in the current iteration could become important in subsequent iterations. Consequently, H_2O exhibits high similarity until around 200 iterations (i.e., within the KV cache budget), but as the sequence length extends beyond the KV cache budget, it starts to struggle with the dynamic nature of the attention pattern, resulting in lower cosine similarity than the Optimal case. Note that while we only show the scenario of a KV cache budget of 200 out of a total sequence length of 2000 tokens for brevity, this issue would become more pronounced as the sequence length surpasses it.

C2: Adjusting the number of KV entries across layers. Figure 4 also illustrates that the impact of the KV cache eviction *varies* across the layers in LLMs. For Layer 0, both H_2O and Optimal show a significant drop in cosine similarity as the token ID increases. This implies that Layer 0 has a *broader* attending pattern than other layers; i.e., the attention weights are relatively similar between key tokens. Thus, the selected 200 tokens with the large attention weight do *not* adequately represent the attention pattern of the baseline model for this layer, as they are likely only slightly larger than the others, not strongly so. In such cases, it becomes necessary to compute the attention weight with a larger number of tokens.



(a) Layer 0

(b) Layer 18

Figure 5: Histogram that shows the number of key tokens needed to achieve 0.9 out of 1.0 total attention weight for (a) Layer 0 and (b) Layer 18 of the OPT-6.7B model. The bin width is set to 16. We observe that the distribution dynamically changes across the layers.

To estimate how many keys/values from the KV cache need to be retained, we sort the attention weight for each query token in descending order and sum the key tokens until the cumulative weight reaches 0.9. Figure 5 presents a histogram of the number of query tokens (y-axis) requiring the number of key tokens (x-axis) needed to reach a weight of 0.9 (out of the total attention weight of 1.0) in two different layers: Layer 0 and Layer 18. Layer 0 shows a broad distribution, indicating a significant variation in the number of key tokens required to achieve a weight of 0.9 for each query token. In contrast, Layer 18 exhibits a highly skewed distribution, suggesting that the majority of the query tokens in this layer require only a few key tokens to reach a weight of 0.9. This implies that we need to *dynamically* adjust the number of key tokens participating in attention computation across different layers to make efficient use of the KV cache budget.

C3: Adjusting the number of KV entries across queries. H_2O sets the number of key/value tokens to retain as a fixed percentage of the input sequence length. The KV cache budget remains constant regardless of how many tokens have been generated. By analyzing the data from Figure 5 on Layer 18, we observe that this fixed KV cache budget has some limitations. For instance, with an input sequence length of 200 and a 20% KV cache budget, H_2O maintains 40 key/value tokens throughout token generations. However, most of the subsequent query tokens require more than 40 tokens to effectively represent the attention weight of the baseline model; for example, the 500^{th}, 1000^{th}, 1500^{th}, and 2000^{th} tokens need 80, 146, 160, and 164 key tokens, respectively, to reach a cumulative attention weight of 0.9. This implies an inadequate amount of the key/value tokens to properly represent the attention weight of the baseline model. Furthermore, the

[3] The cosine similarity measures how much each row of the attention weight is similar to the case of the full KV cache. If they are similar, the generated tokens will also be similar. Thus, a low cosine similarity indicates low accuracy far from the baseline model with the full KV cache.

number of key tokens required to reach 0.9 varies even for the adjacent query tokens; for instance, the 998th, 999th, 1000th, 1001st, and 1002nd tokens need 172, 164, 146, 154, and 140 key tokens, respectively. Fixing the KV cache budget without accounting for the variance between query tokens inevitably results in ineffective KV cache management. Therefore, we need to *dynamically* adjust the amount of the key/value tokens loaded and computed for each query token to efficiently manage the KV cache.

Summary. Prior works aiming to reduce the KV cache size through token eviction inherently have some challenges. Given the dynamic attention pattern across iterations, permanently excluding evicted tokens from future token generation can result in a non-negligible drop in accuracy. Instead, we need to dynamically *select* critical tokens from the KV cache while avoiding the outright eviction of less important ones. Furthermore, the fixed size of the KV cache budget in prior works leads to inefficient KV cache management. The number of key/value tokens required for each layer differs, and each query token demands a varying number of key/value tokens to effectively represent the attention pattern of the baseline model. Failing to account for these variations may result in ineffective KV cache management. Thus, we need to dynamically adjust the number of key/value tokens to select from the KV cache while considering the variances between layers and query tokens.

4 InfiniGen Design

In this section, we present InfiniGen, a KV cache management framework for offloading-based inference systems. We first show the high-level overview of our proposed KV cache management solution (Section 4.1) and discuss the opportunities of KV cache prefetching that we observe (Section 4.2). We then explain our prefetching module (Section 4.3), which builds on the offloading-based inference systems, and discuss how InfiniGen manages the KV cache on CPU memory regarding the memory pressure (Section 4.4).

4.1 Overview

Figure 6 shows an overview of our KV cache management framework, InfiniGen, which enables offloading the KV cache with low data transfer overhead. The key design principle behind InfiniGen is to exploit the abundant CPU memory capacity to increase the *window size* when identifying the important tokens in the KV cache. As such, the majority of the tokens for the KV cache are kept in the CPU memory as we generate new tokens, not completely discarding them unlike prior works [37, 78]. However, we do not bring the entire KV cache to the GPU for attention computation, but load and compute with only the keys and values of a *few important tokens*, dropping other unimportant ones dynamically. To do so, we maintain the KV cache pool in the CPU memory and selectively and speculatively load a few of tokens.

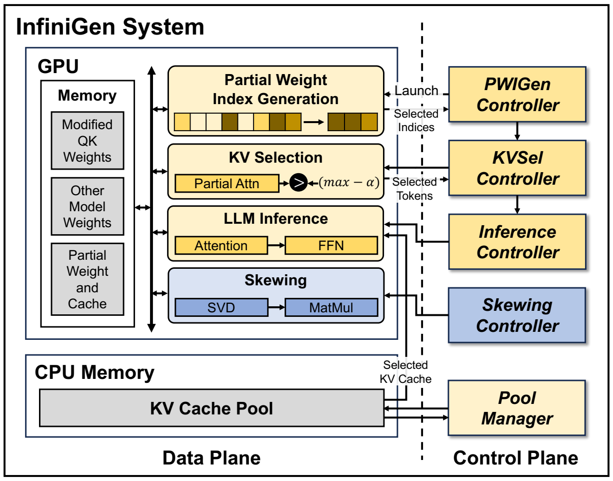


Figure 6: Overview of InfiniGen design.

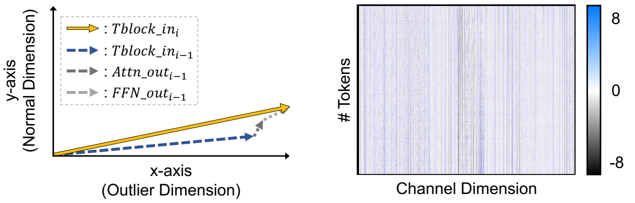
In detail, we use the attention input of the *previous* Transformer layer to speculate and prefetch the keys and values of the important tokens for the *current* layer. The speculation is done by performing a minimal rehearsal of attention computation of the current layer in the preceding layer. This allows for reducing the waste of PCIe bandwidth by only transferring the keys and values critical for attention computation while preserving model accuracy. In addition, although the KV cache is offloaded to CPU memory, which is much cheaper and larger than GPU memory, we manage the KV cache pool size so as not to put too much pressure on CPU memory.

As shown in Figure 6, there are two major components in the InfiniGen runtime. The first includes the Partial Weight Index Generation Controller, KV Selection Controller, and Inference Controller. These controllers cooperate to speculate and prefetch the critical KV cache entries while serving LLM inference. Additionally, to aid in prefetching, the Skewing Controller performs offline modifications on the model weights. We explain each operation in Section 4.3. The second component is the Pool Manager. It manages the KV cache pool on CPU memory under CPU memory pressure, which we discuss in Section 4.4.

4.2 Prefetching Opportunities

In the following, we first explain why using the attention input of the previous layer for speculation makes sense. We then show how we modify the query and key weight matrices to make our speculation far more effective.

Attention Input Similarity. Our prefetching module builds on the key observation that the attention inputs of consecutive attention layers are highly similar in LLMs. There are two major reasons behind this. The first is the existence of *outliers* in LLMs, as discussed in Section 2.3, and the second is due to layer normalization (LayerNorm).



(a) Input Similarity (b) Query

Figure 7: (a) Visualization of input similarity between consecutive Transformer blocks. (b) Query matrix of Layer 18 of the OPT-13B model. We only show channels from 3000 to 4000 for a clearer view of column-wise patterns.

To begin with, the input to the Transformer block i ($\texttt{Tblock\_in}_\texttt{i}$) can be formulated as follows:

$$\begin{aligned} Attn\_out_{i-1} &= Attn(LN(Tblock\_in_{i-1})) \\ FFN\_out_{i-1} &= FFN(LN(Tblock\_in_{i-1} + Attn\_out_{i-1})) \\ Tblock\_in_i &= Tblock\_in_{i-1} + Attn\_out_{i-1} + FFN\_out_{i-1}, \end{aligned} \tag{1}$$

where $\texttt{Tblock\_in}_{\texttt{i}-1}$ is an input for Layer $i-1$, which is first layer-normalized (*LN*) and is fed into the attention layer in the Transformer block. After performing attention, we obtain the output ($\texttt{Attn\_out}_{\texttt{i}-1}$), which is added to $\texttt{Tblock\_in}_{\texttt{i}-1}$ because of the residual connection. Then, the sum of $\texttt{Tblock\_in}_{\texttt{i}-1}$ and $\texttt{Attn\_out}_{\texttt{i}-1}$ is again layer-normalized and is fed into the FFN layer. Afterward, we obtain the FFN output ($\texttt{FFN\_out}_{\texttt{i}-1}$), which is added to the sum of $\texttt{Tblock\_in}_{\texttt{i}-1}$ and $\texttt{Attn\_out}_{\texttt{i}-1}$ again due to the residual connection. Finally, the sum of $\texttt{Tblock\_in}_{\texttt{i}-1}$, $\texttt{FFN\_out}_{\texttt{i}-1}$, and $\texttt{Attn\_out}_{\texttt{i}-1}$ is used as input to the next Transformer block ($\texttt{Tblock\_in}_\texttt{i}$).

Now, we show why the attention input of Layer i is similar to the one of Layer $i-1$ with the example in Figure 7(a). In the figure, there are four vectors, each of which corresponds to a term in Equation 1. The x-axis represents an outlier channel among the model dimension, while the y-axis represents a normal channel (i.e., other than the outlier channel). In practice, there exist more normal channels and only a few outlier channels in the input tensors, but we only present one channel each for both outlier and normal channels for clarity.

$\texttt{Tblock\_in}_{\texttt{i}-1}$ is highly skewed along the outlier channel (x-axis) due to a few outlier channels containing significantly large values compared to those in the normal channels. In contrast, $\texttt{Attn\_out}_{\texttt{i}-1}$ and $\texttt{FFN\_out}_{\texttt{i}-1}$ have relatively small values for both outlier and normal channels (i.e., short vectors). This is because the attention and FFN inputs are *layer-normalized*, reducing the magnitude of each value. The small magnitude of the attention and FFN inputs naturally results in their output values being relatively small compared to $\texttt{Tblock\_in}_{\texttt{i}-1}$. Consequently, $\texttt{Tblock\_in}_\texttt{i}$ is highly influenced by $\texttt{Tblock\_in}_{\texttt{i}-1}$, rather than $\texttt{Attn\_out}_{\texttt{i}-1}$ or $\texttt{FFN\_out}_{\texttt{i}-1}$. Highly similar inputs between consecutive Transformer blocks lead to similar inputs across the attention layers, as the attention input is a layer-normalized one of the Transformer block input.

Table 1: Average cosine similarity between the Transformer block input of Layer i ($\texttt{Tblock\_in}_\texttt{i}$) and the other three tensors ($\texttt{Tblock\_in}_{\texttt{i}-1}$, $\texttt{Attn\_out}_{\texttt{i}-1}$, $\texttt{FFN\_out}_{\texttt{i}-1}$) across the layers. We use a random sentence with 2000 tokens from the PG-19 dataset [52].

Tensors	OPT-6.7B	OPT-13B	OPT-30B	Llama-2-7B	Llama-2-13B
$\texttt{Tblock\_in}_{\texttt{i}-1}$	**0.95**	**0.96**	**0.97**	**0.89**	**0.91**
$\texttt{Attn\_out}_{\texttt{i}-1}$	0.29	0.28	0.36	0.31	0.27
$\texttt{FFN\_out}_{\texttt{i}-1}$	0.34	0.28	0.35	0.37	0.34

Table 1 shows the cosine similarity between $\texttt{Tblock\_in}_\texttt{i}$ and the other three tensors ($\texttt{Tblock\_in}_{\texttt{i}-1}$, $\texttt{Attn\_out}_{\texttt{i}-1}$, $\texttt{FFN\_out}_{\texttt{i}-1}$). As shown in the table, $\texttt{Tblock\_in}_\texttt{i}$ is highly dependent on the $\texttt{Tblock\_in}_{\texttt{i}-1}$ rather than others. InfiniGen leverages this key observation to speculate on the attention pattern of Layer i using the attention input of Layer $i-1$. Note that $\texttt{Tblock\_in}$ gradually changes across the layers; the inputs to distant layers are distinct.

Skewed Partial Weight. We observe that the attention score highly depends on a few columns in the query and key matrices. Figure 7(b) shows the values in a query matrix of Layer 18 of the OPT-13B model, where the column-wise patterns indicate that there exist certain columns with large magnitudes in the matrix; we observe the same patterns in the key and query matrices across different layers and models. The large magnitude columns have a great impact on the attention pattern because the dot product between the query and key is highly affected by these few columns. The column-wise pattern in the attention input indicates that there is little variance between each row in the outlier channels. Thus, the dot product between any row of the attention input and a column of the weight matrix could have a similarly large magnitude, which induces the outlier channels in the query and key matrices.

Going one step further, if we make a few columns in the query and key matrices have much larger magnitude than others, a much smaller number of columns significantly affects the attention pattern. We can do this by multiplying the query and key weight matrices with the same orthogonal matrix A. Since the transpose of the orthogonal matrix is the inverse of itself, the proposed operation does not change the final result, as shown in Equation 2 (i.e., this is mathematically equivalent to QK^T, not an approximation):

$$\begin{aligned} \tilde{Q} = X_a \times W_Q \times A, &\qquad \tilde{K} = X_a \times W_K \times A \\ \tilde{Q} \times \tilde{K}^T &= X_a \times W_Q \times A \times (X_a \times W_K \times A)^T \\ &= X_a \times W_Q \times A \times A^T \times W_K^T \times X_a^T \\ &= X_a \times W_Q \times W_K^T \times X_a^T \\ &= X_a \times W_Q \times (X_a \times W_K)^T \\ &= Q \times K^T, \end{aligned} \tag{2}$$

where $\tilde{Q}$ and $\tilde{K}$ are skewed query and key matrices, while W_Q and W_K are query and key weight matrices. X_a denotes the attention input. We set the orthogonal matrix A whose direction

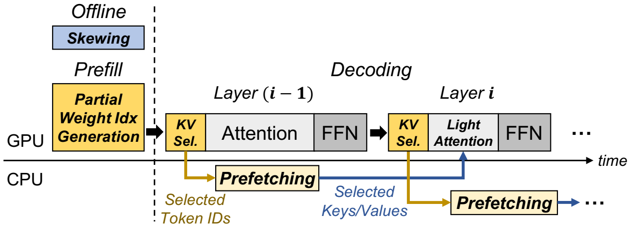


Figure 8: Operation flow of the prefetching module of InfiniGen.

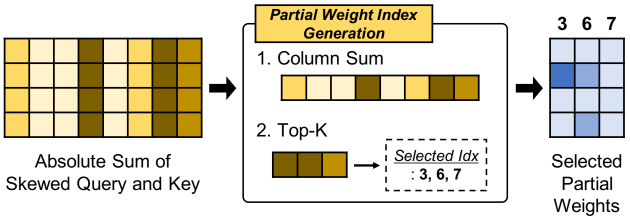


Figure 9: Partial weight generation in the prefill stage.

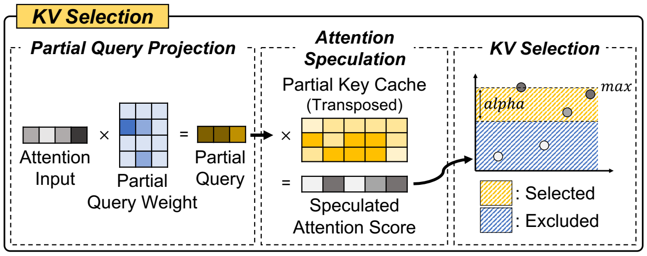


Figure 10: Attention score speculation in the decoding stage.

aligns with the direction that the query matrix stretches the most. Specifically, we first decompose the query matrix using SVD and obtain $\mathbf{U}$, Σ, and $\mathbf{V}$. We then set A to orthogonal matrix $\mathbf{V}$ to align the column vectors with the standard unit vectors as $\mathbf{V}^T\mathbf{A} = \mathbf{V}^T\mathbf{V} = I$, where I is an identity matrix. We formulate the skewed query matrix as follows:

$$\tilde{Q} = Q \times A = \mathbf{U}\Sigma\mathbf{V}^T \times A = \mathbf{U}\Sigma\mathbf{V}^T \times \mathbf{V} \tag{3}$$

In this way, we can make a few columns with large magnitudes in $\tilde{Q}$ without altering the result of computation, as discussed in Section 2.4.

4.3 Efficiently Prefetching KV Cache

Prefetching Scheme. Figure 8 shows the operation flow of the prefetching module in InfiniGen. In the offline phase, InfiniGen modifies the *weight* matrices to generate skewed query and key matrices. To achieve this, InfiniGen first runs the forward pass of the model once with a sample input. During this process, InfiniGen gathers the query matrix from each layer and performs singular value decomposition (SVD) of each query matrix. The skewing matrix (A_i) of each layer is obtained using the decomposed matrices of the query matrix, as shown in Equation 3. This matrix is then multiplied with each of the query and key weight matrices in the corresponding layer. Importantly, after the multiplication, the dimensions of the weight matrices remain unchanged. Note that the skewing is a *one-time* offline process and does not incur any runtime overhead because we modify the weight matrices that are invariant at runtime. As we exploit the column-wise pattern, which stems from the intrinsic property of the model rather than the input, whenever we compute the query and key for different inputs after the skewing, the values exhibit a high degree of skewness, thereby improving the effectiveness of our prefetching module. Note that skewing does not alter the original functionality. Even with the skewing, the attention layer produces identical computation results.

Prefill Stage. In the prefill stage, InfiniGen selects several important columns from the query weight matrix and the key cache to speculate on the attention pattern, and generates *partial* query weight and key cache matrices used in the decoding stage. Figure 9 shows how InfiniGen creates these partial matrices. Because we multiply each column in the query matrix with the corresponding row in the transposed key matrix, it is essential to select the *same* column indices in the query weight matrix and the key cache to obtain a proper approximation of the attention score. However, the indices of the outlier columns of the skewed query ($\tilde{Q}$) and key ($\tilde{K}$) matrices may not align exactly. To obtain partial matrices that capture the outliers, we first take the element-wise absolute values of the skewed query and key matrices, then add these two matrices together. This helps us calculate the sum of each column and perform top-k operation only once while accommodating the outlier columns of both query and key matrices. We then sum the elements in each column and select the top-k columns in the matrix; we choose 30% of the columns in our work. Using the sum of column values captures the global trend of each column while minimizing the effect of variance in each row. The selected columns better approximate the attention pattern because of the use of skewed query and key matrices.

Decoding Stage. In the decoding stage, InfiniGen speculates on the attention pattern of the next layer and determines the critical keys and values to prefetch. Figure 10 shows how InfiniGen computes the *speculated* attention score. At Layer $i-1$, we use the partial query weight matrix and key cache of Layer i, which are identified in the prefill stage, along with the attention input of Layer $i-1$. After multiplying the partial query and partial key cache, InfiniGen selects tokens with high attention scores.

We set the threshold considering the maximum value of the speculated attention score. We select only the tokens with an attention score greater than the maximum score subtracted by *alpha*. It is noted that subtraction from the attention score results in division after softmax. For example, assume that the attention score of the 3rd token is the maximum attention score minus 5. Once we apply softmax to the attention scores, the attention weight of the 3rd token is the maximum attention weight divided by $e^5 \approx 148.4$. Even though we do not use this token, it does not noticeably hurt the accuracy of the model since it accounts for less than 1% of importance ($\approx 1/148.4$) after softmax. Thus, InfiniGen only prefetches the keys and values of the tokens with an attention score larger than the highest attention score minus *alpha*. As multiple attention heads are computed in parallel, we ensure that each head in the same layer fetches the same number of tokens by averaging the number of tokens between the maximum score and the threshold across the heads.

By reducing the amount of KV cache to load and compute, InfiniGen effectively reduces the loading latency (i.e., data transfer from CPU to GPU) while maintaining an output quality similar to that of the original model with a full KV cache. Moreover, as InfiniGen does not require a fixed number of tokens to load from CPU memory, it utilizes only the necessary PCI interconnect bandwidth. InfiniGen initiates speculation and prefetching from Layer 1 because the outliers, which are essential for exploiting input similarity, emerge during the computation in Layer 0.

4.4 KV Cache Pool Management

We manage the KV cache as a pool, offloading to the CPU memory and prefetching only the essential amount to the GPU. While CPU memory is more affordable and larger than GPU memory, it still has limited capacity. Hence, for certain deployment scenarios, it might be crucial to confine the size of the KV cache pool and remove less important KV entries that are infrequently selected by query tokens. We extend the design to incorporate a user-defined memory size limit. During runtime, when the size of the CPU memory reaches a user-defined limit, the KV cache pool manager selects a victim KV entry for eviction. Subsequently, the manager overwrites the selected victim with the newly generated key and value, along with updating the corresponding partial key cache residing in the GPU. It is noted that the order of KV entries can be arbitrary, as long as the key and value of the same token maintain the same relative location in the KV cache pool.

The policy of selecting a victim is important since it directly impacts model accuracy. We consider a counter-based policy along with two widely used software cache eviction policies: FIFO [7, 69, 70] and Least-Recently-Used (LRU) [2]. The FIFO-based policy is easy to implement with low overhead but results in a relatively large accuracy drop since it simply evicts the oldest residing token. The LRU-based policy generally exhibits a smaller decrease in accuracy but often entails a higher runtime overhead. In general, LRU-based policy uses a doubly linked list with locks to promote accessed objects to the head, which requires atomic memory updates for accessed KV entries. In the case of the counter-based policy, the pool manager increments a counter for each prefetched KV entry and selects a victim with the smallest count in the KV cache pool. If any counter becomes saturated, all the counter values are reduced by half. We observe that the counter-based policy and the LRU-based one show comparable model accuracy, which we discuss in Section 5.2. We opt for a counter-based approach due to its simpler design and to avoid atomic memory updates for better parallelism.

5 Evaluation

5.1 Experimental Setup

Model and System Configuration. We use Open Pre-trained Transformer (OPT) models [77] with 6.7B, 13B, and 30B parameters for evaluation. The 7B and 13B models of Llama-2 [60] are also used to demonstrate that InfiniGen works effectively across different model architectures. We run the experiments on a system equipped with an NVIDIA RTX A6000 GPU [44] with 48GB of memory and an Intel Xeon Gold 6136 processor with 96GB of DDR4-2666 memory. PCIe 3.0 ×16 interconnects the CPU and GPU.

Workload. We evaluate using few-shot downstream tasks and language modeling datasets. We use five few-shot tasks from the lm-evaluation-harness benchmark [23]: COPA [54], OpenBookQA [42], WinoGrande [55], PIQA [8], and RTE [62]. The language modeling datasets used are WikiText-2 [41] and Penn Treebank (PTB) [38]. Additionally, randomly sampled sentences from the PG-19 dataset [52] are used to measure the speedup with long sequence lengths.

Baseline. We use two inference environments that support KV cache offloading: CUDA Unified Virtual Memory (UVM) [4] and FlexGen [57]. On UVM, all data movements between the CPU and GPU are implicitly managed by the UVM device driver, thereby enabling offloading without requiring intervention from the programmer. In contrast, FlexGen uses explicit data transfers between the CPU and GPU. For the FlexGen baseline, unless otherwise specified, we explicitly locate all the KV cache in the CPU memory. The model parameters are stored in the GPU memory as much as possible, with the remainder in the CPU memory. We compare InfiniGen with two different KV cache management methods: H_2O [78] and Quantization [57]. H_2O, a recent method in KV cache management, maintains the KV cache of the important or recent tokens by assessing the importance of each token and discarding others. Quantization-based compression applies group-wise asymmetric quantization to the KV cache.

Key Metric. We evaluate accuracy (%) to assess the impact of approximation when InfiniGen, H_2O, and Quantization are used. For the language modeling tasks with WikiText-2 and PTB, we use perplexity as a metric; lower perplexity means better accuracy. To present performance improvements, we measure the wall clock time during inference with varying batch sizes and sequence lengths. The partial weight ratio is set to 0.3. We set alpha to 4 for OPT and 5 for Llama-2, resulting in using less than 10% of the KV cache on average across the layers. For each layer, we allow sending up to 20% of the total KV cache to the GPU if it contains more candidates.

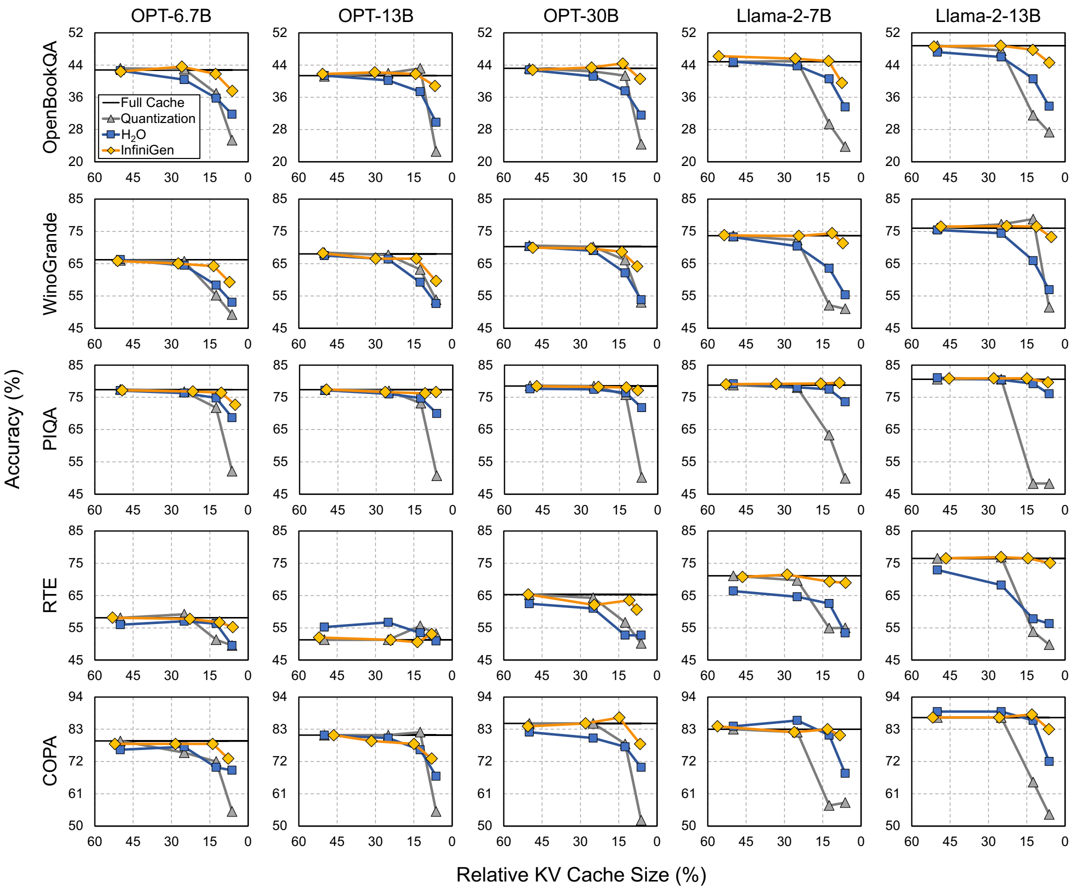
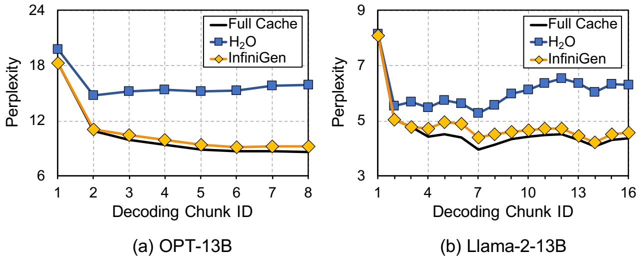


Figure 11: Accuracy of LLMs on 5-shot tasks in lm-evaluation-harness.



(a) OPT-13B

(b) Llama-2-13B

Figure 12: Perplexity of OPT-13B and Llama-2-13B for WikiText-2 dataset. Lower is better. Perplexity is computed for each decoding chunk that contains 256 tokens.

The partial weight ratio and alpha are determined based on a sensitivity study for each model to balance accuracy and inference latency, which we discuss in Section 6.1.

5.2 Language Modeling

Accuracy on lm-evaluation-harness. Figure 11 shows the accuracy of the baselines and InfiniGen across different models with 5-shot tasks. The relative KV cache size indicates the size of the KV cache involved in the attention computation compared to the full-cache baseline (e.g., a 10% relative KV cache size means that 10% of the full KV cache size is used). InfiniGen consistently shows better accuracy across the models and tasks when the relative KV cache size is less than 10%, whereas the others exhibit a noticeable accuracy drop due to insufficient bit widths (Quantization) or permanent KV cache elimination (H_2O). This implies that our proposed solution can effectively reduce the KV cache transfer overhead while preserving model accuracy. For relative KV cache sizes larger than 10%, the accuracy with InfiniGen closely matches that of the full-cache baseline. In some cases, InfiniGen even shows slightly better accuracy than the full-cache baseline. This is likely because reducing the amount of the KV cache participating in the attention computation can help the model focus more on critical tokens.

Sequence Length. Figure 12 shows the perplexity of two different models with InfiniGen and the baselines, as the sequence length increases. In this experiment, H_2O is configured to use the same amount of KV cache as InfiniGen. The sequence lengths are 2048 and 4096 for OPT-13B and Llama-2-13B, respectively. For a clearer view, we evaluate perplexity with consecutive 256 tokens as a group, which is referred to as a decoding chunk in the figure. The results show that even though the sequence length becomes longer (i.e., the decoding chunk ID increases), the perplexity of InfiniGen remains consistently comparable to the full-cache baseline,

Table 2: Perplexity on WikiText-2 and PTB with 2048 sequence length with or without KV cache memory limits. Lower is better.

Scheme	OPT-6.7B		OPT-13B		OPT-30B		Llama-2-7B		Llama-2-13B	
	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB
100%	11.68	13.86	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94
80-FIFO%	19.64	16.82	30.99	33.84	30.66	35.45	22.26	61.88	21.41	32.34
80-LRU%	11.68	13.85	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94
80-Counter%	11.68	13.86	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94

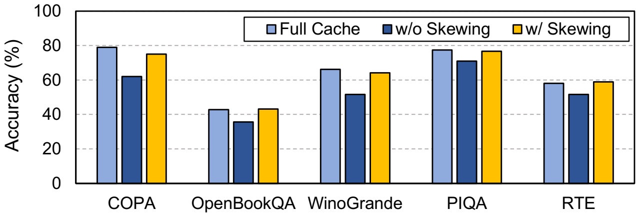


Figure 13: Accuracy on the lm-evaluation-harness benchmark with or without skewing on OPT-6.7B.

while H_2O shows an increasing divergence from the baseline. H_2O suffers from permanent KV cache elimination and may not retain a sufficient amount of KV cache in certain layers due to its fixed budget. In contrast, InfiniGen dynamically computes attention using only the essential amount of KV cache for each layer. The difference is likely to widen as the models become capable of handling much longer sequences.

Effect of Skewing. Figure 13 shows the accuracy with or without key/query skewing on the OPT-6.7B model. For the experiment, we use a fixed KV cache budget of 20%, instead of using a dynamic approach, to clearly show the effect of skewing. We observe that several language models (e.g., Llama-2) show a small drop in accuracy without skewing. For some models such as OPT-6.7B, however, we see a large accuracy drop if we do not apply the skewing method as shown in Figure 13. This indicates that in the case of OPT-6.7B, the partial weight does not adequately represent the original matrix without skewing. After applying our skewing method, we achieve accuracy similar to the full-cache baseline. Our skewing method effectively skews key and query matrices such that a few columns can better represent the original matrices.

KV Cache Pool Management. Table 2 shows the perplexity of five different models with or without limiting the memory capacity for WikiText-2 and PTB. We compare FIFO-based, LRU-based, and Counter-based victim selection policies in Section 4.4 under the 80% memory limit of a full KV cache. We also present the perplexity results with no memory limit (100%). The FIFO-based approach shows the worst model performance because it simply deletes the oldest KV entry regardless of its importance. The LRU and Counter-based approaches show perplexity that is almost similar to that with no memory limit. We choose a Counter-based victim selection policy instead of an LRU-based approach because the LRU-based approach typically needs to maintain a doubly linked list queue with locks for atomic memory updates.

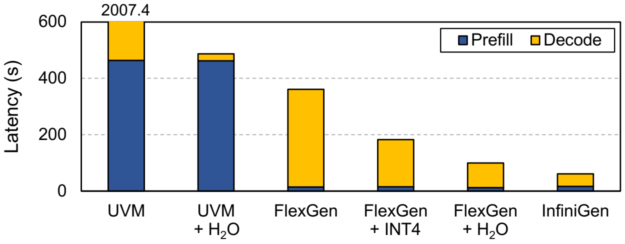


Figure 14: Inference latency on OPT-13B with a sequence length of 2048 (1920 input and 128 output tokens) and a batch size of 20.

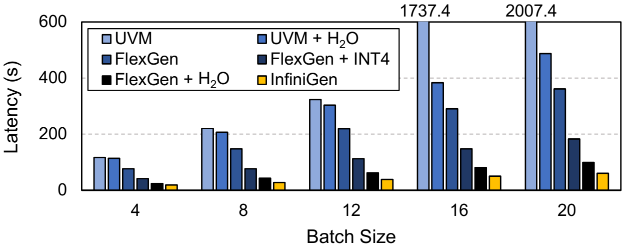


Figure 15: Inference latency for 5 different batch sizes on OPT-13B with a sequence length of 2048 (1920 input and 128 output tokens).

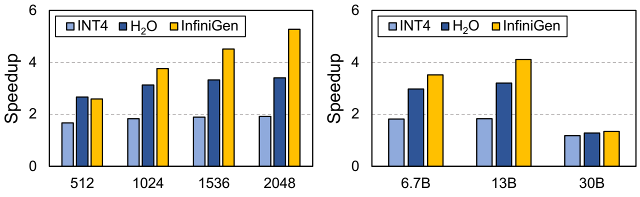
5.3 Performance

In this section, we refer to H_2O (with a KV cache budget of 20%) and 4-bit quantization implemented on top of FlexGen as H_2O and INT4.

Inference Latency. Figure 14 shows the inference latency including the prefill and decoding stages. We use the OPT-13B model with 1920 input tokens, 128 output tokens, and a batch size of 20. InfiniGen achieves 1.63×-32.93× speedups over the baselines. The performance benefit mainly comes from the significantly reduced amount of KV cache to load from the CPU memory due to our dynamic approach.

UVM shows an extremely long latency because the working set size (i.e., the size of the model parameters and KV cache) is larger than the GPU memory capacity, thereby leading to frequent page faults and data transfers between the CPU and GPU. The prefill stage of UVM + H_2O also shows a long latency due to the page faults and data transfers. However, because all required data are migrated to the GPU memory after the prefill stage, UVM + H_2O shows a substantially shorter decoding latency. FlexGen loads the full KV cache with high precision (i.e., FP16) from the CPU memory for every attention computation. On the other hand, INT4 and H_2O load relatively small amounts of the data from the CPU because of the low-bit data format (INT4) or a smaller size of the KV cache (H_2O). However, they still load larger amounts of data than InfiniGen; even with low precision, INT4 loads the KV cache of all the previous tokens; H_2O always loads the same amount of data no matter how many tokens are actually important in each layer. As a result, InfiniGen achieves better performance than both of them.

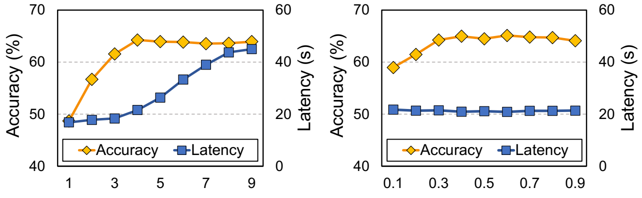
Batch Size. Figure 15 shows the inference latency across different batch sizes. The results show that InfiniGen achieves



(a) Sequence Length

(b) Model Size

Figure 16: Speedup over the FlexGen baseline across (a) sequence lengths and (b) model sizes.



(a) Alpha Value

(b) Partial Weight Ratio

Figure 17: Accuracy and inference latency across (a) alpha values and (b) partial weight ratios.

lower latency than others across the batch sizes (1.28×-34.64×). As the batch size increases, the performance gap between InfiniGen and others becomes larger. UVM and UVM + H_2O show increasing latency mainly due to frequent page faults in the prefill stage. For UVM, the latency also rapidly increases at a batch size of 16 because the working set size exceeds the GPU memory capacity for *both* prefill and decoding stages. As the batch size keeps increasing, UVM + H_2O will face the same problem as well.

The latency of FlexGen almost linearly increases with the batch size because the KV cache transfer occupies the majority of the inference latency. As we increase the batch size from 4 to 20, the throughput (tokens per second) of InfiniGen increases from 27.36 to 41.99, while INT4 and H_2O offer a small increase in throughput (from 12.22 to 14.02 and from 21.31 to 25.70 each). By dynamically adjusting the amount of the KV cache to load, InfiniGen achieves scalable performance across the batch sizes.

Sequence Length. Figure 16(a) shows the speedup of INT4, H_2O, and InfiniGen over FlexGen on OPT-13B across different sequence lengths. With a batch size of 8, we use four different input/output configurations. Each configuration comprises 128 output tokens and 384, 896, 1408, 1920 input tokens (i.e., a total number of tokens ranging from 512 to 2048). The speedup of InfiniGen continues to increase across the sequence lengths (up to 5.28×), whereas INT4 and H_2O show saturating speedups (up to 1.92× and 3.40×). This suggests that neither INT4 nor H_2O provides a scalable solution for KV cache management. INT4 shows a negligible increase in speedup due to the inherent growth in the size of the KV cache. Similarly, H_2O lacks scalability due to its fixed ratio of the KV cache budget; as the sequence length increases, H_2O stores and loads more KV cache.

Even though the sequence length increases, the number of tokens that each token attends to does not increase linearly. For instance, in the OPT-13B model, we count the number of important tokens with attention scores larger than $(max - 4)$ and identify that, on average, 37, 60, 66, and 73 tokens are assessed as *important* for sequence lengths of 512, 1024, 1536, and 2048, respectively. H_2O, which employs 20% of a fixed KV cache budget, loads 409 tokens for the sequence length of 2048, while only 73 tokens are relatively important. In contrast, InfiniGen naturally captures this trend (i.e., a non-linear increase in the number of important tokens) by dynamically observing the speculated attention score.

Model Size. Figure 16(b) shows the speedup of INT4, H_2O, and InfiniGen over FlexGen on three different model sizes. We use 1920 input tokens and 128 output tokens with a batch size of 4 for the experiment. The results show that InfiniGen outperforms others across the model sizes. As the model size increases from 6.7B to 13B, the speedup of InfiniGen also increases by 1.17×, while others do not lead to a noticeable increase in speedup. For most of the layers, InfiniGen loads a smaller amount of KV cache than H_2O because a relatively small number of tokens are needed. Thus, InfiniGen performs better than H_2O as the model size becomes larger due to the increased number of Transformer blocks. For the 30B model, the model parameters do not fit in the GPU memory. As such, we offload 30% of the model parameters to the CPU. In this case, the size of the offloaded parameters is 1.7× larger than the KV cache size. Even so, InfiniGen shows a 1.34× speedup over FlexGen, while others achieve 1.18× and 1.28× each.

6 Analysis and Discussion

6.1 Sensitivity Study

We use the OPT-6.7B model with 1920 input tokens, 128 output tokens, and a batch size of 8. The accuracy is evaluated with the WinoGrande task in lm-evaluation-harness.

Threshold and Alpha. As discussed in Section 4.3, we load the KV cache of the tokens with a speculated attention score greater than the threshold (i.e., the maximum attention score minus alpha). Increasing alpha results in fetching more KV entries to the GPU, thus increasing inference latency but also improving accuracy. Figure 17(a) shows such trade-offs between accuracy and inference latency for nine different alpha values with a partial weight ratio of 0.3. The results show that more KV entries are fetched and involved in attention computation as alpha increases, thereby leading to better accuracy. For the alpha values beyond 4, however, since most important tokens are already included, the accuracy does not further increase, while the cost for KV transfers and attention computation keeps increasing. This trend is similarly observed in other models, and we thus opt for an alpha value of 4 or 5 to strike a balance between inference latency and accuracy.

Partial Weight Ratio. Figure 17(b) shows the accuracy and

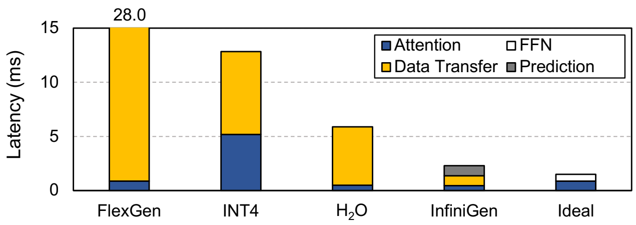


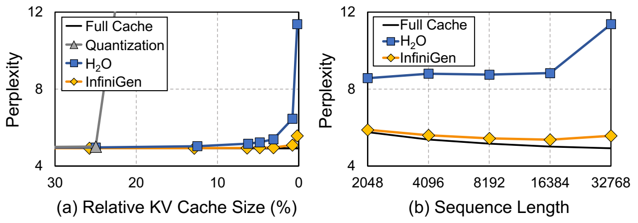
Figure 18: Latency breakdown of a Transformer block for OPT-13B with a sequence length of 2048 and a batch size of 8.

inference latency across different partial weight ratios with an alpha value of 4. As shown in the figure, the amount of partial weights has a negligible impact on inference latency because the cost for computing the speculated attention score is relatively small. Note that the amount of KV cache to transfer is *not* related to the partial weight ratio. However, increasing the partial weight ratio results in higher memory consumption for partial weights and key cache (e.g., doubling the ratio doubles the memory consumption overhead). The accuracy also does not noticeably differ beyond a ratio of 0.3. In our work, we opt for a partial weight ratio of 0.3 to achieve better accuracy while considering memory consumption overhead.

6.2 Overhead

Prefetching Overhead. Figure 18 shows the latency breakdown of executing a single Transformer block for the OPT-13B model; FFN is not shown in the figure for schemes other than Ideal because it is fully overlapped with data transfer time. Ideal is a scenario where all the computations (i.e., attention and FFN) are performed on the GPU without any data transfer between the CPU and GPU. As shown in the results, the key performance bottleneck of FlexGen and H_2O is the data transfer overhead, which occupies 96.9% and 91.8% of the execution time, respectively. For INT4, due to the quantization and dequantization overhead, attention computation also occupies a large portion of the execution time in addition to the data transfer. InfiniGen, on the other hand, considerably improves the inference speed over FlexGen by reducing the amount of data transfer with our dynamic KV cache prefetching. Furthermore, InfiniGen is only 1.52× slower than Ideal, while others show 3.90×-18.55× slowdowns.

Memory Consumption. InfiniGen uses the partial query weight and key cache for speculation. For a ratio of 0.3, the sizes of the partial query weight and key cache are only 2.5% and 15% of the total model parameters and total KV cache, respectively. While we simply store them in the GPU during our experiments, we can manage the storage overhead in various optimized ways if needed. For example, we can store only the column indices of the partial query weight and retrieve the column vectors from the full query weight matrix (which already resides in the GPU) as needed for partial query projection. Additionally, we can place the partial key cache in the CPU and perform speculation on the CPU after fetching the partial query from the GPU. Even a naïve method of lowering the partial ratio would likely still provide better accuracy compared to other methods while reducing storage overhead. In summary, by minimally sacrificing inference performance, we can greatly reduce the storage overhead on the GPU if necessary.



(a) Relative KV Cache Size (%)

(b) Sequence Length

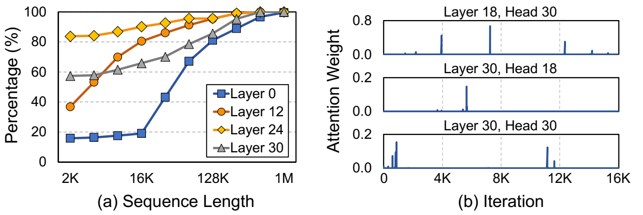
Figure 19: Perplexity of Llama-2-7B-32K across (a) relative KV cache sizes with a sequence length of 32768 and (b) sequence lengths while retaining 64 tokens. Lower is better. Llama-2-7B-32K is a fine-tuned version capable of processing up to 32K tokens using position interpolation [12]. Quantization is omitted in (b) since the KV cache cannot be compressed below 6.25% (i.e., 1 bit).

6.3 Long Context Window

Figure 19 shows the perplexity of the Llama-2-7B-32K model, which can process up to 32K tokens, across the relative cache sizes and sequence lengths. We use the WikiText-2 dataset for the experiment. As the context window size increases for future LLMs, the relative portion of the KV cache that the GPU can retain would decrease due to the limited capacity of GPU memory.

Figure 19(a) shows that InfiniGen maintains perplexity levels close to the full-cache baseline even as the relative KV cache size decreases, without leading to a noticeable increase in perplexity even with much smaller cache sizes. In contrast, other methods increase perplexity compared to the full-cache baseline and significantly diverge at certain sizes due to insufficient bit widths for preserving adequate information on all keys and values (Quantization) or the permanent removal of KV cache entries (H_2O). As shown in Figure 19(b), the perplexity gap between InfiniGen and H_2O widens for longer sequence lengths, which is likely to increase further for sequence lengths beyond 32K. This implies that InfiniGen can scale to longer sequences and better preserve model accuracy compared to others.

We further speculate on how InfiniGen would benefit in an era of million-token context windows by analyzing a model capable of handling 1 million tokens. Figure 20(a) shows that the percentage of query tokens that attend to less than 1% of key tokens increases as the sequence length becomes longer. InfiniGen can adapt to this changing trend by dynamically adjusting the amount of the KV cache to load, whereas prior fixed-budget/pruning approaches would not easily adjust the effective KV cache size. Figure 20(b) further shows that the attention weights of key tokens can change across iterations;



(a) Sequence Length

(b) Iteration

Figure 20: Analysis of 1 million tokens using Llama-3-8B-1048K. (a) Percentage of query tokens that attend to less than 1% of the key tokens across the sequence lengths. (b) Attention weight of sampled key tokens from different layers and heads across the last 16K iterations out of 1 million tokens.

the sampled key tokens show sudden *spikes* after thousands of iterations with significantly low attention weights (e.g., the 7425th iteration out of the last 16K iterations in Layer 18, Head 30). We observe that the prior approaches that permanently eliminate tokens while they are unimportant could lose the critical contexts if they become important again at later iterations. In contrast, InfiniGen can preserve model performance by keeping the temporarily unimportant KV entries for potential future use.

7 Related Work

DNN Serving Systems. A systematic approach to enabling an efficient and fast model serving system is an important topic that has been widely studied by both academia and industry. Some prior works focus on distributed systems with predictable latency for service-level objectives (SLOs) [15, 16, 25, 56]. Other works improve parallelism and throughput of the system through preemption [28, 75], fine-grained batching [17, 21, 71], or memory optimizations [18, 35, 58].

Several other works aim at achieving high throughput execution with limited GPU memory by offloading parameters to secondary storage (e.g., CPU memory and disk). Some of them build on CUDA Unified Memory [46] with prefetching [31, 39], while others explicitly move tensors in and out as needed for computation [29, 30, 48, 72, 73]. FlexGen [57] is a recent LLM serving system that enables high-throughput inference on a single GPU by offloading weights and KV cache to CPU memory and disk. InfiniGen is orthogonal to FlexGen and can work in conjunction with it to efficiently offload and prefetch the KV cache.

KV Cache Management. vLLM [35] mitigates the KV cache memory waste from fragmentation and duplication. StreamingLLM [67] enables LLMs to generate longer sequence lengths than the trained ones. However, since neither vLLM nor StreamingLLM reduces the size of the KV cache, data transfers still incur a significant overhead in offloading-based inference systems. InfiniGen complements the KV cache management to reduce the data transfer overhead, which is a major bottleneck in offloading-based systems.

Efficient LLM Inference. There are lines of research that exploit *quantization* or *sparsity* to make LLMs efficient through algorithmic methods [13, 19, 22, 34, 66] or hardware-software co-design [26, 27, 36, 50]. Regarding sparsity, most algorithm-based works focus on reducing the model size by exploiting the sparsity of weights. Alternatively, H_2O and Sparse Transformer [13] leverage the row-wise (i.e., token-level) sparsity in the KV cache by *permanently* removing certain KV entries. On the other hand, most hardware-software co-design studies focus on relaxing the quadratic computational complexity in the prefill stage by skipping non-essential key tokens with the aid of specialized hardware. However, they often do not reduce memory access as they identify the important key tokens only after scanning all the elements of the key tensors.

Kernel fusion [18, 32] is another approach to mitigating the quadratic memory overhead of attention in the prefill stage. InfiniGen can be implemented with kernel fusion techniques to alleviate the overhead of KV cache access in the decoding stage. To our knowledge, this is the *first* work to enable efficient LLM inference by prefetching only essential KV entries in offloading-based inference systems.

8 Conclusion

The size of the KV cache poses a scalability issue in high-throughput offloading-based inference systems, even surpassing the model parameter size. Existing KV cache eviction policies show a large accuracy drop and do not efficiently use the interconnect bandwidth when they are employed in offloading-based LLM systems. We propose InfiniGen, an offloading-based dynamic KV cache management framework that efficiently executes inference of large language models. InfiniGen exploits the attention input of the previous layer to speculatively prefetch the KV cache of important tokens. We manipulate the query and key weights to make the speculation more efficient. InfiniGen shows substantially shortened inference latency while preserving language model performance. It also shows much better scalability regarding the batch size, sequence length, and model size compared to prior solutions.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Petros Maniatis for their valuable feedback. This work was supported in part by a research grant from Samsung Advanced Institute of Technology (SAIT) and by the artificial intelligence semiconductor support program to nurture the best talents (No. RS-2023-00256081) supervised by Institute for Information & Communications Technology Planning & Evaluation (IITP). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

References

[1] DeepL. https://www.deepl.com/translator.

[2] Memcached. https://memcached.org.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Tyler Allen and Rong Ge. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[6] Anthropic. The claude 3 model family: Opus, sonnet, haiku. 2024. https://www.anthropic.com/claude.

[7] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[8] Yonatan Bisk, Rowan Zellers, Ronan bras, Jianfeng Gao, and Choi Yejin. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[9] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[10] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[13] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[14] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[15] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2020.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[17] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[18] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[19] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[20] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[21] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021.

[22] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[23] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation, 2021.

[24] GitHub. Copilot. `https://github.com/features/copilot`.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[26] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2023.

[27] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[28] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[29] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[30] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[31] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. Deepum: Tensor migration and prefetching in unified memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[32] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. Flat: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[33] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[34] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2023.

[36] Jungi Lee, Wonbeom Lee, and Jaewoong Sim. Tender: Accelerating large language models via tensor decomposition and runtime requantization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2024.

[37] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[38] Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, 1994.

[39] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[40] Pierre-Emmanuel Mazare, Samuel Humeau, Martin Raison, and Antoine Bordes. Training millions of personalized dialogue agents. In *Conference on Empirical*

Methods in Natural Language Processing (EMNLP), 2018.

[41] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[42] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[44] NVIDIA. NVIDIA RTX A6000 Graphics Card. https://www.nvidia.com/en-us/design-visualization/rtx-a6000/.

[45] NVIDIA. Triton inference server. https://developer.nvidia.com/triton-inference-server.

[46] Nvidia. Unified memory programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[47] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[48] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[49] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of the Machine Learning and Systems (MLSys)*, 2023.

[50] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. Dota: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[52] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[53] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[54] Melissa Roemmele, Cosmin Bejan, and Andrew Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium Series*, 2011.

[55] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 2021.

[56] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2019.

[57] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single gpu. In *Proceedings of International Conference on Machine Learning (ICML)*, 2023.

[58] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[62] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *EMNLP Workshop BlackboxNLP*, 2018.

[63] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[64] Yiming Wang, Zhuosheng Zhang, and Rui Wang. Element-aware summarization with large language models: Expert-aligned evaluation and chain-of-thought method. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[65] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[66] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023.

[67] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[68] Haoran Xu, Young Jin Kim, Amr Sharaf, and Hany Hassan Awadalla. A paradigm shift in machine translation: Boosting translation performance of large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[69] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2023.

[70] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[71] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[72] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[73] Haoyang Zhang, Yirui Eric Zhou, Yu Xue, Yiqi Liu, and Jian Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2023.

[74] Haoyu Zhang, Jingjing Cai, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. In *Conference on Computational Natural Language Learning (CoNLL)*, 2019.

[75] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[76] Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[77] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[78] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H_2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Llumnix: Dynamic Scheduling for Large Language Model Serving

Biao Sun[*†], Ziming Huang[*†], Hanyu Zhao[*], Wencong Xiao, Xinyi Zhang[†], Yong Li, Wei Lin

Alibaba Group

Abstract

Inference serving for large language models (LLMs) is the key to unleashing their potential in people's daily lives. However, efficient LLM serving remains challenging today because the requests are inherently heterogeneous and unpredictable in terms of resource and latency requirements, as a result of the diverse applications and the dynamic execution nature of LLMs. Existing systems are fundamentally limited in handling these characteristics and cause problems such as severe queuing delays, poor tail latencies, and SLO violations.

We introduce Llumnix, an LLM serving system that reacts to such heterogeneous and unpredictable requests by *runtime rescheduling* across multiple model instances. Similar to context switching across CPU cores in modern operating systems, Llumnix reschedules requests to improve load balancing and isolation, mitigate resource fragmentation, and differentiate request priorities and SLOs. Llumnix implements the rescheduling with an efficient and scalable live migration mechanism for requests and their in-memory states, and exploits it in a dynamic scheduling policy that unifies the multiple rescheduling scenarios elegantly. Our evaluations show that Llumnix improves tail latencies by an order of magnitude, accelerates high-priority requests by up to 1.5×, and delivers up to 36% cost savings while achieving similar tail latencies, compared against state-of-the-art LLM serving systems. Llumnix is publicly available at `https://github.com/AlibabaPAI/llumnix`.

1 Introduction

Large language models (LLMs) such as the GPT series [15, 49] are bringing generative AI to an unprecedented level. Their human-level generation capabilities are being quickly adopted in a wide range of domains, inspiring many imaginations for future applications, and expected to have profound influences on how people live and work.

Inference serving of LLMs plays a key role in LLM-powered services, becoming a critical workload in datacenters. Such services are typically backed by multiple instances of the LLM deployed on a GPU cluster. The system involves a *scheduler* and an *inference engine*, where a request is first dispatched by the scheduler to a model serving instance, then gets executed by the inference engine inside. The requests are typically batched for execution on each instance to increase throughput and cost efficiency.

We observe unique characteristics of LLMs that call for new design philosophy of the serving infrastructure. The first is *workload heterogeneity*. LLMs are designed to be universal, by learning as much knowledge as possible from whatever domains. People can query the same LLM in totally different situations or even build custom applications atop LLMs for various scenarios; for all of these, a context-specific input (*i.e.*, prompt) is all you need [15]. Such universality and application diversity lead to heterogeneity of the inference requests, in terms of input lengths, output lengths, expected latencies, *etc*. For instance, the task of summarizing long text can introduce significant input lengths, where the latency of returning the first token (word) is often important to user experience [38].

The second characteristic is *execution unpredictability*. Serving an LLM request needs to run the model for multiple iterations, each producing a single output token; however, it is not known a priori how many tokens will be generated eventually. Moreover, the iterative generation also brings considerable GPU memory consumption that dynamically grows with the tokens. As such, the execution time and the resource demand of a request are both unpredictable.

These characteristics make an LLM inherently a *multi-tenant* and *dynamic* environment, serving heterogeneous and unpredictable workloads on multiple instances. This behavior is fundamentally different from traditional DNN models, where the requests are homogeneous and the execution is one-shot, stateless, and deterministic. Instead, we find LLMs more similar to modern operating systems hosting processes with dynamic working sets and different priorities on multiple cores. Managing such systems has complex goals, which goes

[*] Equal contribution.

[†] Work done during internship at Alibaba Group.

beyond what existing inference serving systems are designed for. Although there has been a series of LLM-tailored inference engines that shows superior performance, such systems concentrate on the sole goal of maximizing throughput within a single instance [34, 46, 67]. The request scheduling across instances, on the other hand, has received relatively little attention; the common practice today is still to use generic scheduling systems or policies inherited from the era of traditional DNNs [4, 28, 35, 47, 53]. Such a clear gap introduces challenges in the following aspects that are crucial in multi-tenant environments and online services.

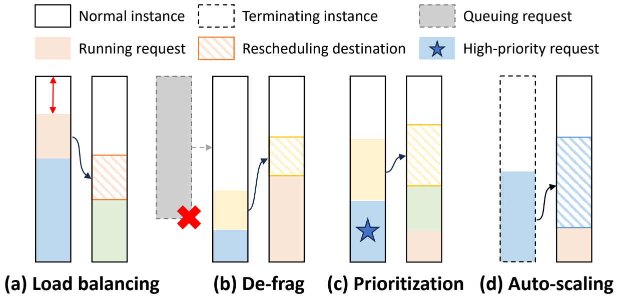
Isolation. The system can hardly provide performance isolation to requests as their memory consumption grows unpredictably. Memory contention incurs performance interference and even preemptions of certain requests in a batch [34], leading to highly unstable latencies and service-level objective (SLO) violations, significantly sacrificing user experiences.

Fragmentation. The varying request lengths and memory demands inevitably result in memory fragmentation across instances, which introduces conflicting scheduling objectives. The running requests prefer load balancing to reduce preemptions and interference, but such load balancing fragments the free memory space across instances at the same time. The fragmentation can cause long queuing delays of new requests that instead require a large space on one instance for the input sequences. This conflict is difficult for the scheduler to reconcile with unpredictable arrivals and lengths of requests.

Priorities. Requests from different applications and scenarios naturally come with different latency objectives. Online chatbots [6, 8] are interactive applications and are therefore with tight SLO constraints. On the contrary, offline applications, such as evaluation [51], scoring [36], or data wrangling [43], are less sensitive to latency. Such different latency objectives are also a consequence of the commercial purpose of earning more profits from LLMs via diversified service classes (*e.g.*, ChatGPT Plus [2]). However, existing LLM inference systems [34, 67] often treat all requests for a model equally and cannot differentiate their priorities, which has limitations in meeting different latency objectives of requests.

We introduce Llumnix, a new scheduling system for LLM serving that addresses the challenges above via *runtime rescheduling* of requests across model instances. Analogous to context switching across CPU cores in OS process management, rescheduling enables Llumnix to *react* to the unpredictable workload dynamics at runtime, instead of having to address all the complex scheduling concerns and tradeoffs with the one-shot dispatching of requests. Llumnix reschedules requests for multiple purposes (Figure 1): load balancing for reducing preemptions and interference, de-fragmentation for mitigating queuing delays, prioritization of urgent requests by creating even higher degree of isolation, saturating or draining out instances during auto-scaling more quickly.

Llumnix reschedules requests via an efficient and scalable *live migration* mechanism of requests along with their GPU memory states across instances. Straightforward rescheduling approaches could introduce substantial downtimes to rescheduled requests, especially for long sequences. By contrast, Llumnix introduces *near-zero* downtime that is *constant* to sequence lengths, by carefully coordinating the computation and the memory transfer to hide the cost.



(a) Load balancing (b) De-frag (c) Prioritization (d) Auto-scaling

Figure 1: Example rescheduling scenarios in Llumnix.

To exploit such great scheduling flexibility of migration, Llumnix adopts a distributed scheduling architecture that enables continuous rescheduling with high scalability. Llumnix further introduces a dynamic scheduling policy under this architecture that unifies all the rescheduling scenarios with different goals elegantly. This unification is achieved via a concept called *virtual usage*: Llumnix just needs to define a set of rules for setting the virtual usages of GPU memory for requests in different scenarios, and then use a simple load-balancing policy based on the virtual usages.

We have implemented Llumnix as a scheduling layer on top of inference engines. Llumnix currently supports a representative system, vLLM [34], as the underlying engine. Evaluation on a 16-GPU cluster using realistic workloads shows that Llumnix improves P99 first-token latency by up to 15× and P99 per-token generation latency by up to 2×, compared against a state-of-the-art scheduler INFaaS [53]. Llumnix also accelerates high-priority requests by 1.5×, and achieves 36% cost saving when delivering similar tail latencies.

In summary, this paper makes the following contributions.

- We reveal the unique characteristics and scheduling challenges of LLM serving that necessitate new scheduling goals such as isolation, de-fragmentation, and priorities.
- We propose request rescheduling as a key measure to achieve these goals and realize it with an efficient migration mechanism of requests and their GPU memory states.
- We design a distributed scheduling architecture and an accompanying scheduling policy that exploit request migration to achieve the multiple goals in a unified manner.
- We implement and evaluate Llumnix to show its advantages over state-of-the-art inference serving systems.

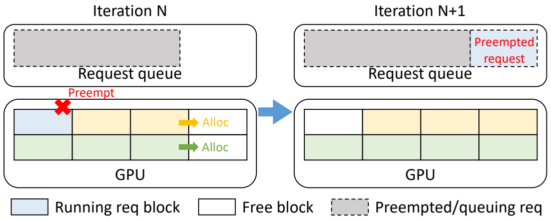


Figure 2: Request queuing and preemption using continuous batching and dynamic memory allocation.

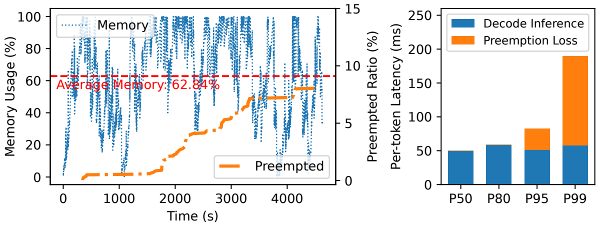


Figure 3: Request preemptions in LLaMA-7B serving.

2 Background

Application diversity of LLMs. Recent LLMs are becoming *task-agnostic*. That is, the same model can work for various tasks with context-specific inputs (*a.k.a.* the "prompts") provided. This is achieved by both increasingly larger model and dataset sizes and advanced pre-training approaches such as few-shot learning [14]. Task-agnostic models enable diverse applications, from chatbots, search engines, summarization, coding, AI assistants, to AI agents, to name a few.

The diverse applications lead to requests with different requirements for the serving. An important aspect is the *sequence lengths*. LLMs are racing to support longer sequence lengths — for example, from March to November 2023, the maximum sequence lengths of the GPT family have scaled from 32k [1] (GPT-4 [49]) to 128k (GPT-4 Turbo [50]). We expect this trend to continue as longer sequences are necessary for broader applications of LLMs. Consider an intuitive example of the tasks for summarizing and writing an article: they require sufficiently long input and output lengths, respectively. Another aspect is *expected latencies*. A real product example is that OpenAI introduces a subscription plan called ChatGPT Plus [2] to offer faster responses of common ChatGPT services. In general, different applications and situations also naturally have different levels of urgency. For example, more interactive applications like personal assistants expect shorter latencies than tasks like summarizing an article.

Autoregressive generation. The inference for state-of-the-art LLMs is *autoregressive*: the model iteratively accepts the input sequence plus all the previous output tokens to generate the next output token, until an "end-of-sequence" (EOS) token is generated. The phase for generating the first token and that for each new token afterwards are usually referred to as *prefill* and *decode*, respectively. LLM services typically return the generated tokens in a streaming manner. Therefore, the prefill and decode latencies are both user-perceivable and important to user experiences. The prefill latency determines how long it takes to start receiving the response, which can be dominated by the queuing delay. The decode latency determines the speed of receiving the following tokens subsequently.

During the autoregression, the intermediate results (key and value tensors used in the attention operation [59]) for each token are involved in the generation of all following tokens. Therefore, the inference engine typically stores these states in GPU memory for reuse, known as the *KV cache* [52].

Batching and memory management. State-of-the-art inference engines apply the *continuous batching* technique [34,67] to handle the varying sequence lengths and dynamic arrivals of requests. That is, a new/completed request can join/leave the running batch immediately, instead of waiting for all the running requests to complete. Batching also raises concern about memory management of KV cache. Since the memory demand of KV cache is not known a priori, it would clearly limit the batch size and batching benefits if the memory is reserved to the maximum length. For example, a LLaMA-2-13B [58] model supports sequence lengths up to 4k, which translates to 3.2 GB KV cache for a single request; while the memory of current GPUs remain tens of GBs, let alone the space for model weights (26 GB for LLaMA-2-13B). Therefore, recent work (vLLM [34]) proposed *dynamic memory allocation* for KV cache to increase batch size and throughput, enabled by a technique named PagedAttention: the KV cache tensors are stored in dynamically allocated blocks as the KV cache grows. Figure 2 presents an example of using continuous batching with dynamic memory allocation. The running requests are chosen based on the free memory blocks, hence there is a queuing request (the gray one) at iteration N as the memory is insufficient. At the next iteration, the system runs out of memory for the new blocks of the running requests. Therefore, the system preempts certain running requests (the blue one), which then goes back to the queue.

3 Motivation

We motivate the design of Llumnix with a series of key characteristics of LLM serving as follows.

Unpredictable memory demands and preemptions. With dynamic memory allocation, request preemptions are inevitable as a result of the unpredictable memory demands, which can significantly increase the latencies of the preempted requests. Figure 3 shows an experiment of LLaMA-7B model serving using vLLM on an A10 GPU running a trace of 2,000 requests generated from a Poisson distribution. The input and output lengths follow a power-law distribution with a mean

[1] 1k stands for 1,024 when describing sequence length in this paper.

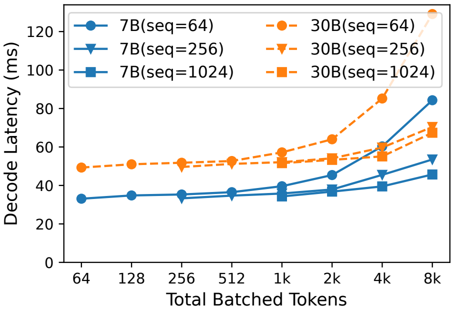


Figure 4: Latencies of one decode step of LLaMA-7B and LLaMA-30B with different sequence lengths and batch sizes.

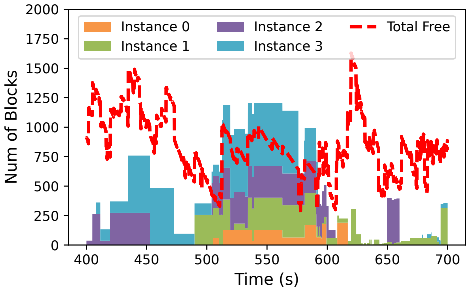


Figure 5: Total free memory vs. demands of the head-of-line queuing requests across four LLaMA-7B instances.

value of 256 tokens (details in §6). We control the request rate (0.42 req/s) to get a moderate memory load (62% on average) with some spikes due to the varying sequence lengths. Under such load, we still observe 8% of the requests being preempted. We quantify the preemption loss by measuring the latency penalty caused by preemption, including the extra queuing time and the recomputing for previous KV cache. We show different percentiles of per-token decode latency (averaged across all decode iterations of a request). We do not use the end-to-end latency because it depends on the number of iterations. We observe that the P99 per-token decode latency is much worse than the P50 (3.8×), and the preemption loss accounts for 70% for the P99 request. In particular, the P99 request experiences a total preemption loss of 50 seconds (preempted twice), showing severe service stalls and degradation of user experiences due to preemptions.

Performance interference among requests. We also observe performance interference of requests in a batch to each other, due to resource competition on GPU compute and memory bandwidth resources. Figure 4 shows the times for a decode step of LLaMA-7B (1-GPU) and LLaMA-30B (4-GPU) using different sequence lengths and batch sizes (the X-axis shows the total number of tokens in a batch for each data point). The decode speed decreases with more requests and higher interference, and the gap between the same sequence length is up to 2.6×.

Memory fragmentation. Considering the aforementioned problems, it would be better to spread requests across instances to reduce preemptions and interference. However, such spreading will make the available memory of the cluster fragmented across instances simultaneously. Here fragmentation refers to *external* fragmentation, *i.e.*, unallocated memory on an instance. Dynamic allocation techniques like PagedAttention [34] can eliminate external fragmentation during the *decode* phase, where the blocks are allocated one at a time. However, external fragmentation remains a significant problem for the *prefill* phase, which requires many blocks on an instance in one allocation to accommodate the KV cache of all tokens in the inputs. Therefore, external fragmentation can cause long queuing delays of new requests, especially those with long inputs.

Figure 5 shows an experiment of four LLaMA-7B instances, where the trace also uses the input/output length distribution with mean value 256 and a Poisson distribution with a request rate of 1.9 req/s. We implement a spreading dispatching policy that dispatches new requests to the instance with the lowest memory load for load balancing. We demonstrate the fragmentation by showing the total free memory blocks across the cluster, against the demand of the head-of-line queuing request on each instance. For most of the time span, the total free memory can accommodate the queuing requests on at least three instances (sometimes all of them). The request are queuing despite enough total memory because they exceed the free space on their own instances, which demonstrates the fragmentation and also the potential of defragmentation to reduce queuing delays.

Different emergency and priorities of requests. With requirements of products like ChatGPT Plus and the diverse application scenarios of LLMs, we foresee more applications with different latency sensitivities. However, existing systems usually treat all requests equally, where the latency-sensitive could easily be interfered by other normal ones, *e.g.*, excessive queuing delays or performance interference. This calls for a systematic approach to differentiating the request priorities for an LLM to meet their respective latency objectives.

Opportunity: request rescheduling across instances. This paper explores a new dimension that is missing in current LLM serving systems: the multiple model instances of a deployment and their interaction. A simple intuition is that when the aforementioned problems occur on a certain instance, it is possible that the whole cluster still has enough space for avoiding preempting requests, accommodating new requests, or mitigating interference. This is also a natural consequence of the varying request lengths and memory loads across instances. However, existing systems cannot exploit such free space on other instances because requests are tied on the same instance once scheduled throughout the autoregressive execution. Llumnix unifies the request scheduling component and the model inference engine to explore the potentials of fine-grained coordination among inference instances.

4 Llumnix Design

4.1 Overview

Llumnix builds upon the key idea of rescheduling LLM inference requests at runtime across model instances. Llumnix inherits continuous batching [67] and dynamic memory allocation [34] from state-of-the-art systems for high throughput. Beyond that, Llumnix exploits request rescheduling to react to the unpredictable workload dynamics in various situations with different scheduling goals, as illustrated in Figure 1.

A first goal is *load balancing* (Figure 1-a) to reduce request preemptions and interference on high-load instances. Although the dispatching can also consider load balancing of memory usage, it could be sub-optimal as the final memory usages of requests are unknown at the arrivals, due to the unpredictability of output lengths. Rescheduling complements it by reacting to the real usage growths of requests. Meanwhile, as shown before, load balancing can also lead to higher memory fragmentation and longer queuing delays of long inputs probably. Therefore, Llumnix also reschedules requests for *de-fragmentation* (1-b), *i.e.*, creating contiguous space on an instance by moving requests onto others. Although these two goals remain a tradeoff, Llumnix has a much larger space to balance them with rescheduling. Another goal is *prioritization* (1-c) of certain requests by rescheduling co-located requests away for lower load and avoiding interference. Such rescheduling provides "decicated" resources to high-priority requests dynamically, without the need for reserving machines statically. Finally, Llumnix also reschedules requests during *auto-scaling*, *e.g.*, to drain out an instance to be terminated (1-d) or saturate a new instance more quickly.

Realizing such highly dynamic rescheduling efficiently is challenging, considering the large request context states (*i.e.*, the KV cache). Naïve solutions include recomputing or copying the KV cache of the rescheduled requests, however with high computation stalls and downtime, reaching over 50× of the decoding cost (§6.2). What's more, the KV cache states increase with sequence lengths, limiting the scheduling flexibility under the trend of growing context lengths [50]. Such a high inference delay in generating next tokens greatly degrades the user experiences of LLM serving and thus prohibits request rescheduling. Llumnix addresses this challenge with a *live migration* mechanism that pipelines and coordinates the KV cache copying and the token generation computation, thereby bringing negligible downtime (§4.2).

To exploit the benefits of migration, Llumnix adopts a scalable architecture that combines global and local scheduling to decentralize the scheduling decisions and the coordinated migration actions, facilitating continuous rescheduling at scale (§4.3). Under this architecture, we further design an efficient heuristic scheduling policy that centers around the *virtual usage* concept to abstract the requirements of the different scheduling goals in a unified manner (§4.4).

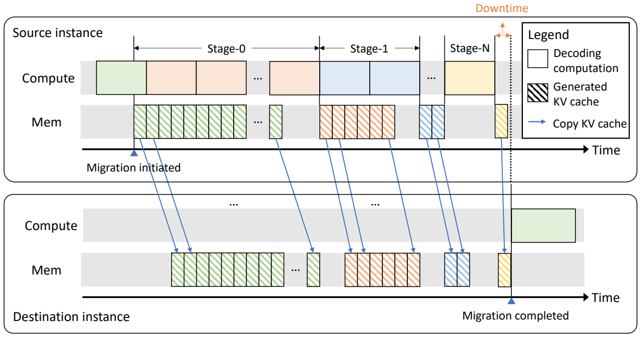


Figure 6: Llumnix adopts multi-stage migration to overlap the computation and KV cache copying for minimal downtime.

4.2 Live Migration of LLM Requests

The significant KV cache states of requests can potentially introduce great cost and serving stalls during rescheduling. Llumnix addresses this challenge by exploiting a key characteristic of LLM inference: the KV cache is *append-only*. LLM inference iteratively concatenates the output token of the current iteration with the input tokens, which is set as the input for the next iteration. In this way, inference engines also keep appending the calculated KV state of the current iteration to the KV cache parameters, leaving the parameters generated by previous iterations remain constant.

The live migration mechanism of Llumnix utilizes the inherent append-only characteristic of KV cache to pipeline the KV cache copying with the decoding computation. Because the KV cache already generated won't be modified in the following iterations, Llumnix can safely copy the KV cache of previous tokens in parallel with the computation for new tokens. In this way, Llumnix achieves *near-zero* and *constant* downtime to the rescheduled request. As shown in Figure 6, when migration is initiated, the source instance starts to copy the KV cache blocks of completed iterations, and continues the computation at the same time (stage 0). When the copying for the previous KV cache blocks is done, there will be a few more iterations (*i.e.*, blocks in Figure 6) computed in stage 0. Then, it switches to stage 1 to copy the KV cache generated by stage 0, while continuing the computation afterwards. The copying is generally much faster than the computation, thus the number of new blocks is typically small such that we can copy them in a very short period. To the end, only one iteration of computation is conducted for the KV cache migration (*i.e.*, stage-N). Therefore, Llumnix suspends the computation for the request by draining it out of the current batch and copies the remaining block, which introduces the downtime of this request. Once it is finished, the migration completes and the request resumes on the destination instance. Although the total copying duration of the whole sequence depends on the sequence length, the downtime for the request is only the period of copying the KV cache generated by one iteration,

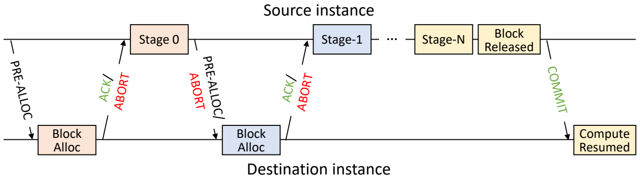


Figure 7: Handshake during migration.

which is negligible regardless of the sequence length.

The request migration approach of Llumnix borrows the key concept introduced in virtual machine (VM) live migration [17], which gradually reduces the working set to minimize the downtime. Llumnix does not require the dirty page tracing in VM migration as the working set (*i.e.*, KV cache) is append-only and does not change during migration. However, LLM serving further introduces additional challenges. Firstly, as both the source and destination instances are continually processing requests, the request might run out of memory during migration. Secondly, the request can complete in the middle of migration, due to the unpredictable execution (*i.e.*, generating EOS token) and the continuous batching [67]. To handle such exceptions and guarantee correctness during the asynchronous computation and memory copying, Llumnix introduces fine-grained coordination between the participating instances with a *handshake* process (Figure 7). Before each stage, the source instance issues a pre-allocate request with the number of blocks to migrate to make sure that the destination has enough space. The destination will try to allocate and reserve the blocks; if it succeeds or fails, the destination will notify the source to proceed or abort the migration and clean the states, respectively. Similarly, after each stage, the source instance also checks whether the request being migrated has completed or been preempted — if it has, the source will notify the destination to abort and release the reserved blocks; otherwise the source will go ahead to the next stage. The source or destination will also abort the migration if the other side fails. After the final stage finishes, the source releases its local blocks and notifies the destination to commit the migration and resume the execution of the request.

4.3 Distributed Scheduling Architecture

The live migration mechanism provides the foundation for runtime rescheduling of LLM inference requests. However, achieving fully dynamic scheduling is still non-trivial due to the higher scheduling pressure than in traditional schedulers. In particular, Llumnix would need to continuously track and reschedule every single running request throughout the cluster, rather than only dispatch incoming requests for one time or only manage running requests on one instance. This implies a higher scheduling frequency and a larger number of requests for the scheduler to track and schedule in each round.

Llumnix devises a scalable architecture that combines a cluster-level *global scheduler* and distributed instance-level schedulers, named *llumlet*s, to enable continuous rescheduling efficiently (Figure 8). Llumnix defines a clean separation of concerns with a narrow interface between the two levels. The global scheduler does not directly track or schedule the running requests; instead, it makes all scheduling decisions oriented to the *instances*, according to the *memory loads* of them. This way, the complexity of the global scheduler remains independent from the running requests, thereby preserving similar scalability to schedulers without dynamic scheduling. The loads are reported by the llumlets periodically, based on the request status and Llumnix's scheduling policy.

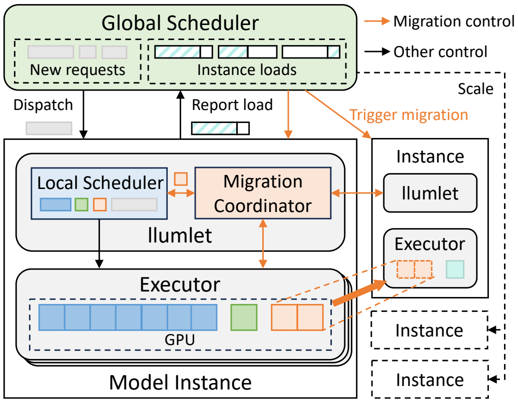


Figure 8: Llumnix architecture.

The global scheduler utilizes the load information to dispatch new requests, trigger migration across instances, and control the instance auto-scaling. In particular, for migration, the decisions are not made for specific requests; the global scheduler just pairs the source and destination instances, only based on the loads, and marks them as in the corresponding states to trigger the migration. The llumlets will decide the requests to migrate and execute the migration automatically.

The llumlet of each instance consists of a local scheduler and a migration coordinator. In addition to the functionalities of similar roles in existing systems like queuing, batching, and block management, an important new task of the local scheduler is to calculate the memory load of the instance. The load is not simply the physical memory being used; instead, it is a sum of the "virtual usages" (§4.4) of the requests. The local scheduler is also responsible for deciding the requests to migrate when triggered. Given the chosen requests, the migration coordinator will coordinate with the local scheduler and the other instance, and instruct the model executor to do the memory copying, as described before.

4.4 Dynamic Scheduling Policy

4.4.1 Goals and Definitions

Llumnix's scheduling policy is designed with the following goals. The first is to improve **prefill and decode latencies**, by reducing queuing delays, preemptions, and interference.

The second goal is **load-adaptivity** to handle varying cluster load and improve cost efficiency. We notice that the benefits of rescheduling is also relevant to cluster load, which could be limited under too high/low load. Llumnix incorporates instance auto-scaling to keep appropriate cluster load for both saving costs and maximizing the benefits of rescheduling.

Besides these two goals similar to those of existing systems, Llumnix introduces a new goal of request **priorities** that comes from the new requirements of LLMs. Priorities present a systematic approach for the same LLM to serve certain requests with higher emergency, *e.g.*, from ChatGPT Plus or more interactive applications. Llumnix provides applications with an interface for specifying request priorities to meet different SLOs, in terms of *scheduling priority* and *execution priority*. Requests with higher scheduling priorities will get scheduled earlier to reduce their queuing delays. Those with higher execution priorities will be given lower instance load and hence less interference to accelerate their execution. Currently, Llumnix supports two priority classes, high and normal, to demonstrate the ability of Llumnix to prefer high-priority requests, but our design also generalizes to more priorities.

4.4.2 Virtual Usage

To achieve the multiple goals above under the distributed scheduling architecture, Llumnix needs a scheduling policy that can express these goals using simple instance-level metrics, to improve the efficiency and scalability of the global scheduler. To this end, Llumnix introduces the *virtual usage* abstraction to unify these different, sometimes conflicting goals into a simple load metric of instances. The key observation here is that the aforementioned rescheduling scenarios fall into two categories: *load balancing*, and *creating free space on one instance* (de-fragmentation, prioritization, and draining out instances). We find that they can be unified into load balancing by assuming a virtual load on the instance: to create free space on an instance, we just need to set the virtual usages of certain requests to make the instance virtually overloaded, then a load balancing policy will be triggered to migrate the requests to other instances.

This observation leads us to a simple heuristic with load-balancing as the basis, combined with a set of rules for setting request virtual usages in different situations. We summarize the rules in the function `CalcVirtualUsage` in Algorithm 1 and illustrate example scenarios in Figure 9. In normal cases, the virtual usage of a request is just its physical memory usage to enable routine load balancing, as shown in Figure 9(a). We discuss the rules for other cases as follows.

Queuing requests. For the head-of-line queuing request on an instance, we assign a positive virtual usage to it to reflect its resource demand in terms of the required memory, although the physical usage is 0. Thus, queuing requests will increase the total virtual usage of the instance, then the policy will trigger migration for load balancing (which in effect is de-fragmentation for the queuing request), as shown in Figure 9(b). There could be a lot of heuristics to explore for setting the virtual usage, which controls the tradeoff between reducing queuing delays and load balancing — for example, gradually increasing the virtual usage of a queuing request until it reaches the real memory demand. Llumnix currently uses a simple rule that directly uses its real demand (line 4 in Algorithm 1), which favours reducing queuing delays. This rule is based on our observation that queuing delay can dominate the end-to-end latency and worth such preference. Our evaluation also shows that this rule preserves the benefits of load balancing, due to the high flexibility of migration.

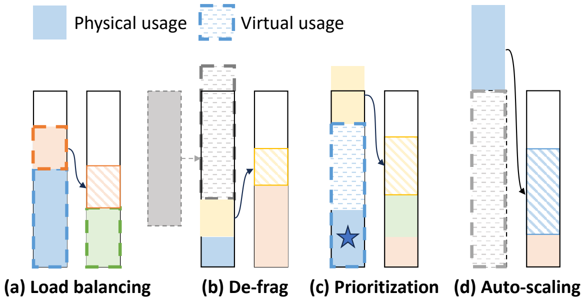


Figure 9: Llumnix combines virtual usages with a load-balancing policy to unify multiple scheduling goals.

Execution priorities. For a request with high execution priorities, Llumnix tries to prevent the instance the request is running on from exceeding a given level of real load, by reserving a memory space as headroom, as shown in Figure 9(c). This is achieved by adding such a headroom on the physical usage of a high-priority request to get the virtual usage (line 8). When there are multiple high-priority requests on an instance, this headroom is divided among them (line 10). The headroom for high-priority requests is currently defined as that required to preserve the ideal decode speed (*i.e.*, no visible interference), which is obtained through profiling. The headroom for normal requests is 0. Llumnix can also support more execution priorities by specifying the sizes for the headroom. When the headroom for a high-priority request is running up, the other normal requests will be migrated away by the load balancing policy because the instance is overloaded in terms of the total virtual usage.

Auto-scaling. When a new instance is launched, Llumnix's load balancing policy will automatically saturate it by migrating requests from other instances to it. When an instance is terminating, we artificially add a fake request with a virtual usage of infinity on it (line 7), then the remaining requests will be migrated to other instances, as shown in Figure 9(d).

4.4.3 Policies

We then describe how the specific scheduling decisions are made based on the virtual usages.

Algorithm 1: Virtual Usage and Freeness Calculation

```
Function CalcVirtualUsage(req, instance):
    if $req.isQueuing$ then
        if $req.isHeadOfLine$ then
            return $req.demand$
        return 0
    if $req.isFake$ then
        return $\infty$
    return $req.physicalUsage$+GetHeadroom($req.priority, instance$)
Function GetHeadroom(p, instance):
    return $headroomForPriority[p]/instance.numRequests[p]$
Function CalcFreeness(instance):
    if $instance.isTerminating$ then
        AddFakeReq($instance.requests$)
    $totalVirtualUsages = 0$
    for $req$ in $instance.requests$ do
        $totalVirtualUsages+=$CalcVirtualUsage($req, instance$)
    $freeness = (instance.M - totalVirtualUsage)/instance.B$
    return $freeness$
```

Dispatching. Llumnix dispatches new requests with higher scheduling priorities first. Within the same priority, it adopts a simple first-come-first-serve order. On each instance, requests are scheduled in the same order. Llumnix uses a load-balancing policy that dispatches each request to the freest instance. We introduce a metric for measuring the *freeness* of an instance defined as $F = (M - \sum V)/B$, where M is the total memory, V is the virtual usage of each request, and B is the batch size. While $(M - \sum V)$ already measures the free space, we divide it by the batch size because it determines the consumption speed, *i.e.*, the number of new tokens per iteration. Thus the metric suggests how many iterations the batch can still run for. Then Llumnix dispatches each incoming request to the instance with the highest freeness. Because the virtual usage of a request can be larger than the physical, it is possible that F is a negative value, *e.g.*, when there are queuing requests or high-priority requests. Such negative freeness values help Llumnix automatically treat such instances as overloaded and prefer dispatching requests to other instances. The freeness metric also guides the migration and auto-scaling, as shown later.

Migration. Llumnix triggers the migration policy periodically. In each round, Llumnix selects the candidate sets of source and destination instances by choosing those with freeness values smaller or greater than given thresholds, respectively. Llumnix pairs the instances from both sets by picking the two with the lowest and the highest freeness values repeatedly, and then sets them in corresponding states. The llumlet of each source instance then starts to migrate requests to the destination continuously, until it is no longer set in the source state. The llumlet prefers the requests with lower priorities and shorter sequence lengths when choosing the requests to migrate. In the next round, if an instance during migration is no longer beyond the thresholds, Llumnix will unset the migration state and the migration will stop.

Auto-scaling. Llumnix scales the instances according to the cluster load in terms of the averages freeness for the normal priority across instances. The policy maintains the average freeness within a range $[x, y]$, and adds or terminates an instance when the freeness is smaller than x or greater than y for a period, respectively. Llumnix chooses the instance with fewest running requests for termination.

5 Implementation

We implement Llumnix with 3,300 lines of Python code. Llumnix is a standalone library comprising both its own components and an interface to integrate and communicate with backend inference engines. This architecture makes Llumnix non-intrusive and extensible to different backends. Llumnix currently supports vLLM [11] as the backend, which is an open-source state-of-the-art inference engine that features continuous batching, PagedAttention, and tensor-parallel distributed inference [34, 56].

Multi-instance serving. Llumnix instantiates the multiple instances of the backend and the other components as Ray [42] actors. Ray's Python-native distributed runtime enables fine-grained coordination among these actors in a simple and efficient manner. Llumnix also launches a set of request frontend actors that exposes an OpenAI-style API endpoint [48]. Although a request can be migrated across backend instances, the generated tokens are forwarded to the frontend and then returned to end users, ensuring a steady API service.

KV cache transfer. We use the Gloo collective communication library [5] (the `Send`/`Recv` primitives) for the KV cache transfer during migration. A potential alternative is NCCL [1], which is generally faster than Gloo on GPUs but has been adopted in communication for distributed inference. However, Llumnix needs to migrate requests in parallel with the inference to minimize the downtimes, but concurrent invocations of NCCL are known to be unsafe [45]. The pipelined migration design allows us to use Gloo while maintaining negligible downtimes. Using Gloo needs to copy the KV cache between CPU and GPU memory, which is done in another CUDA stream to avoid blocking the inference computation. Note that in typical deployments, the communication-heavy tensor parallelism is limited in a single machine for high-speed transfer [44]. In such cases, migration between instances (machines) will not interfere with the tensor-parallel inference.

Block fusion. vLLM stores the KV cache in non-contiguous small blocks that are dynamically allocated. For example, the block size of a 16-bit LLaMA-7B model is 128 KB (for key or value tensors of 16 tokens in each layer), and a sequence of 1k tokens translates to 4k such blocks (32 layers). To avoid the overhead of sending these blocks using many small messages, we fuse the blocks by copying them from GPU memory to a contiguous CPU memory buffer and use Gloo to send the buffer as a whole, thereby improving the transfer efficiency.

Fault tolerance. Llumnix provides fault tolerance for each component to ensure high service availability. When the global scheduler fails, Llumnix temporarily falls back to a scheduler-bypassing mode, thus not affecting the service availability: that is, the request frontends directly dispatch requests to certain instances using simple rules, and migration is disabled. When an instance (or the co-located llumlet) fails, the requests running on it will be aborted. In particular, ongoing migration on failed instances will also be aborted (the request being migrated is not necessarily aborted, depending on if its source instance is healthy), which is handled by the handshake process. These failed actors will be automatically restarted by Ray, after which the service could go back to normal state.

6 Evaluation

We evaluate Llumnix on a 16-GPU cluster using realistic models and various workloads. Overall, our key findings include:

- Llumnix introduces near-zero downtime to requests being migrated and near-zero overhead to other running requests.
- Llumnix improves prefill latencies by up to 15×/7.7× (P99/mean) over INFaaS on 16 LLaMA-7B instances via de-fragmentation. Llumnix also improves P99 decode latency by up to 2× by reducing preemptions.
- Llumnix improves high-priority request latencies by up to 1.5× by reducing their queuing delays and accelerating their execution, while preserving similar performance of the normal requests.
- Llumnix achieves up to 36% cost saving while preserving similar P99 latencies with efficient auto-scaling.

6.1 Experimental Setup

Testbed. We use a 16-GPU cluster with 4 GPU VMs on Alibaba Cloud (type `ecs.gn7i-c32g1.32xlarge`), each with 4 NVIDIA A10 (24 GB) GPUs connected via PCI-e 4.0, 128 vCPUs, 752 GB memory, and 64 Gb/s network bandwidth.

Models. We conduct experiments using a popular model family, LLaMA [57]. We test two different specifications: LLaMA-7B, which runs on a single GPU, and LLaMA-30B, which runs on 4 GPUs of a machine using tensor parallelism. The models adopt the commonly used 16-bit precision. The version of vLLM that we based on only supports the original LLaMA with a maximum sequence length of 2k, but there have been a series of recent LLaMA variants supporting longer sequence lengths ranging from 4k to 256k [3, 7, 58, 65]. Since the model architectures and inference performance of these variants are mostly similar to those of LLaMA, we believe that our results are representative of more model types and larger sequence length ranges from a systems perspective.

Traces. Similar to prior work [34, 35, 67], we synthesize request traces to asses Llumnix's online serving performance.

Distribution			Mean	P50	P80	P95	P99
Real	ShareGPT	In	306	74	348	1484	3388
		Out	500	487	781	988	1234
	BurstGPT	In	830	582	1427	2345	3549
		Out	271	243	434	669	964
Gen	Short (S)		128	38	113	413	1464
	Medium (M)		256	32	173	1288	4208
	Long (L)		512	55	582	3113	5166

Table 1: Real and generated distributions of sequence lengths (numbers of tokens) used in our evaluation. The real distributions include those of both inputs ("In") and outputs ("Out").

We use Poisson and Gamma distributions with different request rates (requests per second) to generate request arrivals. For Gamma, we also use varying coefficients of variance (CVs) to adjust the burstiness of the requests. Each trace has 10,000 requests. We choose an appropriate range of request rates or CVs for the traces to maintain the loads within a reasonable range: nearly no queuing delays and preemptions for P50 requests, and queuing delays within a few tens of seconds for P99 requests when using Llumnix.

For the input/output lengths of requests, we use two public ChatGPT-4 conversation datasets, ShareGPT (GPT4) [10] and BurstGPT (GPT4-Conversation) [62], for an evaluation on real workloads. Considering that Llumnix targets more diversified applications, we also use generated power-law length distributions to emulate long-tail workloads that mix both frequent, short sequences (*e.g.*, for interactive applications like chatbots and personal assistants) and seldom, long sequences (*e.g.*, summarizing or writing articles). We generate multiple distributions with different long-tail degrees and mean lengths (128, 256, 512), as shown by the Short (S), Medium (M), and Long (L) distributions in Table 1. These distributions have a maximum length of 6k, thus the total sequence length of a request (input plus output) will not exceed the capacity of an A10 GPU when running LLaMA-7B (13,616 tokens). To observe the performance with different workload characteristics, we construct the traces by picking different combinations of the length distributions for inputs and outputs as follows: S-S, M-M, L-L, S-L, and L-S.

Baselines. We compare Llumnix with the following schedulers. All the baselines and Llumnix use vLLM as the underlying inference engine to focus the comparison on the request scheduling across instances.

- *Round-robin dispatching*: a simple dispatching policy to distribute requests across instances evenly, which is a typical behavior of production-grade serving systems [4, 9, 47].
- *INFaaS++*: an optimized version of INFaaS [53], a state-of-the-art scheduler for multi-instance serving. We evaluate its load-balancing dispatching and load-aware auto-scaling policies. We improve it by making it focus on the GPU

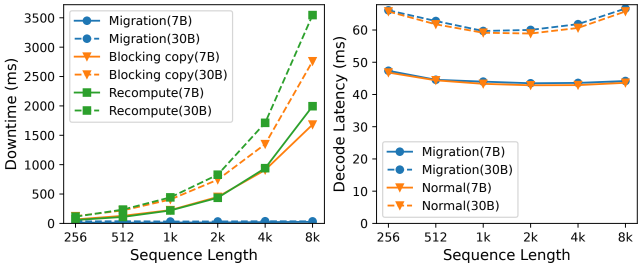


Figure 10: Downtime and overhead of migration.

memory load as it is the dominant resource in LLM serving. This load also counts in the memory required by queuing requests on each instance to reflect the queue pressure.

- *Llumnix-base*: a base version of Llumnix that is priority-agnostic (*i.e.*, treats all requests as the same priority) but enables all the other features including migration.

Key metrics. We focus on request latency, in terms of end-to-end, prefill (that of the first generated token), and decode (that since first generated token to the last, averaged over all generated tokens). We report both mean and P99 values.

6.2 Migration Efficiency

We first examine the performance of Llumnix's migration mechanism, in terms of the downtimes introduced to the migrated requests and the performance overheads for the running requests. We test both the 1-GPU LLaMA-7B and the 4-GPU LLaMA-30B models. For each model, we deploy two instances on two different machines. We use different sequence lengths, for each of which we run a batch of requests with the same total length of 8k on both instances. We migrate one of the requests from one instance to another and measure its downtime and the decode speeds of the running batches on both instances during migration.

We compare the downtime during migration with two simple approaches: recomputing, and blocking copying of the KV cache using Gloo (non-blocking for other requests). As shown in Figure 10 (left), the downtime of migration is nearly constant with increasing sequence lengths (roughly 20-30 ms), even shorter than a single decode step. In comparison, the downtimes of baselines increase with the sequence lengths, reaching up to 111× that of migration. For example, recomputing an 8k sequence for LLaMA-30B takes 3.5s, which translates to a service stall similar to 54 decode steps. We also notice that for all sequence lengths, the migration only takes two stages, which is the minimum. This is because the data copying is sufficiently fast and the number of new tokens generated during the first stage is small.

Figure 10 (right) also compares the per-step decode times during migration on the source instance with that during normal execution (results on the destination are mostly similar). We observe up to 1% performance differences for both LLaMA-7B and LLaMA-30B, showing the negligible migration overhead. Also note that such overhead exists only when there are requests being migrated (in or out) on an instance. We find that in all the serving experiments in the following sections, the average fraction of time span with ongoing migration for each instance is only roughly 10%. This implies an effective overhead that is even much smaller, which is worthwhile for the great scheduling benefits of migration.

6.3 Serving Performance

We evaluate the scheduling performance of Llumnix in online serving using 16 LLaMA-7B instances (auto-scaling is disabled except in experiments in §6.5).

Real datasets. We first compare Llumnix with round-robin and INFaaS++ using the ShareGPT and BurstGPT traces (the top two rows in Figure 11). Llumnix outperforms the baselines in end-to-end request latency by up to 2× and 2.9× for mean and P99, respectively. In particular, we observe that round-robin always performs much worse than both INFaaS++ and Llumnix: since the sequence lengths have high variance, simply distributing requests evenly can still lead to unbalanced load, impacting both prefill and decode latencies. Llumnix achieves significant gains in prefill latency over round-robin, by up to 26.6× for mean and 34.4× for P99. This is because round-robin can possibly dispatch new requests to overloaded instances, leading to long queuing delays. Llumnix also improves P99 decode latency by up to 2×, by load balancing to reduce preemptions. This margin seems smaller as the latency penalty caused by preemptions is averaged over all generated tokens. However, whenever preemption occurs, it results in a sudden service stall, which impacts user experience. Figure 11 (the rightmost column) reports the preemption loss in terms of the extra queuing and recomputing times (mean value of all requests). Llumnix reduces preemption loss by 84% on average compared to round-robin. These results highlight the importance of load balancing in LLM serving. In the following experiments using generated distributions with higher variance, round-robin showed up to two orders of magnitude worse latencies. Therefore, we omit it for the other traces for clarity of the figures and focus on the comparison between INFaaS++ and Llumnix.

Llumnix outperforms INFaaS++ in mean and P99 prefill latencies by up to 2.2× and 5.5×, and P99 decode latencies by up to 1.3×, respectively, showing the extra benefits of migration, beyond dispatch-time load balancing. Next we use more traces with different characteristics to further evaluate them for a deeper understanding of the improvements.

Generated distributions. We compare Llumnix and INFaaS++ using multiple generated distributions (bottom five rows in Figure 11). Llumnix outperforms INFaaS++ across all traces in end-to-end request latency by up to 1.5× and 1.6× for mean and P99, respectively. For prefill, the improvements are up to 7.7× for mean and 14.8× for P99. Despite dispatch-

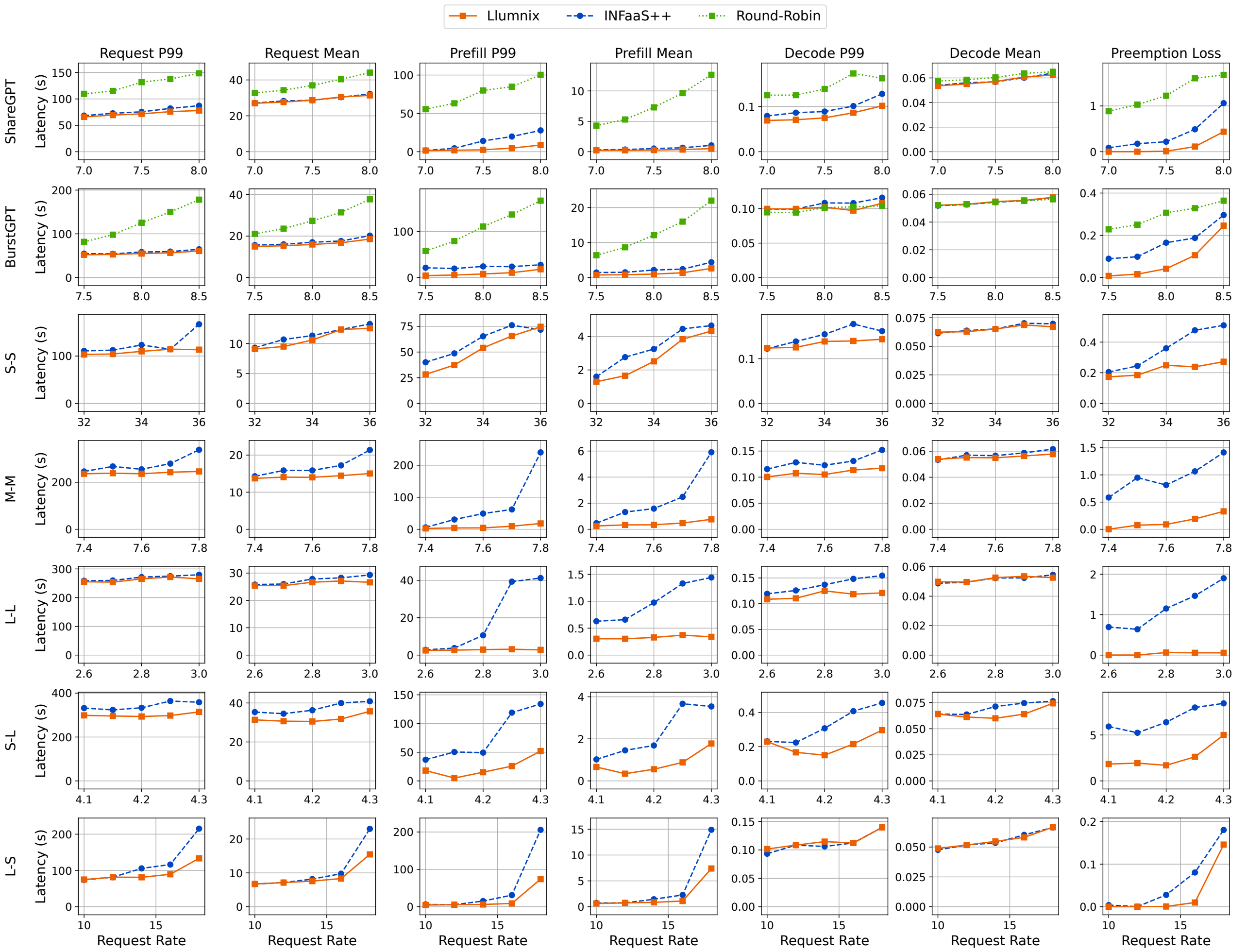


Figure 11: Request end-to-end, prefill, and decode latencies and preemption loss of serving 16 LLaMA-7B instances. Each row shows a set of experiments using a trace with a specific sequence length distribution, as annotated on the Y-axis labels.

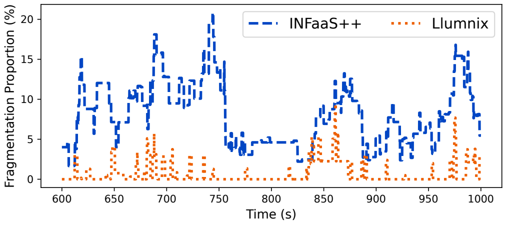


Figure 12: Memory fragmentation over time.

ing requests to instances with the lowest load, INFaaS++ can still exhibit long queuing delays due to fragmentation, especially for the long-tail requests with long inputs. Llumnix uses migration for de-fragmentation to reduce such queuing delays, showing more gains in traces with more long inputs.

To take a closer look at the memory fragmentation, we further present a case study on the experiment of the M-M trace with the request rate of 7.5. We define the fragmented memory at each moment as the portion of cluster free memory that could satisfy the demands of the head-of-line blocking requests across all instances, if no fragmentation. For example, if the total free memory is 8 GB, with three head-of-line blocking requests each requiring 3 GB, then the fragmented memory is counted as 6 GB, *i.e.*, this 6 GB memory could satisfy two queuing requests if no fragmentation. This metric suggests the memory space wasted due to fragmentation. We report the proportion of fragmented memory in the cluster total memory. In the example, if the total memory is 16 GB, then the proportion is 37.5% (6/16). Figure 12 shows the fragmentation proportion of the experiment during a busy period. We observe that INFaaS++ often shows higher than 10% fragmentation, wasting a significant amount of cluster memory. In comparison, the fragmentation is often 0 in Llumnix. The average values during this period are 0.7% and 7.9% for Llumnix and INFaaS++ respectively (92% reduction), highlighting the effect of de-fragmentation using migration.

Llumnix also improves the P99 decode latency by up to

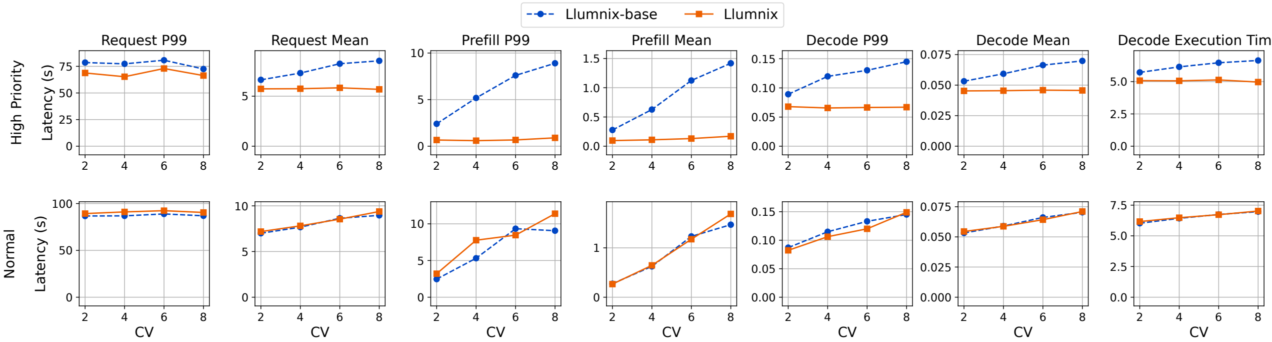


Figure 13: Performance of high-priority and normal requests, as annotated on the Y-axis labels.

2×, through migration to reduce preemptions. Although INFaaS++ already implements load balancing in dispatching to reduce preemptions, migration complements it by reacting to the real sequence lengths, which are unknown at request arrivals. As shown in Figure 11, Llumnix significantly reduces the preemption loss, in many cases down to near zero. The reduction is 70.4% on average across all experiments, which translates to an average reduction of 1.3 seconds in the end-to-end request latency.

6.4 Support for Priorities

We evaluate the support for priorities of Llumnix by randomly picking 10% of the requests and assigning high scheduling and execution priorities. We use traces with the Short-Short length distribution and Gamma arrival distribution. We vary the CV parameter to show the interference to high-priority requests due to bursty workloads and load spikes. We empirically choose a target memory load of 1,600 tokens for high-priority requests, as we observe that such load preserves near-ideal decode speed (refer to Figure 4). Llumnix translates this target load to the corresponding memory headroom for high-priority requests. We compare Llumnix with Llumnix-base, which simply treats all requests as the same priority.

As shown in the first row in Figure 13, Llumnix improves mean request latencies for the high-priority by 1.2× to 1.5× with increasing CVs. Higher CVs leads to more high-load periods, where high-priority requests can suffer more interference if not protected. Even with higher CVs, Llumnix still delivers similar latencies of high-priority requests, showing the isolation Llumnix provides to such requests. This is because Llumnix can handle changing high-priority loads by dynamically creating space for them, which is difficult in approaches like static resource reservation. For prefill latencies, Llumnix shows 2.9× to 8.6× gains for the mean, and 3.6× to 10× for the P99, respectively. This is achieved by reducing the queuing delays with high scheduling priorities. Llumnix also improves decode latencies by 1.2× to 1.5× for the mean and 1.3× to 2.2× for the P99, respectively. This improvement comes from the acceleration of the decode computation by giving lower instance loads and interference to high execution priorities, shown by the similar gains in the average decode computation time (the rightmost column). We also notice that Llumnix preserves similar performance of the normal requests (the second row in Figure 13): Llumnix increases the mean request, prefill, and decode latencies of normal requests by up to 4.5%, 13%, and 2%, respectively.

6.5 Auto-scaling

We evaluate the auto-scaling capability of Llumnix using larger ranges of request rates and Gamma CVs to show the adaptivity to load variation. By default, Llumnix uses a scaling threshold range of [10, 60], *i.e.*, Llumnix scales instances up or down when the average freeness is under 10 or above 60; recall that this metric represents the most decode steps an instance can still run for given the current batch. We let INFaaS++ use the same scaling strategy, thus both Llumnix and INFaaS++ have the same degree of aggressiveness of scaling up instances. We use a maximum instance number of 16 and the Long-Long sequence length distribution.

We first vary the request rates using Poisson distribution. As shown in the first row of Figure 14, Llumnix consistently achieves latency improvements across all request rates, *e.g.*, up to 12.2× for P99 prefill latency. We also measure the resource cost in terms of average instances used, shown in the rightmost column. Llumnix saves costs by up to 16%, because Llumnix increases the auto-scaling efficiency by saturating or draining out instances more quickly. We also test different workload burstiness with varying CVs of Gamma distribution (request rate = 2). As shown in the second row, Llumnix shows similar improvements in latencies and costs, *e.g.*, up to 11× for P99 prefill latency and 18% for the cost.

Finally, we examine the cost efficiency of Llumnix in terms of how aggressively Llumnix needs to scale out instances to preserve a certain latency objective, *e.g.*, a given P99 prefill latency. We vary the scaling up threshold *t*, and the scaling threshold range is determined as [*t*, *t*+50]. Higher values of *t* means that Llumnix tends to use more instances. Figure 15 shows the P99 prefill latencies and costs with different scaling

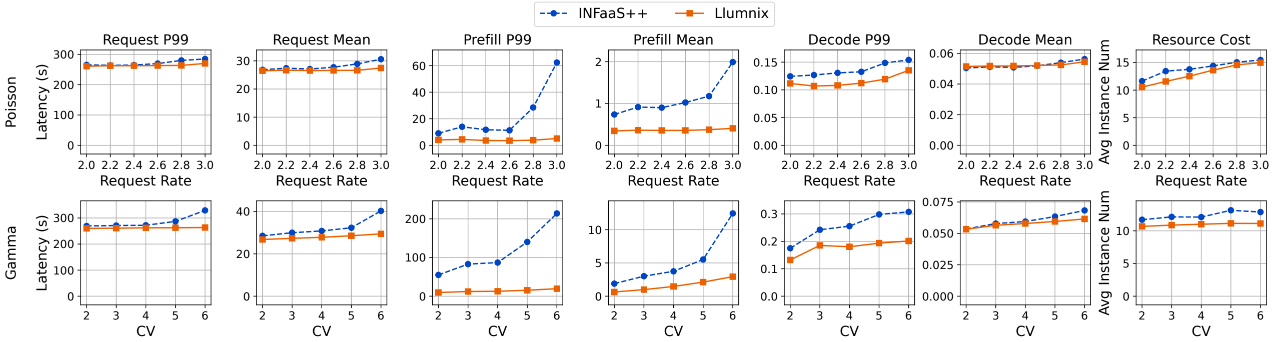


Figure 14: Auto-scaling of LLaMA-7B instances with Poisson and Gamma distributions, as annotated on the Y-axis labels.

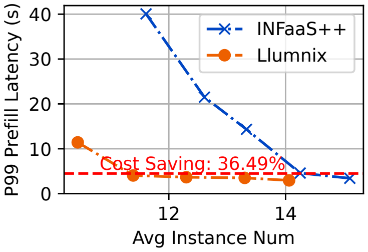


Figure 15: P99 prefill latencies vs. average numbers of instances with varying scaling thresholds.

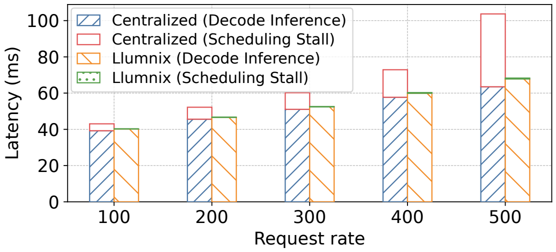


Figure 16: Per-token latencies and scheduling stalls under increasing request rates using 64 LLaMA-7B instances.

thresholds. We observe that Llumnix achieves similar P99 prefill latency (roughly 5s, the red dash line) while saving 36% of the cost compared to INFaaS++, as a result of the combination of the ability to reduce queuing delays via migration and the higher auto-scaling efficiency.

6.6 Scheduling Scalability

We conduct a scheduling stress test to examine the scalability of Llumnix with 64 LLaMA-7B instances using higher request rates. Since this cluster exceeds the size of our testbed, we replace the real GPU execution in vLLM with a simple `sleep` command, whose duration is determined by offline measurement on A10 GPUs with different sequence lengths and batch sizes. We build a simple centralized scheduler as the baseline by extending the vLLM scheduler to manage all requests across all instances. We issue requests with input and output lengths of 64 tokens with increasing request rates.

As shown in Figure 16, with increasing request rates, the baseline experiences scheduling stalls during the inference computation of up to 40ms per iteration, translating to 1.7× slowdown. Such stalls are a result of the communication between instances and the centralized scheduler synchronizing request statuses and scheduling decisions, which becomes a bottleneck under high load. By contrast, Llumnix exhibits near-zero scheduling stalls even under high request rates, showing the scalability of the distributed scheduling architecture. Llumnix offloads and distributes the intra-instance scheduling logic across llumlets so that it is done in parallel and asynchronously with the global scheduling. Moreover, llumlets only report instance-level metrics, instead of the precise status of every single request, further improving the communication efficiency.

7 Related Work

LLM inference. As transformer models show significance in model serving, recent works, such as FasterTransformer [46], TurboTransformer [25], LightSeq [61], and FlashAttention [21, 22], optimize GPU kernels to improve the inference performance. SpotServe [41] supports LLM inference using preemptible instances for improving cost efficiency. FastServe [63] optimizes request completion times using a preemptive time-slicing approach. AlpaServe [35] exploits pipeline parallelism to reduce serving latency for bursty workloads. To further increase the GPU utilization and serving throughput, Orca [67] proposes iteration-level scheduling (referred to as continuous batching in recent works and this paper) and selective batching, while vLLM [34] optimizes the memory usage with PageAttention. [55] proposes fair scheduling of requests on an LLM instance. Prior works mostly target solo-instance serving, therefore complementing to Llumnix. Llumnix explores the challenges and opportunities of deploying multi-instance LLM serving. The key append-only characteristic of KV cache is exploited to enable migration capability of requests in the inference engine. Such a mechanism opens great policy design space to offer priority and performance isolation, improve memory efficiency, and enable instance auto-scaling. We also plan to explore the interplay between the global scheduling across instances with local scheduling techniques inside each instance (*e.g.*, preemptive [63] and fair [55] scheduling) as future works.

Request scheduling. To support deep learning model deployment, numerous systems (*e.g.*, Clipper [19], Nexus [54], DVABatch [20], and TritonServer [47]) have been proposed to optimize request scheduling for DNN inference serving. To meet the SLOs of DNN inference requests, Clockwork [29] utilizes the execution predictability of traditional DNNs, while Reef [33] and Shepherd [68] perform preemptions to serve high-priority requests. AlpaServe [35] uses a simple load-balancing dispatching policy based on queue lengths. These works mostly focus on traditional DNN model serving, where a request requires only one-time inference on the model. However, LLM inference service requires autoregressive computation on models for unpredictable numbers of iterations and introduces intermediate states (*i.e.*, KV cache), showing brand new characteristics. DeepSpeed-MII [4], albeit targeting multi-instance LLM serving, uses a simple round-robin dispatching policy that ignores LLM characteristics. Llumnix steps further to incorporate request migration and ensures high throughput and low latency, provides SLO for prioritized requests, and auto-scales instances for resource efficiency with a unified load-aware dynamic scheduling policy.

Beyond multiple model instances, INFaaS [53] further supports scheduling across multiple model types/variants, considering the performance and accuracy requirements in difference applications. This is also a typical scenario for LLMs: for example, fine-tuned models for a specific task (*e.g.*, coding [3, 13, 30]); variants with different sizes or precisions ([26, 37, 39]) of the base LLM. We plan to extend Llumnix to support multiple model types in future work, considering the larger tradeoff space of latency/throughput and accuracy.

Isolation vs. fragmentation. The tradeoff between isolation and fragmentation, or that between workload packing and spreading, have been a classic scheduling challenge. That is, workload packing improves resource utilization, at the expense of potential interference between co-located workloads; spreading workloads, on the contrary, provides better isolation but also increases resource fragmentation. Many research efforts have been devoted to better balancing isolation and fragmentation in datacenters for big-data jobs and virtual machines, by identifying the interference-sensitivity of workloads and optimized scheduling policies ([16, 18, 23, 24, 27, 31, 32, 40, 60, 66]). This challenge was also identified in GPU clusters for deep learning workloads. Amaral et al proposed a topology-aware placement algorithm to address the tradeoff between packing and spreading deep learning training jobs on multi-GPU servers [12]. Gandiva [64] addresses the heterogeneous sensitivity to packing/spreading of different jobs with introspective job migration. This challenge becomes more complex for LLM serving due to the unpredictable autoregressive execution. Llumnix exploits request migration at runtime to react to the workload dynamics to better reconcile these two goals.

Migration. Gandiva [64] enables introspective migration for deep learning training jobs during scheduling. It utilizes the inherent iterative behavior of deep learning, and conducts checkpoint-resume approach on the minimal working set (*i.e.*, mini-batch boundary) to migrate model weights. Even though LLM inference is iterative as well, directly migrating the entire states of a request is unacceptable, because the latency SLO of an inference request is crucial. Moreover, the working set per request is linear to the sequence length, which can be considerable given the trend of longer contexts [49, 50]. The migration approach in Llumnix is inspired by virtual machine live migration [17]. By carrying out the majority of migration while LLM requests continue decoding tokens on GPUs, Llumnix minimizes the downtime of request migration, making the cost negligible regardless of the sequence lengths.

8 Conclusion

Llumnix, as implied by the name, represents our vision of serving LLMs as Unix. This vision originates in the observation that LLMs and modern operating systems have common natures such as the universality, multi-tenancy, and dynamism, and hence share similar requirements and challenges. This paper takes an important step towards this vision by drawing lessons from conventional OS wisdom including: definition of classic abstractions like isolation and priorities in the new context of LLM serving; implementation of the “context switching” as the key approach with inference request migration; and continuous, dynamic request rescheduling exploiting the migration. All these combined, Llumnix delivers better latency, cost efficiency, and support for differentiated SLOs, pointing to a new way of LLM serving.

Acknowledgement

We thank the anonymous OSDI reviewers and our shepherd for their valuable feedback, which helped improve the presentation of this paper. We thank Shiru Ren for early discussion on VM live migration techniques. This work was supported by Alibaba Group through the Alibaba Research Intern Program. This work was also supported by National Key Research and Development Program of China (Grant No. 2023YFB3001801), National Natural Science Foundation of China (Grant No. 62322201, 62072018, U23B2020 and U22A2028), Fundamental Research Funds for the Central Universities (YWF-23-L-1121), and State Key Laboratory of Software Development Environment (SKLSDE-2023ZX-05).

References

[1] Nccl, https://developer.nvidia.com/nccl/.

[2] Chatgpt plus, 2023. https://openai.com/blog/chatgpt-plus.

[3] Code llama, 2023. https://github.com/facebookresearch/codellama.

[4] Deepspeed-mii, 2023. https://github.com/microsoft/DeepSpeed-MII.

[5] Gloo collective communication library, 2023. https://github.com/facebookincubator/gloo.

[6] Google bard, 2023. https://bard.google.com/.

[7] Longllama, 2023. https://github.com/CStanKonrad/long_llama.

[8] Openai chatgpt, 2023. https://chat.openai.com/.

[9] Ray serve, 2023. https://docs.ray.io/en/latest/serve/index.html.

[10] Sharegpt_gpt4, 2023. https://huggingface.co/datasets/shibing624/sharegpt_gpt4.

[11] vllm. https://github.com/vllm-project/vllm, 2023.

[12] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17, New York, NY, USA, 2017. ACM.

[13] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[16] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[17] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[20] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, Carlsbad, CA, July 2022. USENIX Association.

[21] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691, 2023.

[22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[23] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: An efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 389–402, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

[27] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the thirteenth EuroSys conference*, pages 1–13, 2018.

[28] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[30] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[31] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[32] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. Interference management for distributed parallel applications in consolidated clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 443–456, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[36] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yüksekgönül,

Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *CoRR*, abs/2211.09110, 2022.

[37] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024.

[38] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023.

[39] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits, 2024.

[40] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[41] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances, 2023.

[42] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[43] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. Can foundation models wrangle your data? *Proc. VLDB Endow.*, 16(4):738–746, 2022.

[44] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[45] NVIDIA. Using multiple nccl communicators concurrently. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html#using-multiple-nccl-communicators-concurrently.

[46] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

[47] NVIDIA. Triton inference server. https://github.com/triton-inference-server/server, 2023.

[48] OpenAI. Openai api, 2020. https://openai.com/blog/openai-api.

[49] OpenAI. Gpt-4 technical report, 2023.

[50] OpenAI. Gpt-4 turbo. https://help.openai.com/en/articles/8555510-gpt-4-turbo, 2023.

[51] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.

[52] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[53] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[54] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models, 2023.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[57] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[58] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18, New York, NY, USA, 2015. ACM.

[61] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. In Young-bum Kim, Yunyao Li, and Owen Rambow, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 113–120. Association for Computational Linguistics, 2021.

[62] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.

[63] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[65] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, Madian Khabsa, Han Fang, Yashar Mehdad, Sharan Narang, Kshitiz Malik, Angela Fan, Shruti Bhosale, Sergey Edunov, Mike Lewis, Sinong Wang, and Hao Ma. Effective long-context scaling of foundation models, 2023.

[66] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 607–618, New York, NY, USA, 2013. Association for Computing Machinery.

[67] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[68] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

A Artifact Appendix

Abstract

This artifact includes the source code and scripts to run the experiments and reproduce the evaluation results of this paper.

Scope

The artifact can be used to reproduce the results of the following experiments.

- Migration efficiency: Figure 10.
- Serving performance: Figure 11.
- Support for priorities: Figure 13.
- Auto-scaling: Figure 14 and Figure 15.

Contents

This artifact includes the following contents.

- Source code of a prototype implementation of Llumnix.
- Scripts to prepare the environment, run the experiments, plot the figures, and validate the claims in this paper automatically.
- A `README` file including detailed instructions on how to use this artifact.

Hosting

The artifact is publicly available at `https://github.com/AlibabaPAI/llumnix` (the `osdi24ae` branch). Note that this is not the same branch as the official release of Llumnix (the `main` branch). We will describe their difference later.

Requirements

The artifact runs on GPU machines, with software dependencies mostly the same as those of vLLM. To reproduce our results, you would need 4 GPU machines each with 4 A10 GPUs (24 GB). We recommend that you use the same VM type as in our experiments (`ecs.gn7i-c32g1.32xlarge` on Alibaba Cloud).

Difference from the Official Release

This artifact is a research prototype and was used during the experiments of this paper. After the paper submission, we refactored it into a new implementation that is more production-ready, *i.e.*, the official release, as described in §5. Major differences between the two versions include:

- The artifact is directly based on the vLLM code base, whereas the official release is a standalone Python library, making it more extensible and non-intrusive to backend inference engines.
- The artifact is not fault-tolerant, whereas the official release provides fault tolerance for each component.
- The official release is still being actively developed, and has supported or will support a series of new features, such as scalable API servicing via distributed request frontends, support for newer versions of vLLM and more models, further improvements of the scheduling policies, *etc*.

The artifact is sufficient to reproduce the experiment results in this paper. However, if you want to use Llumnix in production or conduct further research, we do recommend the official release.

DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Yinmin Zhong[1] Shengyu Liu[1] Junda Chen[3] Jianbo Hu[1] Yibo Zhu[2] Xuanzhe Liu[1]
Xin Jin[1] Hao Zhang[3]

[1]School of Computer Science, Peking University [2]StepFun [3]UC San Diego

Abstract

DistServe improves the performance of large language models (LLMs) serving by disaggregating the prefill and decoding computation. Existing LLM serving systems colocate the two phases and batch the computation of prefill and decoding across all users and requests. We find that this strategy not only leads to strong prefill-decoding interferences but also couples the resource allocation and parallelism plans for both phases. LLM applications often emphasize individual latency for each phase: time to first token (TTFT) for the prefill phase and time per output token (TPOT) of each request for the decoding phase. In the presence of stringent latency requirements, existing systems have to prioritize one latency over the other, or over-provision compute resources to meet both.

DistServe assigns prefill and decoding computation to different GPUs, hence eliminating prefill-decoding interferences. Given the application's TTFT and TPOT requirements, DistServe co-optimizes the resource allocation and parallelism strategy *tailored* for each phase. DistServe also places the two phases according to the serving cluster's bandwidth to minimize the communication caused by disaggregation. As a result, DistServe significantly improves LLM serving performance in terms of the maximum rate that can be served within both TTFT and TPOT constraints on each GPU. Our evaluations show that on various popular LLMs, applications, and latency requirements, DistServe can serve 7.4× more requests or 12.6× tighter SLO, compared to state-of-the-art systems, while staying within latency constraints for $> 90\%$ of requests.

1 Introduction

Large language models (LLMs), such as GPT-4 [37], Bard [2], and LLaMA [51], represent a groundbreaking shift in generative AI. They start to reshape existing Internet services, ranging from search engines to personal assistants [4], and enable fundamentally new applications, like universal chatbots [1, 16] and programming assistants [15, 42]. Yet, these advances come with a significant challenge: processing an end-to-end LLM query can be substantially slower than a standard search query [41]. In order to meet the stringent latency requirements of various applications, service providers need to over-provision compute resources, particularly many GPUs, leading to a shortfall in cost efficiency. Therefore, optimizing the cost per LLM query while adhering to high *SLO attainment* (the proportion of requests that meet the SLOs) is becoming increasingly essential for all LLM services.

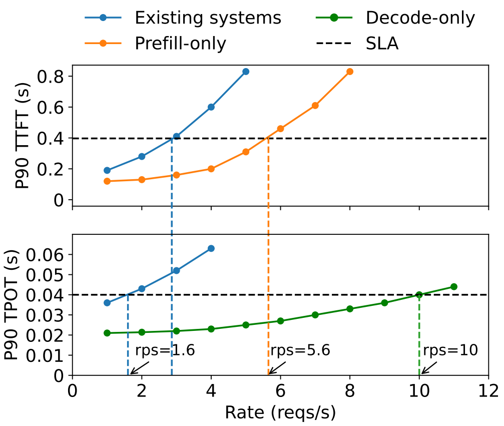


Figure 1: Performance when serving an LLM with 13B parameters under a synthetic workload with input length = 512 and output length = 64 on one NVIDIA 80GB A100. *Upper*: The P90 time-to-first-token (TTFT) latency comparing existing systems vs. a system serving only the prefill phase. *Down*: The P90 time-per-output-token (TPOT) latency comparing existing systems vs. a system serving only the decoding phase.

An LLM service responds to a user query in two phases. The *prefill phase* processes a user's prompt, composed of a sequence of tokens, to generate the first token of the response *in one step*. Following it, the *decoding phase* sequentially generates subsequent tokens *in multiple steps*; each decoding step generates a new token based on tokens generated in previous steps, until reaching a termination token. This dual-phase process distinguishes LLM services from traditional services – an LLM service's latency is uniquely measured by two key metrics: the *time to first token* (TTFT), which is the duration of the prefill phase, and the *time per output* *token* (TPOT), which represents the average time taken to generate a token for each request (except for the first token)[1]. Different applications place varying demands on each metric. For example, real-time chatbots [1] prioritize low TTFT for response promptness, while TPOT only remains important until it is faster than human reading speed (i.e., 250 words/min). Conversely, document summarization emphasizes low TPOT for faster generation of the summary.

Hence, given the application's TTFT and TPOT requirements, an effective LLM serving system should balance these needs and maximize *per-GPU goodput*, defined as the maximum request rate that can be served adhering to the SLO attainment goal (say, 90%) for each GPU provisioned – higher per-GPU goodput directly translates into lower cost per query.

As the prefill and decoding phases share the LLM weights and working memory, existing LLM serving systems typically colocate both phases on GPUs and maximize the overall system throughput – tokens generated per second across all users and requests – by batching the prefill and decoding steps across requests [31, 54]. However, to meet latency requirements, we find these systems must over-provision compute resources. To see this, Figure 1 illustrates how the P90 TTFT and TPOT shift with increasing request rates when serving a 13B LLM using existing systems [32], with workload pattern and two latency constraints set to emulate using LLM to generate a short summary for an article. Under the SLO attainment of 90%, the maximum achievable goodput on a single A100 GPU, which is constrained by the more stringent one of TTFT and TPOT requirements, is about 1.6 requests per second (rps). The performance contrasts sharply when each phase is served independently on a separate GPU, shown by the orange and green curves, which achieve per-GPU goodput of 5.6 rps for the prefill phase and 10 rps for decoding. Ideally, by allocating 2 GPUs for prefill and 1 GPU for decoding, we can effectively serve the model with an overall goodput of 10 rps, or equally 3.3 rps per GPU, which is 2.1x higher than existing systems. The gap in goodput primarily stems from the colocation of the prefill and decoding – two phases with very distinct computational characteristics and latency requirements (§2.1).

First, colocation leads to strong *prefill-decoding interference*. A prefill step often takes much longer than a decoding step. When batched together, decoding steps in the batch are delayed by the prefill steps, significantly elongating their TPOT; similarly, the inclusion of decoding steps contributes to a non-trivial increase in TTFT, as evidenced in Figure 2. Even if we schedule them separately, issues persist as they begin to compete for resources. Decoding tasks awaiting GPU execution are subject to increased queuing delays due to ongoing prefill tasks, and vice versa. Prioritized scheduling of one phase risks failing the latency requirements of the other.

Second, the prefill and decoding computation differ in latency requirements and preference for different forms of parallelism (§3). Colocating prefill and decoding, however, couples their resource allocation, and prevents implementing different parallelism strategies more suited to meeting the specific latency requirements of each phase.

To overcome these challenges, we propose to disaggregate the prefill and decoding phases of LLM inference, assigning them to separate GPUs. Our approach has two benefits. First, operating each phase independently on different GPUs eliminates prefill-decoding interference. Second, it allows to scale each phase independently with tailored resource allocation and model parallelism strategies to meet their specific latency requirements. Although disaggregation causes communication of intermediate states between GPUs, we show that the communication overhead is insubstantial (§3.3) in modern GPU clusters, and when managed appropriately, disaggregation significantly improves per-GPU goodput.

Based on the above insights, in this work, we build DistServe[2], a goodput-optimized LLM serving system by disaggregating the prefill and decoding phases. Given TTFT and TPOT requirements, DistServe first scales each phase independently by co-optimizing the GPU allocation and parallelism strategies of the prefill and decoding phase assuming serving a single model replica. The optimization ensures maximizing the per-GPU goodput and may assign different numbers of GPUs and parallelism strategies to each phase depending on their respective latency requirements. DistServe then scales this allocation to multiple instances via replication until meeting the user-required traffic rate (§4). DistServe also features an algorithm to place the prefill and decoding computation according to their allocation schemes and the cluster's bandwidth to minimize the overhead of communicating intermediate states between phases.

We implement DistServe as an orchestration layer on top of the LLM inference engine. We evaluate DistServe on various LLMs, varying the workloads based on three important real-world LLM applications: chatbots, programming assistant, and document summary. Compared to state-of-the-art solutions, DistServe can serve up to 7.4× more requests or 12.6× tighter SLO under various latency constraints. Our contributions are:

- Identify the problems of prefill-decoding interference and resource coupling in existing LLM serving systems and propose to disaggregate the two phases.
- Design a novel placement algorithm to choose the goodput-optimal schema for prefill and decoding instances automatically.
- Conduct a comprehensive evaluation of DistServe with realistic workloads.

2 Background and Motivation

An LLM service follows a client-server architecture: the client submits a sequence of text as a request to the server; the server

[1] The overall request latency equals TTFT plus TPOT times the number of generated tokens in the decoding phase.

[2] https://github.com/LLMServe/DistServe

hosts the LLM on GPUs, runs inference over the request, and responds (or streams) the generation back to the client. As explained in §1, due to the unique prefill-decoding process, LLM service may impose aggressive service-level objectives (SLOs) on both TTFT and TPOT, varying with the application's needs. The serving system must meet both SLOs while minimizing the cost associated with expensive GPUs. In other words, we want the serving system to maximize the requests served per second adhering to the SLO attainment goal for each GPU provisioned – *maximizing per-GPU goodput*. Next, we detail the LLM inference computation (§2.1) and discuss existing optimizations for LLM serving (§2.2).

2.1 LLM Inference

Modern LLMs [37, 51] predict the next token given an input sequence. This prediction involves computing a hidden representation for each token within the sequence. An LLM can take a variable number of input tokens and compute their hidden representations in parallel, and its computation workload increases superlinearly with the number of tokens processed in parallel. Regardless of the input token count, the computation demands substantial I/O to move LLM weights and intermediate states from the GPU's HBM to SRAM. This process is consistent across varying input sizes.

The prefill step deals with a new sequence, often comprising many tokens, and processes these tokens concurrently. Unlike prefill, each decoding step only processes one new token generated by the previous step. This leads to significant computational differences between the two phases. When dealing with user prompts that are not brief, the prefill step tends to be compute-bound. For instance, for a 13B LLM, computing the prefill of a 512-token sequence makes an A100 near compute-bound (see §3.1). In contrast, despite processing only one new token per step, the decoding phase incurs a similar level of I/O to the prefill phase, making it constrained by the GPU's memory bandwidth.

During both phases, intermediate states, known as KV caches [32], are generated at each token position, which are needed again in later decoding steps. To avoid recomputing them, they are saved in GPU memory. Because of the shared use of LLM weights and KV caches in memory, most LLM inference engines opt to colocate the prefill and decoding phases on GPUs, despite their distinct computational characteristics.

2.2 LLM Serving Optimization

In real-time online serving, multiple requests come and must be served within SLOs. Batching and parallelizing their computation is key for achieving low latency, high throughput, and high utilization of GPUs.

Batching. Current serving systems [9, 32, 54] utilize a batching technique known as *continuous batching*. This method batches the prefill of new requests with the decoding of ongoing ones. It boosts the GPU utilization and maximizes the overall system throughput – tokens generated per second across all users and requests. However, as mentioned in §1 and elaborated later in §2.3, this approach leads to trade-offs between TTFT and TPOT. An advanced variant of continuous batching [9] attempts to balance TTFT and TPOT by segmenting long prefill into chunks and attaching decoding jobs with a chunked prefill – but essentially, it trades TTFT for TPOT and cannot eliminate the interference (§2.3). In summary, batching prefill and decoding invariably leads to compromises in either TTFT or TPOT.

Model parallelism. In LLM serving, model parallelism is generally divided as intra- and inter-operator parallelisms [33, 46, 59]. Both can be used to support larger models but may impact serving performance differently. Intra-operator parallelism partitions computationally intensive operators, such as matrix multiplications, across multiple GPUs, accelerating computation but causing substantial communication. It reduces the execution time[3], hence latency, particularly for TTFT of the prefill phase, but requires high bandwidth connectivity between GPUs (e.g., NVLINK). Inter-operator parallelism organizes LLM layers into stages, each running on a GPU to form pipelines. It moderately increases execution time due to inter-stage communication, but linearly scales the system's rate capacity with each added GPU. In this paper, we reveal an additional benefit of model parallelism: reduced queuing delay of both prefill and decoding phases, steaming from shorter execution time. We delve into this further in §3. Besides model parallelism, replicating a model instance, irrespective of its model parallelism configurations, linearly scales the system's rate capacity.

These parallelism strategies create a complex space of optimization that requires careful trade-offs based on the application's latency requirements.

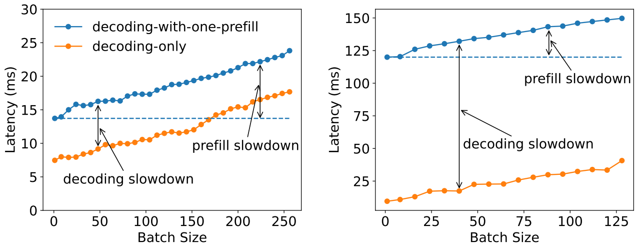
2.3 Problems and Opportunities

Colocating and batching the prefill and decoding computation to maximize the overall system throughput, as in existing systems, is cost-effective for service providers. However, in the presence of SLOs, present approaches struggle to maintain both high service quality and low cost due to the issues discussed below.

Prefill-decoding interference. As Figure 2 shows, adding a single prefill job to a batch of decoding requests significantly slows down both processes, leading to a marked increase in TTFT and TPOT. Specifically, the decoding tasks in the batch must wait for lengthier prefill jobs to complete, thus extending TPOT; the slowdown intensifies with a longer prefill, shown in Figure 2(b). Adding decoding jobs to prefill also increases the time to complete the prefill task, particularly when the GPU is already at capacity (Figure 2 blue curves).

One attempt to mitigate this interference is called *chunked-prefill with piggyback* [3, 9]. It proposes to split the long prefill

[3] we emphasize "execution time" instead of latency here because latency comprises both execution time and queuing delay.



(a) Input length = 128

(b) Input length = 1024

Figure 2: Batch execution time when serving a 13B LLM as batch size increases. Compared between a decoding-only batch and the batch adding one more prefill job.

into chunks and batch a prefill chunk with a few decoding jobs (a.k.a. piggybacking). This technique alleviates the slowdown of the decoding job caused by the long prefill job, but it does not eliminate it. Additionally, it results in an extra overhead for the prefill job which cannot be easily mitigated by adjusting the chunk size. First, if the chunk size is set much lower than the inflection point that can saturate the GPU, then the prefill job will have a longer execution time since it competes with the decoding job in the same batch and cannot solely utilize the GPU resources. Second, if we increase the chunk size to nearly saturate the GPU, the chance of piggybacking will diminish since the remaining slots for decode tokens are limited. Also, chunked-prefill causes significantly more memory access for the prefill jobs. This is because the KV cache of all previous chunks have to be loaded from HBM to SRAM repeatedly to compute each subsequent chunk. Concretely, if a prefill job is split into N equal chunks, we need to load $N+(N-1)+...+1 = O(N^2)$ chunks of KV Cache in total, compared to O(N) in the non-chunked case. This overhead will increase as the context length becomes longer.

Ineffective scheduling. Unbatching prefill and decoding jobs and scheduling them sequentially does not mitigate the interference. Decoding jobs may experience longer queuing delays due to waiting for ongoing prefill jobs on GPUs. Moreover, batches dedicated to decoding often lead to GPU underutilization. Prioritizing tasks in either phase adversely affects the latency of the other, rendering priority scheduling ineffective.

Resource and parallelism coupling. Colocating prefill and decoding phases on the same GPUs unavoidably share their resource and parallelism settings. However, each phase has its unique computational characteristic and latency requirement that calls for more heterogeneous resource allocation. For example, the prefill phase tends to be compute-bound and benefits from more intra-op parallelism to reduce execution time to meet the tight SLO on TTFT. By contrast, the optimal parallelism configuration of the decoding phase depends on the running batch size. In existing systems, due to coupling, resource allocation and parallelism plans are tailored to satisfy the *more demanding* of TTFT and TPOT, which may not be ideal for the other. This often leads to resource over-provisioning to meet both SLOs.

Opportunities. To address these issues, we propose to disaggregate the prefill and decoding phases. We use the term *instance* to denote a unit of resources that manages exactly one complete copy of model weights. One instance can correspond to many GPUs when model parallelism is applied. Note that when we disaggregate the two phases to different GPUs, each phase manages its copy of the model weights, resulting in *prefill instances* and *decoding instances*. A prefill instance, upon receiving a request, performs only the prefill computation for this request to generate the first output token. It then sends the intermediate results (mainly KV caches) to a decoding instance, which is responsible for subsequent decoding steps. Because decoding computation often has low GPU utilization, we may allocate multiple prefill instances per decoding instance. This allows batching more decoding jobs to achieve higher GPU utilization.

Disaggregating prefill and decoding naturally resolves the interference between the two phases and enables each to focus on its optimization target – TTFT or TPOT. Each type of instance can employ different resources and parallelism strategies to meet a variety of latency requirements. By adjusting the number of GPUs and parallelisms provided to the two types of instances, we can maximize the per-device goodput of the overall system, avoiding over-provisioning, eventually translating to reduced cost-per-query adhering to service quality. Next, we develop ways to find out the best resource allocation and parallelism plan for each phase.

3 Tradeoff Analysis

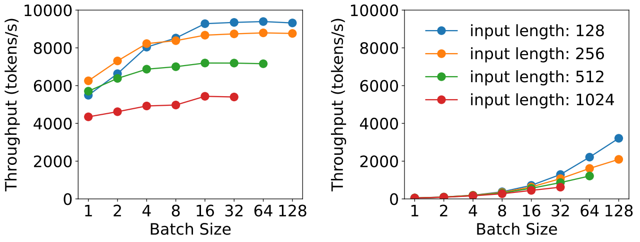
Disaggregation uncouples the two phases and allows a distinct analysis of the characteristics of each phase, providing valuable insights into the algorithm design. It also expands the design space: now each phase needs to be scaled and scheduled independently based on their latency requirements.

In this section, we analyze the computational pattern of prefill (§3.1) and decoding instances (§3.2) *post disaggregation*. We aim to identify key parameters and derive guidelines for batching and parallelism in each phase. We then highlight several practical deployment considerations (§3.3). This section lays the foundation for per-gpu goodput optimization.

3.1 Analysis for Prefill Instance

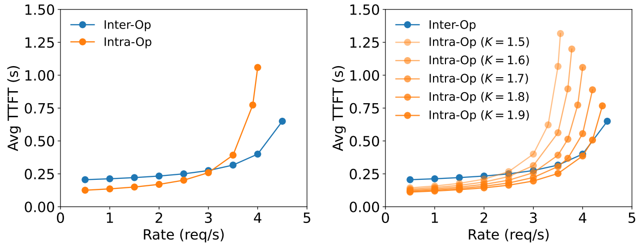
After disaggregation, the prefill phase generates the first token by processing all tokens of the user prompt in parallel. Assuming a given arrival rate, we aim to fulfill the service's latency requirement on TTFT using the least resources.

Batching strategy. The prefill step is typically compute-intensive. Figure 3(a) shows how the throughput of the prefill phase changes with the input length and the batch size. For a 13B parameter LLM, processing a single sequence of 512 tokens can fully engage an A100 GPU. Once the GPU becomes compute-bound, adding more requests to the batch no longer improves GPU efficiency. Instead, it proportionally extends the total processing time for the batch, inadvertently



(a) Prefill phase (b) Decoding phase

Figure 3: Throughput for two phases with different batch sizes and input lengths when serving an LLM with 13B parameters.



(a) Real experiment results (b) Changing intra-op speedup

Figure 4: Average TTFT when serving an LLM with 66B parameters using different parallelism on two A100 GPUs.

delaying all included requests. Hence, for prefill instances, it is necessary to profile the specific LLM and GPUs in advance to identify a critical input length threshold, denoted as L_m, beyond which the prefill phase becomes compute-bound. Batching more requests should only be considered when the input length of the scheduled request is below L_m. In practice, user prompts typically average over hundreds of tokens [8]. Batch sizes for the prefill instance are generally kept small.

Parallelism plan. To study the parallelism preferences for prefill-only instances, we serve a 66B LLM on two A100 GPUs with inter-op or intra-op parallelism strategy. To simplify the problem, we assume uniform requests input lengths of 512 tokens and a Poisson arrival process. We compare the resulting average TTFT at various arrival rates in Figure 4(a): intra-op parallelism is more efficient at lower arrival rates, while inter-op parallelism gains superiority as the rate increases. Disaggregation enables the prefill phase to function analogously to an M/D/1 queue, so we can use queuing theory to verify the observation.

We start by developing notations using the single-device case without parallelism: each request's execution time, denoted as D, remains constant due to uniform prefill length. Since one request saturates the GPU, we schedule requests via First-Come-First-Served (FCFS) without batching. Suppose the Poisson arrival rate is R and the utilization condition of $RD < 1$, the average TTFT (Avg_TTFT) can be modeled by the M/D/1 queue [47] in close form:

$$Avg\_TTFT = D + \frac{RD^2}{2(1-RD)}. \tag{1}$$

where the first term represents the execution time and the second corresponds to the queuing delay. Based on Eq. 1, we incorporate parallelism below.

With 2-way inter-op parallelism, we assume the request-level latency becomes D_s, and the slowest stage takes D_m to finish. We have $D \approx D_s \approx 2 \times D_m$, due to negligible inter-layer activation communication [33, 59]. The average TTFT with 2-way inter-op parallelism is derived as:

$$Avg\_TTFT_{inter} = D_s + \frac{RD_m^2}{2(1-RD_m)} = D + \frac{RD^2}{4(2-RD)}. \tag{2}$$

For intra-op parallelism, we introduce a speedup coefficient K, where $1 < K < 2$, reflecting the imperfect speedup caused by high communication overheads of intra-op parallelism. With the execution time $D_s = \frac{D}{K}$, the average TTFT for 2-degree intra-op parallelism is:

$$Avg\_TTFT_{intra} = \frac{D}{K} + \frac{RD^2}{2K(K-RD)}. \tag{3}$$

Comparing Eq. 2 and Eq. 3: at lower rates, where execution time (first term) is the primary factor, intra-op parallelism's reduction in execution time makes it more efficient. As the rate increases and the queuing delay (second term) becomes more significant, inter-op parallelism becomes advantageous, concurred with Figure 4(a).

The prefill phase's preference for parallelism is also influenced by TTFT SLO and the speedup coefficient K. Seen from Figure 4(a): A more stringent SLO will make intra-op parallelism more advantageous, due to its ability to reduce execution time. The value of K depends on factors such as the input length, model architecture, communication bandwidth, and placement [46, 59]. As shown in Figure 4(b), a decrease in K notably reduces the efficacy of intra-op parallelism. §4 develops algorithms that optimize the resource and parallelism configurations taking into consideration these knobs.

3.2 Analysis for Decoding Instance

Unlike the prefill instance, a decoding instance follows a distinct computational pattern: it receives the KV caches and the first output token from the prefill instance and generates subsequent tokens one at a time. For decoding instances, our optimization goal is to satisfy the application's TPOT requirement using minimal computing resources.

Batching strategy. Since a single decoding job is heavily bandwidth-bound, batching is key to avoiding low GPU utilization (hence high per-gpu goodput), as shown in Figure 3(b). In existing systems where the prefill and decoding phases are colocated, increasing the decoding batch size is difficult because it conflicts with meeting latency goals, particularly in scenarios with high request rates. This is because sharing GPUs cause competition between prefill and decoding jobs, leading to a trade-off between TTFT and TPOT. For

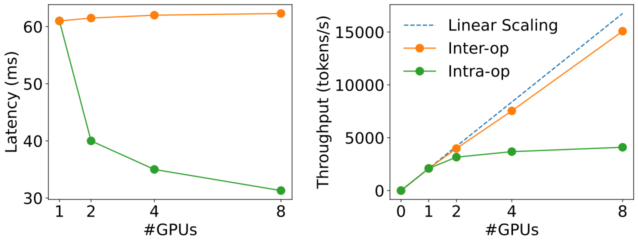


Figure 5: Decoding phase latency and throughput when serving a 13B LLM with batch size = 128 and input length = 256 under different parallel degrees.

example, a higher arrival rate generates more prefill jobs, demanding greater GPU time to meet TTFT requirements if prioritizing prefill jobs, which in turn adversely affects TPOT.

On the contrary, disaggregation offers a solution by enabling the allocation of multiple prefill instances to a single decoding instance. This approach allows for accumulating a larger batch size on dedicated GPUs for the decoding phase without sacrificing TPOT.

Parallelism plan. Post-disaggregation, the batch size for decoding may be constrained by GPU memory capacity, as it is necessary to maintain the KV caches for all active requests. Scaling the decoding instance with model parallelism or leveraging advanced memory management techniques for LLM KV caches, such as Paged-Attention [32] and GQA [10], enable further scaling of the decoding batch size to nearly compute-bound. As the decoding batch size continue to increase to approach the compute-bound, the decoding computation begins to resemble the prefill phase. With this observation, we investigate how the latency and throughput change under different parallelism degrees under large batch conditions in Figure 5: intra-op parallelism reduces latency with diminishing returns, caused by communication and reduced utilization after partitioning. Inter-op parallelism can almost linearly scale the throughput. Hence, when the TPOT SLO is stringent, intra-op parallelism is essential to reduce TPOT to meet latency goals. Beyond this, inter-op parallelism is preferable to enhance throughput linearly.

It is worth noting that when the model can fit into the memory of a single GPU, replication is a competitive option in addition to model parallelism for both prefill and decoding instances, to linearly scale the system's rate capacity. It may also reduce the queuing delay – as indicated by Eq. 1 – by substituting R with R/N assuming requests are equally dispatched to N replicas, at the cost of maintaining additional replicas of the model weights in GPU memory.

3.3 Practical Problems

We have developed foundational principles for selecting batching and parallelisms for each phase. In this section, we discuss and address several challenges encountered during the practical deployment of disaggregated prefill and decoding phases.

Variable prefill length. §3 has assumed uniform prompt length across requests. In real deployments, depending on the LLM application, the lengths of requests are non-uniform. The non-uniformity can cause pipeline bubbles [28, 36] for prefill instances applying inter-op parallelism because the execution time of pipeline stages across requests of different lengths will vary. This results in slight deviations from the conclusions indicated by using the M/D/1 queue model. To address the problem, §4 develops algorithms that search for parallelisms based on workloads, and resort to scheduling to minimize the bubbles (§4.3).

Communication overhead. Transferring KV caches from prefill to decoding instances incurs notable overheads. For example, the KV cache size of a single 512-token request on OPT-66B is approximately 1.13GB. Assuming an average arrival rate of 10 rps, we need to transfer 11.3GB data per second—or equivalently 90Gbps bandwidth to render the overhead invisible. While many modern GPU clusters for LLMs are equipped with InfiniBand (e.g., 800 Gbps), in cases where cross-node bandwidth is limited, DistServe relies on the commonly available intra-node NVLINK, where the peak bandwidth between A100 GPUs is 600 GB/s, again rendering the transmission overhead negligible (see §6.3). However, this requirement imposes additional constraints on the placement of prefill and decoding instances that we take into consideration in the next section.

Through the analysis in this section, we identify the workload pattern, placement constraints, SLO requirements, parallelism strategies, and resource allocation as key parameters that create a web of considerations in designing the disaggregated serving system. How to automatically navigate the search space to find the configuration that achieves optimal per-gpu goodput is challenging, and addressed next.

4 Method

We built DistServe to solve the above challenges. Given the model, workload characteristic, latency requirements, and SLO attainment target, DistServe will determine (a) the parallelism strategies for prefill and decoding instances, (b) the number of each instance type to deploy, as well as (c) how to place them onto the physical cluster. We call the solution a *placement*. Our goal is to find a placement that maximizes the per-gpu goodput.

As explained in §3.3, a key design consideration is to manage communications between disaggregated prefill and decoding phases, given varying cluster setups. In this section, we first present two placement algorithms: one for clusters with high-speed cross-node networks (§4.1) and the other for environments lacking such infrastructure (§4.2); the latter introduces additional constraints. We then develop online scheduling optimizations that adapt to the nuances of real-world workloads (§4.3).

Algorithm 1 High Node-Affinity Placement Algorithm

```
Input: LLM $G$, #node limit per-instance $N$, #GPU per-node $M$, GPU memory capacity $C$, workload $W$, traffic rate $R$.
Output: the placement $best\_plm$.
  $config_p, config_d \leftarrow \emptyset, \emptyset$
  for $intra\_op \in \{1, 2, ..., M\}$ do
      for $inter\_op \in \{1, 2, ..., \frac{N \times M}{intra\_op}\}$ do
          if $\frac{G.size}{inter\_op \times intra\_op} < C$ then
              $config \leftarrow (inter\_op, intra\_op)$
              $\hat{G} \leftarrow$ parallel$(G, config)$
              $config.goodput \leftarrow$ simu_prefill$(\hat{G}, W)$
              if $\frac{config_p.goodput}{config_p.num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ then
                  $config_p \leftarrow config$
              $config.goodput \leftarrow$ simu_decode$(\hat{G}, W)$
              if $\frac{config_d.goodput}{config_d.num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ then
                  $config_d \leftarrow config$
  $n, m \leftarrow \lceil \frac{R}{config_p.goodput} \rceil, \lceil \frac{R}{config_d.goodput} \rceil$
  $best\_plm \leftarrow (n, config_p, m, config_d)$
  return $best\_plm$
```

Algorithm 2 Low Node-Affinity Placement Algorithm

```
Input: LLM $G$, #node limit per-instance $N$, #GPU per-node $M$, GPU memory capacity $C$, workload $W$, traffic rate $R$.
Output: the placement $best\_plm$.
  $config^* \leftarrow \emptyset$
  for $inter\_op \in \{1, 2, ..., N\}$ do
      $\mathcal{P} \leftarrow$ get_intra_node_configs$(G, M, C, inter\_op)$
      for $P_p \in \mathcal{P}$ do
          for $P_d \in \mathcal{P}$ do
              if $P_p.num\_gpus + P_d.num\_gpus \leq M$ then
                  $config \leftarrow (inter\_op, P_p, P_d)$
                  $\hat{G}_p, \hat{G}_d \leftarrow$ parallel$(G, config)$
                  $config.goodput \leftarrow$ simulate$(\hat{G}_p, \hat{G}_d, W)$
                  if $\frac{config.^*goodput}{config.^*num\_gpus} < \frac{config.goodput}{config.num\_gpus}$ then
                      $config^* \leftarrow config$
  $n \leftarrow \lceil \frac{R}{config.^*goodput} \rceil$
  $best\_plm \leftarrow (n, config^*)$
  return $best\_plm$
```

4.1 Placement for High Node-Affinity Cluster

On high node-affinity clusters equipped with Infiniband, KV caches transmission overhead across nodes is negligible, DistServe can deploy prefill and decoding instances across any two nodes without constraints. We propose a two-level placement algorithm for such scenarios: we first optimize the parallelism configurations for prefill and decoding instances separately to attain phase-level optimal per-gpu goodput; then, we use replication to match the overall traffic rate.

However, finding the optimal parallel configuration for a single instance type, such as for the prefill instance, is still challenging, due to the lack of a simple analytical formula to calculate the SLO attainment (a.k.a., percentage of requests that meet TTFT requirement), given that the workload has diverse input, output lengths, and irregular arrival patterns. Gauging the SLO via real-testbed profiling is time-prohibitive. We thus resort to building a simulator to estimate the SLO attainment, assuming prior knowledge of the workload's arrival process and input and output length distributions. Although short-term interval is impossible to predict, the workload pattern over longer timescales (e.g., hours or days) is often predictable [33, 55]. DistServe fits a distribution from the history request traces and resamples new traces from the distribution as the input workload to the simulator to compute the SLO attainment. Next, DistServe simply enumerates the placements and finds the maximum rate that meets the SLO attainment target with binary search and simulation trials.

Algorithm 1 outlines the process. We enumerate all feasible parallel configurations, subject to cluster capacity limit, for both prefill and decoding instances. Then, for a specific prefill phase configuration, we use `simu_prefill` to simulate and find its maximum goodput via binary search (similarly for using `simu_decode` for decoding). After determining the optimal parallel configurations for both prefill and decoding instances, we replicate them to achieve the user-required overall traffic rate according to their goodput.

The complexity of Algorithm 1 is $O(NM^2)$, with N as the node limit per instance and M representing the typical number of GPUs per node in modern clusters (e.g., 8). The search space is manageable and the solving time is under 1.3 minutes in our largest setting, as demonstrated in §6.5.

Simulator building. Algorithm 1 relies on a simulator to estimate the goodput under various SLOs and SLO attainment goals given the workload and the parallelism plan. To build an accurate simulator, we analyze the FLOPs and the number of memory accesses for prefill and decoding phases respectively, and use a latency model to approximate the inference execution time. See details in Appendix A. The simulator aligns well with real profiling results, thanks to the high predictability of DNN workloads [23, 33], verified in §6.4.

By far, we have developed Algorithm 1 assuming we can place the prefill and decoding instance between any two nodes (or on the same node) of the cluster, and the KV cache transmission utilizes high bandwidth network. In many real clusters, GPUs inside a node access to high-bandwidth NVLINK while GPUs distributed across nodes have limited bandwidth. We next develop an algorithm to address this constraint.

4.2 Placement for Low Node-Affinity Cluster

A straightforward solution is to always colocate prefill and decoding instances on the same node, utilizing the NVLINK, which is commonly available inside a GPU node. For large models, e.g. with 175B parameters (350GB), we may be unable to even host a single pair of prefill and decoding instances in an 8-GPU node ($80G \times 8 = 640G < 350 \times 2GB$). We incorporate this as additional placement constraints and co-optimize it with model parallelism, presented in Algorithm 2.

The key insight is that KV cache transfer occurs exclusively between corresponding layers of prefill and decoding instances. Leveraging inter-op parallelism, we group layers into stages and divide each instance into segments, termed as *instance segments*, with each segment maintaining one specific inter-op stage. By colocating prefill and decoding segments of the same stage within a single node, we force the transfer of intermediate states to occur only via NVLINK. Inside a node, we set the same parallelism and resource allocation for segments of the same instance. Given the typical limitation of GPUs per node (usually 8), we can enumerate possible configurations inside one node and use the simulator to identify the configurations that yield the best goodput.

As outlined in Algorithm 2, we begin by enumerating inter-op parallelism degrees to get all the possible instance segments. For each segment, we get all possible intra-node parallelism configurations by calling `get_intra_node_configs`. Then we use simulation to find the optimal one and replicate it to satisfy the target traffic rate.

4.3 Online scheduling

The runtime architecture of DistServe is shown in Figure 6. DistServe operates with a simple FCFS scheduling policy. All incoming requests arrive at a centralized controller, then dispatched to the prefill instance with the shortest queue for prefill processing, followed by dispatch to the least loaded decoding instance for decoding steps. This setup, while simple, is optimized with several key enhancements tailored to the nuances of real-world workloads.

Reducing pipeline bubbles. To mitigate the pipeline bubbles caused by non-uniform prompt lengths (§3.3), we schedule the requests in a way that balances the execution time across all batches in the pipeline. This is achieved by noting that, for both prefill and decoding instances, the number of new tokens in the batch is a reliable indicator of the batch's real execution time. For prefill instances, we profile the target model and GPU to figure out the shortest prompt length L_m needed to saturate the GPU. We schedule prefill batches with a total sequence length close to L_m, by either batching multiple requests shorter than L_m or individually scheduling requests longer than L_m. For decoding instances, we set L_m as the largest batch size.

Combat busrtiness. Burstiness in workloads can cause a deluge of KV caches to transfer from prefill to decoding instances, risking memory overload on decoding instances. To circumvent this, DistServe employs a "pull" method for KV cache transmission rather than a "push" approach – decoding instances fetch KV cache from prefill instances *as needed*, using the GPU memory of prefill instances as a queuing buffer. This way, the prefill instance can continue handling other prefill jobs by simply retaining the KV Cache in the GPU memory after processing the prompt. Hence, each type of instance operates at its own pace without complex coordination.

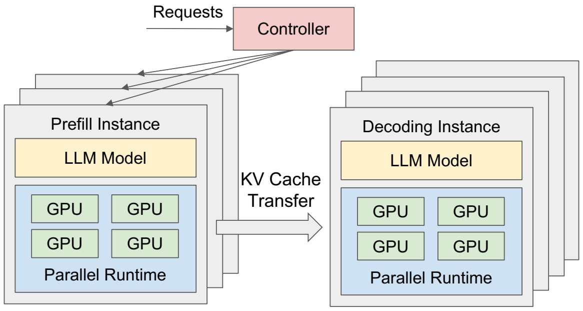


Figure 6: DistServe Runtime System Architecture

Replaning. The resource and parallelism plan in DistServe is optimized for a specific workload pattern, which may become suboptimal if the workload pattern changes over time. DistServe implement periodic replanning. A workload profiler monitors key parameters such as the average input and output length of the requests, the average arrival rate, etc. If a significant pattern shift is detected, DistServe will trigger a rerun of the placement algorithm based on recent historical data. This process is expedient – the proposed algorithm runs in seconds (§6.5) and reloading LLM weights can be completed within minutes – far shorter than the hourly scale at which real-world workload variations tend to occur.

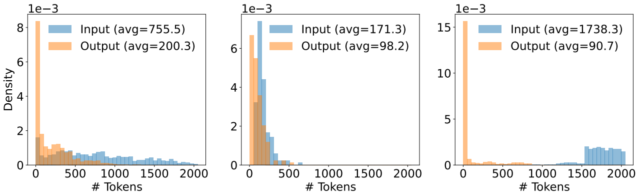
Preemption and fault tolerance. DistServe does not implement advanced runtime policies like preemption [26] and fault tolerance [58], which are complementary to disaggregation. Nevertheless, we discuss how they fit into DistServe. In DistServe, the FCFS policy can lead to a "convoy effect", where longer requests block shorter ones in the prefill stage. Incorporating preemptive strategies, as suggested in existing literature [53], could enhance efficiency and is feasible within our system's architecture. While not a primary focus in the current DistServe, fault tolerance is a critical aspect for consideration. In traditional colocation- and replication-based systems, a fault in one instance typically does not disrupt other replica instances. However, in DistServe, the dependency between prefill and decoding instances introduces the risk of fault propagation. For example, a fault in a single decoding instance mapped to multiple prefill instances could potentially cripple the entire service and cluster. We leave both as future work.

5 Implementation

DistServe is an end-to-end distributed serving system for LLMs with a placement algorithm module, a RESTful API frontend, an orchestration layer, and a parallel execution engine. The algorithm module, frontend, and orchestration layer are implemented with 6.5K lines of Python code. The parallel execution engine is implemented with 8.1K lines of

Application	Model Size	TTFT	TPOT	Dataset
Chatbot OPT-13B	26GB	0.25s	0.1s	ShareGPT [8]
Chatbot OPT-66B	132GB	2.5s	0.15s	ShareGPT [8]
Chatbot OPT-175B	350GB	4.0s	0.2s	ShareGPT [8]
Code Completion OPT-66B	132GB	0.125s	0.2s	HumanEval [14]
Summarization OPT-66B	132GB	15s	0.15s	LongBench [13]

Table 1: Workloads in evaluation and latency requirements.



(a) ShareGPT (b) HumanEval (c) LongBench

Figure 7: The input and output length distributions of (a) ShareGPT, (b) HumanEval, and (c) LongBench datasets.

C++/CUDA code.

The placement algorithm module implements the algorithm and the simulator mentioned in §4 which gives the placement decision for a specific model and cluster setting. The frontend supports an OpenAI API-compatible interface where clients can specify the sampling parameters like maximum output length and temperature. The orchestration layer manages the prefill and decoding instances, responsible for request dispatching, KV cache transmission, and results delivery. It utilizes NCCL [6] for cross-node GPU communication and asynchronous CudaMemcpy for intra-node communication, which avoids blocking the GPU computation during transmission. Each instance is powered by a parallel execution engine, which uses Ray [35] actor to implement GPU workers that execute the LLM inference and manage the KV Cache in a distributed manner. It integrates many recent LLM optimizations like continuous batching [54], FlashAttention [20], PagedAttention [32] and supports popular open-source LLMs such as OPT [56] and LLaMA [51].

6 Evaluation

In this section, we evaluate DistServe under different sizes of LLMs ranging from 13B to 175B and various application datasets including chatbot, code-completion, and summarization. The evaluation shows that DistServe consistently outperforms the current state-of-the-art system across all the settings (§6.2). Specifically, DistServe can handle up to 7.4× higher rates and 12.6× more stringent SLO while meeting the latency requirements for over 90% requests. Additionally, we analyze the latency breakdown in DistServe to show the communication overhead is insubstantial thanks to our bandwidth-aware placement algorithm (§6.3) and do ablation studies of our techniques (§6.4). Finally, we profile the execution time of our placement algorithm (§6.5).

6.1 Experiments Setup

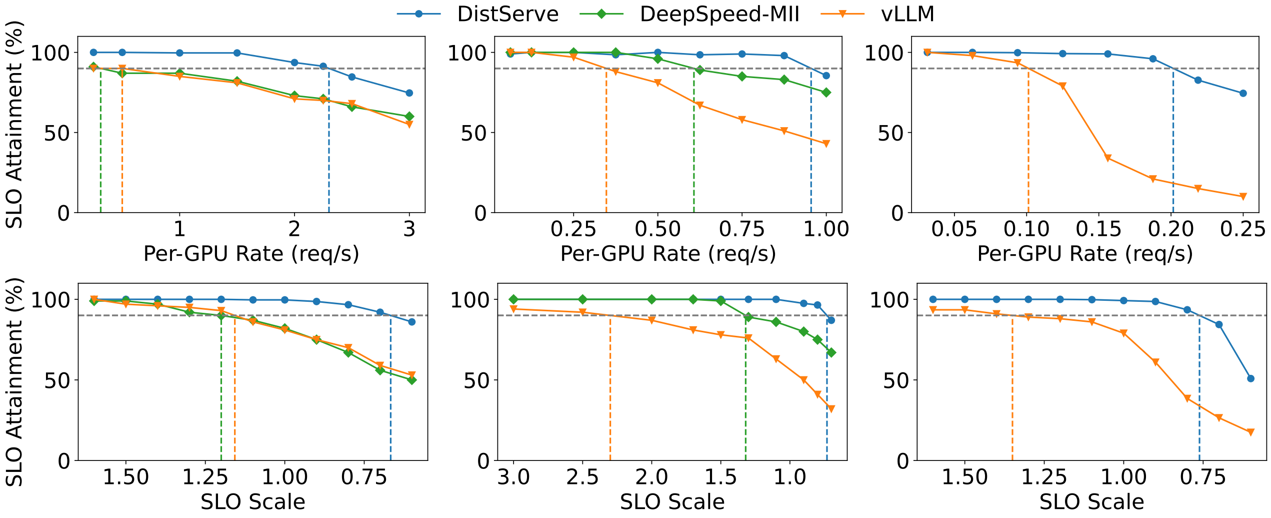
Cluster testbed. We deploy DistServe on a cluster with 4 nodes and 32 GPUs. Each node has 8 NVIDIA SXM A100-80GB GPUs connected with NVLINK. The cross-node bandwidth is 25Gbps. Due to the limited cross-node bandwidth, we use the low node-affinity placement algorithm (§2) for DistServe in most of the experiments except for the ablation study (§6.4) which uses simulation.

Model and workloads setup. Similar to prior work on LLM serving [32], we choose the OPT [56] model series, which is a representative LLM family widely used in academia and industry. Newer GPT model families are adopting memory-efficient attention mechanisms like GQA [10] and MQA [44]. DistServe will show better performance on these models because the transmission overhead is lower due to the decrease in KV cache size. We choose OPT which uses the classic MHA [52] to put enough pressure on the transmission overhead. We use FP16 precision in all experiments. For workloads, as shown in Table 1, We choose three typical LLM applications and set the SLOs empirically based on their service target because there exists no available SLO settings for these applications as far as we know. For each application, we select a suitable dataset and sample requests from it for evaluation. Since all the datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates. Due to the space limit, we test the chatbot workload on all three OPT models and the other two workloads on OPT-66B, which matches the largest size in the recent open-source LLM series [51].

- **Chatbot** [1]: We use the ShareGPT dataset [8] for the chatbot application, which is a collection of user-shared conversations with ChatGPT. For OPT-13B, the TTFT SLO is set to 0.25s for responsiveness and the TPOT SLO is set to 0.1s which is higher than the normal human read speed. For OPT-66B and OPT-175B, we slightly relax the two SLOs due to the increase in model execution latency.
- **Code completion** [14]: We use the HumanEval [14] dataset for the code completion task. It includes 164 programming problems with a function signature or docstring which is used to evaluate the performance of code completion models. Since the code completion model is used as a personal real-time coding assistant, we set both SLOs to be stringent.
- **Summarization** [5]: It is a popular LLM task to generate a concise summary for a long article, essay, or even an academic paper. We use LongBench [13] dataset which contains the summarization task[4]. As shown in Figure 7, LongBench has much longer input lengths than the other two datasets. So we set a loose TTFT SLO but require a stringent TPOT.

Metrics. We use *SLO attainment* as the major evaluation metric. Under a specific SLO attainment goal (say, 90%), we are

[4] We capped the input lengths in LongBench because OPT's absolute positional embedding only supports a maximum length of 2048.



(a) OPT-13B (b) OPT-66B (C) OPT-175B

Figure 8: Chatbot application with OPT models on the ShareGPT dataset.

concerned with two things: the maximum per-GPU goodput and the minimal SLO the system can handle. We are particularly interested in an SLO attainment of 90% (indicated by the vertical lines in all curve plots), but will also vary the rate and latency requirements to observe how the SLO attainment changes. We also include the results in the Appendix for an SLO attainment of 99% to show the system performance under a more stringent SLO attainment target.

Baselines. We compare DistServe to two baseline systems:

- **vLLM [32]:** vLLM is a representative LLM serving system widely used in both academia and industry. It supports *continuous batching* [54] to increase throughput and *paged-attention* [32] to reduce memory fragmentation during KV cache allocation. However, it colocates the prefill and decoding computation to maximize the overall system throughput and struggles to meet the latency requirements cost-efficiently. Since vLLM only supports intra-op parallelism, we follow previous work [32] to set intra-op equals 1, 4, and 8 for the three OPT models, respectively.
- **DeepSpeed-MII [3]:** DeepSpeed Model Implementations for Inference (MII) supports *chunked-prefill* by decomposing long prompts into smaller chunks and composing with short prompts to exactly fill a target token budget. It mitigates but cannot eliminate the prefill-decoding interference caused by the long prefill job. We set its intra-op the same as vLLM for OPT-13B and OPT-66B for a fair comparison. However, DeepSpeed-MII cannot serve OPT-175B whose $vocab\_size = 50272$ because its underlying kernel implementation requires $vocab\_size/intra\_op$ is a multiple of 8 where intra-op equals 8 does not satisfy. Setting intra-op equals 4 can satisfy this requirement but will cause the out-of-memory issue.

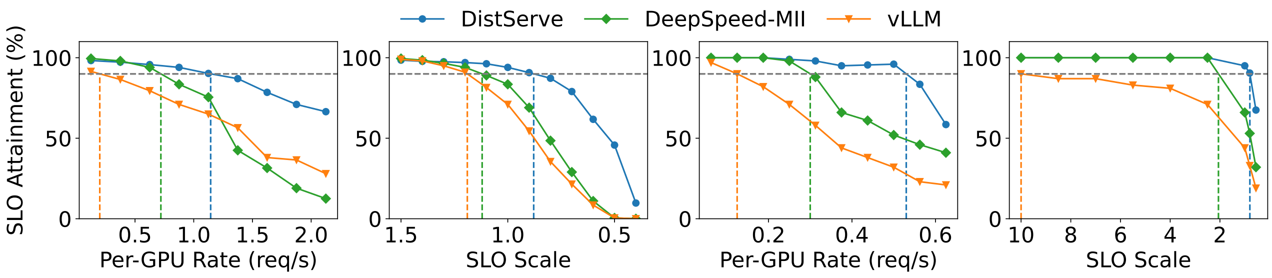
6.2 End-to-end Experiments

In this Section, we compare the end-to-end performance of DistServe against the baselines on real application datasets.

Chatbot. We evaluate the performance of DistServe on the chatbot application for all three OPT models. The first row of Figure 8 illustrates that when we gradually increase the rate, more requests will violate the latency requirements and the SLO attainment decreases. The vertical line shows the maximum per-GPU rate the system can handle to meet latency requirements for over 90% of the requests.

On the ShareGPT dataset, DistServe can sustain 2.0×–4.6× higher request rate compared to vLLM. This is because DistLLM eliminates the prefill-decoding interference through disaggregation. Two phases can optimize their own objectives by allocating different resources and employing tailored parallelism strategies. Specifically, by analyzing the chosen placement strategy[5] for 175B, we find the prefill instance has inter-op = 3, intra-op = 3; and the decoding instance has inter-op = 3, intra-op = 4. Under this placement, DistServe can effectively balance the load between the two instances on ShareGPT, meeting latency requirements at the lowest cost. This non-trivial placement strategy is challenging to manually find, proving the effectiveness of the algorithm. In the case of vLLM, collocating prefill and decoding greatly slows down the decoding phase, thereby significantly increasing TPOT. Due to the stringent TPOT requirements of chatbot applications, although vLLM meets the TTFT SLO for most requests, the overall SLO attainment is dragged down by a large number of requests that violate the TPOT SLO. Compared to DeepSpeed-MII, DistServe can sustain 1.6×–7.4× higher request rate. DeepSpeed-MII shows better performance

[5] All the placements chosen by DistServe can be found in Appendix B.



(a) Code Completion

(b) Summarization

Figure 9: Code completion and summarization tasks with OPT-66B on HumanEval and LongBench datasets, respectively.

on larger models because the prefill job is larger and chunked-prefill mitigates the interference to some extent. However, due to the reasons discussed in §2.3, chunked prefill is slower than full prefill, so it struggles to meet the TTFT SLO as a sacrifice for better TPOT.

The second row of Figure 8 indicates the robustness to the changing latency requirements of the two systems. We fix the rate and then linearly scale the two latency requirements in Table 1 simultaneously using a parameter called *SLO Scale*. As SLO Scale decreases, the latency requirement is more stringent. We aim to observe the most stringent SLO Scale that the system can withstand while still achieving the attainment target. Figure 8 shows that DistServe can achieve 1.8×–3.2× more stringent SLO than vLLM and 1.7×–1.8× more stringent SLO than DeepSpeed-MII, thus providing more engaging service quality to the users.

Code completion. Figure 9(a) shows the performance of DistServe on the code completion task when serving OPT-66B. DistServe can sustain 5.7× higher request rate and 1.4× more stringent SLO than vLLM. Compared to DeepSpeed-MII, DistServe can sustain 1.6× higher request rate and 1.4× more stringent SLO. As a real-time coding assistant, the code completion task demands lower TTFT than chatbot, this leads to both systems ultimately being constrained by the TTFT requirement. However, in comparison, by eliminating the interference of the decoding jobs and automatically increasing intra-operation parallelism in prefill instances through the searching algorithm, DistServe reduces the average latency of the prefill jobs, thereby meeting the TTFT requirements of more requests.

Summarization. Figure 9(b) shows the performance of DistServe on the summarization task when serving OPT-66B. DistServe achieves 4.3× higher request rate and 12.6× more stringent SLO than vLLM. Compared to DeepSpeed-MII, DistServe achieves 1.8× higher request rate and 2.6× more stringent SLO. The requests sampled from LongBench dataset have long input lengths, which brings significant pressure to the prefill computation. However, due to the loose requirement of TTFT for the summarization task, the TPOT service quality becomes particularly important. Since vLLM collocates prefill and decoding phases, it experiences a greater slowdown in the decoding phase with long prefill jobs and fails to meet the TPOT requirement.

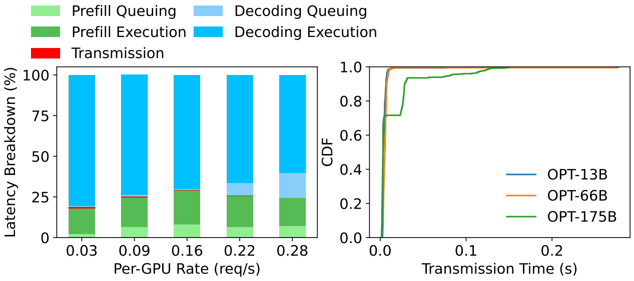


Figure 10: *Left:* Latency breakdown when serving OPT-175B on ShareGPT dataset with DistServe. *Right:* The CDF function of KV Cache transmission time for three OPT models.

The results above are all under the 90% SLO attainment target. We observe that DistServe can have better performance under a more stringent attainment target (say, 99%) and present the results in Appendix C.

6.3 Latency Breakdown

To understand DistServe's performance in detail, we make a latency breakdown of the requests in DistServe. We divide the processing lifecycle of a request in DistServe into five stages: *prefill queuing*, *prefill execution*, *transmission*, *decoding queuing*, and *decoding execution*. The total time consumed by all requests in each stage is then summed up to determine their respective proportions in the system's total execution time.

Figure 10(a) shows the latency breakdown for the OPT-175B models on the ShareGPT dataset. We chose OPT-175B because the KV Cache transmission is more demanding for larger models. In fact, even for OPT-175B, the KV Cache transmission only accounts for less than 0.1% of the total latency. Even by examining the CDF of the absolute transmission time shown in Figure 10(b), we observe that over 95% of requests experience a delay of less than 30ms, despite our testbed having only limited cross-node bandwidth. This is due to the algorithm described in §4.2, where we require the prefill and decoding instance to maintain the same stage on one machine, enabling the use of intra-node NVLINK bandwidth for transmission, thus significantly reducing transmission delay.

Rate	vLLM		DistServe-Low	
(req/s)	Real System	Simulator	Real System	Simulator
1.0	97.0%	96.8%	100.0%	100.0%
1.5	65.5%	65.1%	100.0%	100.0%
2.0	52.8%	51.0%	99.3%	99.3%
2.5	44.9%	46.1%	87.3%	88.3%
3.0	36.7%	38.3%	83.0%	84.1%
3.5	27.8%	28.0%	77.3%	77.0%
4.0	23.6%	24.1%	70.0%	68.9%

Table 2: Comparison of the SLO attainment reported by the simulator and the real system under different rates.

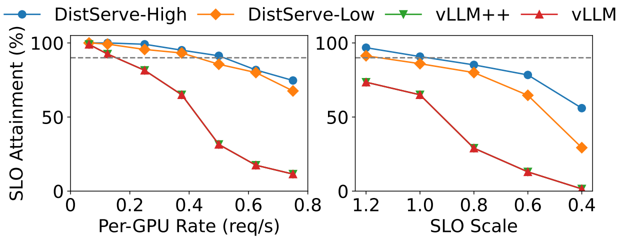


Figure 11: Ablation experiments.

6.4 Ablation Studies

We study the effectiveness of the two key innovations in DistServe: disaggregation and the placement searching algorithm. In §6.2, we choose the default parallelism setting for vLLM following its original paper [32]. So we implement "vLLM++" which enumerates different parallelism strategies and chooses the best. For DistServe, We also compare the placement found by Alg. 2 (DistServe-Low) with the one found by Alg. 1 (DistServe-High) which has fewer searching constraints and assumes high cross-node bandwidth. Since vLLM does not support inter-op parallelism and our physical testbed does not have high cross-node bandwidth, we use simulation for this experiment.

Simulator accuracy. Noticing that DNN model execution [24] has high predictability, even under parallel settings [33, 59]. We study the accuracy of the simulator in Tab. 2. For "vLLM" and "DistServe-Low", we compare the SLO attainment reported by the simulator and by real runs on our testbed under different rates. The error is less than 2% in all cases, verifying the accuracy of our simulator.

Results. Figure 11 shows the performance of the four systems when serving OPT-66B on the ShareGPT dataset. "vLLM++" has the same performance as "vLLM" because we find the default parallelism setting (intra-op=4) has the best per-GPU goodput. This further demonstrates the importance of disaggregation. The interference between the prefill and decoding phases significantly reduces the potential performance improvement through adjusting parallelism. In contrast, "DistLLM-High" can achieve further improvements over "DistLLM-Low" because it is not constrained by the deployment constraint that the prefill and decoding instance on one node should share the same model stage. Through disaggregation, we can use tailored parallelism strategies for prefill and decoding instances and optimize their targets without the coupling effects.

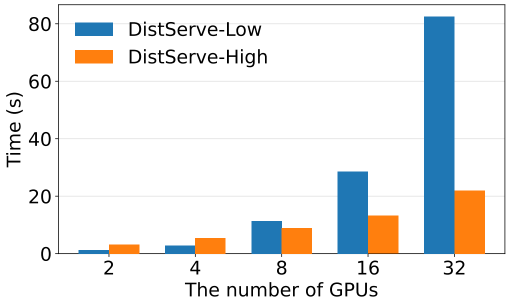


Figure 12: Algorithm Running Time

6.5 Algorithm Running Time

Figure 12 shows the running time for Alg. 1 (DistServe-Low) and Alg. 2 (DistServe-High) on an AWS m5d.metal instance with 96 cores as the number of GPUs ($N \times M$) provided to a single instance increases. According to the results, DistServe scales well with the number of GPUs and is independent of the model size. This is because the simulator only simulates discrete events and the running time is the same no matter how big the model is. On the other hand, both algorithms are highly parallelizable, as the searches for different parallelism strategies are independent of each other, allowing the execution time of the algorithms to accelerate almost linearly with more CPU cores.

As the number of GPUs increases, the execution time of "Dist-Low" becomes higher than that of "Dist-High". This is because the search for parallelism strategies for prefill and decoding instances in "Dist-High" is independent and can be parallelized. But for "Dist-Low", due to additional restrictions on deployment, we need to enumerate all the possible intra-node parallelism combinations for prefill and decoding instances. Even so, the execution time of the algorithm is in minutes, and since it only needs to be executed once before each redeployment, this overhead is acceptable.

7 Discussion

In this paper, we focus on the goodput-optimized setting and propose DistServe under the large-scale LLM serving scenario. As LLMs are widely used and deployed across various service scenarios with different optimization targets and resource limits, it becomes almost impossible to find a one-size-fits-all solution that effectively addresses all aspects of LLM serving. In this section, we discuss the pros and cons of DistServe and potentially better solutions in other scenarios.

Throughput-optimized scenarios. In offline applications that are not latency-sensitive, users typically have lower requirements for response time [45]. This allows serving systems to shift focus towards maximizing overall throughput instead of goodput and the effectiveness of DistServe may be compromised. In this case, techniques such as chunked-prefill with piggyback [3, 9] may be preferred since it can fill each batch to the compute-bound threshold, thereby maintaining higher GPU utilization in every iteration.

Resource-constrained scenarios. Small-scale enterprises and individual researchers often lack the resources to deploy LLMs on large-scale clusters [45, 48]. In resource-constrained scenarios, such as environments with only a few or even a single GPU, the design space for DistServe is significantly limited. It struggles or even fails to adjust the parallel strategies and resource allocation to effectively enhance serving performance. In this case, simpler architectural choices like non-disaggregated systems [3, 32] may reduce deployment complexity and optimize operational efficiency.

Long-context scenarios. Nowadays, more and more GPT models support extremely long contexts, such as Claude-3 [11], Gemini-1.5 [22], and Large World Model (LWM) [34], which all have a 1M context window. In such scenarios, the transmission overhead will increase as the size of the KV cache grows linearly with the prompt length. However, the prefill computation grows quadratically, so the relative duration of transmission and prefill job decreases. Meanwhile, a longer context further exacerbates the disparity in computational demands between prefill and decoding jobs, leading to increased interference between them. Therefore, the disaggregation approach proposed in DistServe remains promising in long-context serving.

8 Related Work

Inference serving. There has been plenty of work on inference serving recently. They range from general-purpose production-grade systems like TorchServe [7] and NVIDIA Triton [19] to systems optimized specifically for Transformer-based LLMs [9, 18, 21, 33, 50, 53, 54, 60]. Among them, Orca [54] introduces continuous batching to increase throughput. vLLM [32] proposes paged-attention for fine-grained KV cache management. SARATHI [9] suggests a chunked-prefill approach, splitting a prefill request into chunks and piggybacking decoding requests to improve hardware utilization. FastServe [53] implements iteration-level preemptive scheduling to mitigate the queuing delay caused by long jobs. However, they all employ a colocation approach for prefill and decoding processing, thus leading to severe interference. There are also concurrent works such as Splitwise [38], TetriInfer [27] and DéjàVu [49] which adopt similar disaggregation idea to optimize LLM inference, further confirming the effectiveness of this method. Differently, DistServe emphasizes the goodput optimization scenario more and takes a closer look at the aspect of network bandwidth.

Goodput-optimized systems. Optimizing goodput is a hot topic in DL applications. Pollux [39] improves scheduling performance in DL clusters by dynamically adjusting resources for jobs to increase cluster-wide goodput. Sia [29] introduces a heterogeneous-aware scheduling approach that can efficiently match cluster resources to elastic resource-adaptive jobs. Clockwork [23] and Shepherd [55] provide latency-aware scheduling and preemption to improve the serving goodput, but they only target traditional small models. AlpaServe [33] focuses on LLMs, employing model parallelism to statistically multiplex the GPU execution thus improving the resource utilization. However, it only targets the non-autoregressive generation. DistServe is the first work to optimize the goodput for autoregressive LLM inference.

Resource disaggregation. Resource disaggregated systems [17, 25, 43] decouple the hardware resources from the traditional monolithic server infrastructure and separate them into resource pools to manage independently. It allows for more flexible, efficient, and scalable deployment and increases resource utilization. Many applications benefit from a truly disaggregated data center with high-speed network bandwidth and heterogenous hardware support [12, 30, 57]. DistServe shares the concept by disaggregating its system components, allowing for independent resource scaling and management.

Model parallelism for training. DistServe is orthogonal to the large body of work on model parallelism in training [28, 36, 40, 46, 59]. As described in §3.3, inference-serving workloads have unique characteristics not found in training settings. Where these systems do intersect with DistServe, is in their methods for implementing model parallelism along various dimensions. DistServe can integrate new parallelism optimizations into its placement searching algorithm.

9 Conclusion

We present DistServe, a new LLM serving architecture that disaggregates the prefill and decoding computation. DistServe maximizes the per-gpu goodput – the maximum request rate that can be served adhering to the SLO attainment goal for each GPU provisioned, hence resulting in up to 7.4× lower cost per LLM query with guaranteed satisfaction of SLOs. Our findings affirm that as latency becomes an increasingly important metric for LLM services, prefill and decoding disaggregation is a vital strategy in promising improved performance and service quality guarantees.

Acknowledgments. We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback. This work was supported by the National Natural Science Foundation of China under the grant numbers 62172008, 62325201, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Junda Chen is supported by UCSD fellowship and Hao Zhang is supported by UCSD faculty startup fund. Xin Jin is the corresponding author. Yinmin Zhong, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] Introducing chatgpt. https://openai.com/blog/chatgpt, 2022.

[2] Bard, an experiment by google. https://bard.google.com/, 2023.

[3] Deepspeed model implementations for inference (mii), 2023.

[4] Inflection tech memo. https://inflection.ai/assets/Inflection-1.pdf, 2023.

[5] Lanchain usecase: Summarization, 2023.

[6] Nvidia collective communications library (nccl), 2023.

[7] Serve, optimize and scale pytorch models in production, 2023.

[8] Sharegpt teams. https://sharegpt.com/, 2023.

[9] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[10] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[11] Anthropic. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family, 2024.

[12] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A. Thekkath. A case for disaggregation of ml data processing, 2022.

[13] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2023.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[16] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, 2023.

[17] Compute Express Link Consortium. Compute express link, 2023. Accessed: 2023-12-07.

[18] NVIDIA Corporation. Fastertransformer, 2019.

[19] NVIDIA Corporation. Triton inference server: An optimized cloud and edge inferencing solution., 2019.

[20] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[21] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *ACM PPoPP*, 2021.

[22] Google. Our next-generation model: Gemini 1.5. https://deepmind.google/technologies/gemini/, 2024.

[23] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[24] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *USENIX OSDI*, 2020.

[25] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[27] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[29] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657, 2023.

[30] Xin Jin, Zhihao Bai, Zhen Zhang, Yibo Zhu, Yinmin Zhong, and Xuanzhe Liu. Distmind: Efficient resource disaggregation for deep learning workloads. *IEEE/ACM Transactions on Networking*, pages 1–16, 2024.

[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[33] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv*, 2023.

[34] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. World model on million-length video and language with ringattention. *arXiv*, 2024.

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *USENIX OSDI*, 2018.

[36] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM SOSP*, 2019.

[37] OpenAI. Gpt-4 technical report, 2023.

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[39] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[40] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[41] Reuters, 2023.

[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[44] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[45] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[47] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[48] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu, 2023.

[49] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024.

[50] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. Hotgpt: How to make software documentation more useful with a large language model? In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 87–93, 2023.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Neural Information Processing Systems*, 2017.

[53] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *USENIX OSDI*, 2022.

[55] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. 2023.

[56] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[57] Yiying Zhang. Make it real: An end-to-end implementation of a physically disaggregated data center. *SIGOPS Oper. Syst. Rev.*, 57(1):1–9, jun 2023.

[58] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2021.

[59] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *USENIX OSDI*, 2022.

[60] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *USENIX ATC*, 2022.

A Latency Model for LLM Inference

To accurately simulate the goodput of different placement strategies, we use an analytical model to predict the execution time of the prefill and decoding phases in LLM inference.

In modern LLM serving systems [18, 32, 53], memory-bound operations like Softmax and LayerNorm are usually fused with matrix multiplication kernels for efficiency. Thus the GEMMs dominate the overall latency and our analysis primarily focuses on them.

A.1 Symbol Definition

Here are symbols related to the architecture of the model:

- h: hidden size
- n: number of heads
- s: head size ($h = n \cdot s$)
- m: FFN intermediate size

Note: If tensor parallelism is used, h, n, and m should be divided by the tensor parallelism size.

Below are symbols that characterize the batch to be executed:

- B: batch size
- $l_0, l_1, \dots, l_{B-1}$: input length of each request within the batch
- t: number of tokens in the batch, ($t = \sum_{i=0}^{B-1} l_i$)
- t_2: squared sum of the input lengths ($t_2 = \sum_{i=0}^{B-1} l_i^2$)
- b: block size in the attention kernel. This parameter is used in FlashAttention [20], a common kernel optimization technique adopted by current LLM serving systems.

A.2 Prefill Phase Latency Modeling

Since the attention operation uses specially optimized kernels, we first discuss the other four matrix multiplications in the prefill phase:

GEMM Name	Shape of M	Shape of N
QKV Linear	(t,h)	$(h,3h)$
Attn Output	(t,h)	(h,h)
FFN Input	(t,h)	(h,m)
FFN Output	(t,m)	(m,h)

The arithmetic intensity (AI) of these operations is $O(t)$. On NVIDIA A100-80GB GPU, it is compute-bound when AI is over 156. Since t usually can reach several hundred in real cases, all of these operations are compute-bound. Therefore, we can model the latency of these operations according to the total FLOPs:

$$T_1 = C_1 \cdot (4th^2 + 2thm)$$

Next, we discuss the prefill attention operation with FlashAttention [20] optimization. Since the attention only operates among the tokens in the same request, current implementations launch attention kernels for each request in the same batch. For one attention head and a request with l tokens, the attention kernel needs to perform a total of $2sl + 3sl \cdot (l/b) \approx 3sl \cdot (l/b)$ memory reads and writes, alongside $2sl^2 + sl(l/b) \approx 2sl^2$ FLOPs. So the AI is $2b/3 = 10.677$ (when $b = 16$) or 21.333 (when $b = 32$), indicating that it is a memory-bound operation on A100 GPU. Therefore, the whole attention layer latency (including all requests and all heads) can be modeled as:

$$T_2 = C_2 \cdot n \cdot \sum_{i=0}^{B-1} \frac{3sl_i^2}{b} = C_2 \cdot \frac{3nst_2}{b} = C_2 \cdot \frac{3ht_2}{b}$$

Overall, the latency of the prefill phase can be modeled as:

$$T_{Prefill} = C_1 \cdot (4th^2 + 2thm) + C_2 \cdot \frac{3ht_2}{b} + C_3$$

We use C_3 to quantify other overheads like Python Runtime, system noise, and so on. Then we use profiling and interpolation to figure out the values of C_1, C_2, and C_3.

A.3 Decoding Phase Latency Modeling

Similarly, we first focus on the following GEMMs in the decoding phase:

GEMM Name	Shape of M	Shape of N
QKV Linear	(B,h)	$(h,3h)$
Attn Output	(B,h)	(h,h)
FFN Input	(B,h)	(h,m)
FFN Output	(B,m)	(m,h)

The AI of these operations is $O(B)$. B is limited by the GPU memory size and stringent latency requirements, so in existing serving scenarios, these operations are memory-bound. The total memory reads and writes is $8Bh + 4h^2 + 2hm + 2Bm$, and since h and m are usually significantly larger than B, we can model the latency as:

$$T_3 = C_4 \cdot (4h^2 + 2hm)$$

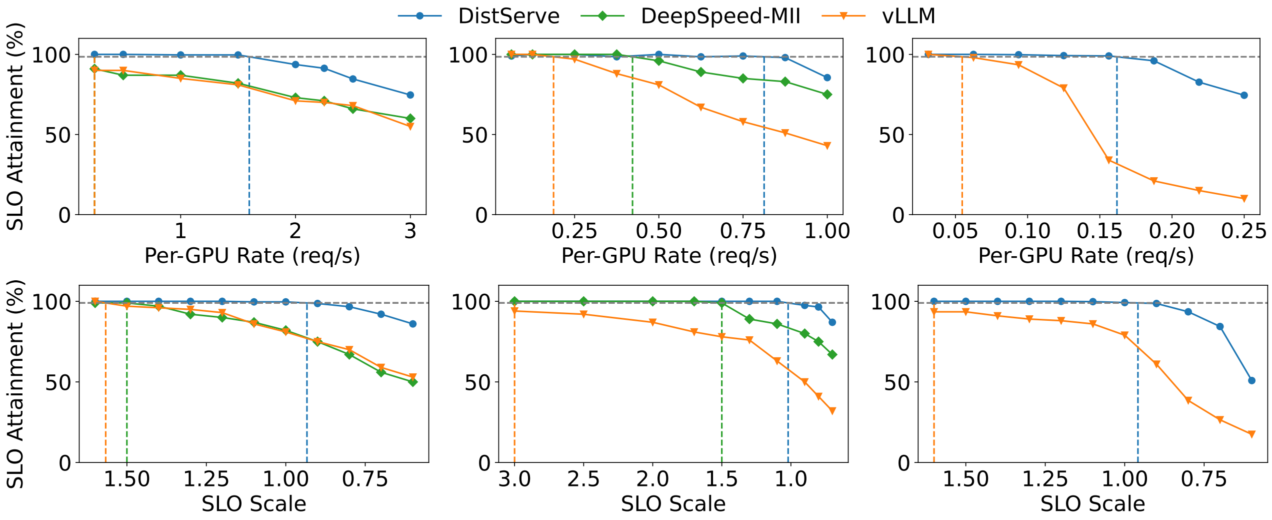
As for the decoding attention operation, for one attention head and a request with l generated tokens, it needs to perform $3sl$ memory reads and writes, alongside $2sl$ FLOPs. It is memory-bound, so we can model the latency of decoding attention as:

$$T_4 = C_5 \cdot n \cdot 3s \sum_{i=0}^{B-1} l_i = C_5 \cdot 3ht$$

Summing up, the latency of the decoding phase is:

$$T_{Decoding} = C_4 \cdot (4h^2 + 2hm) + C_5 \cdot 3ht$$

Here we do not introduce the overhead term (like C_3 in the profiling stage) because $4h^2 + 2hm$ is already a constant, and the overhead can be put into C_4. Similarly, we use profiling and interpolation to figure out the values of C_4 and C_5.

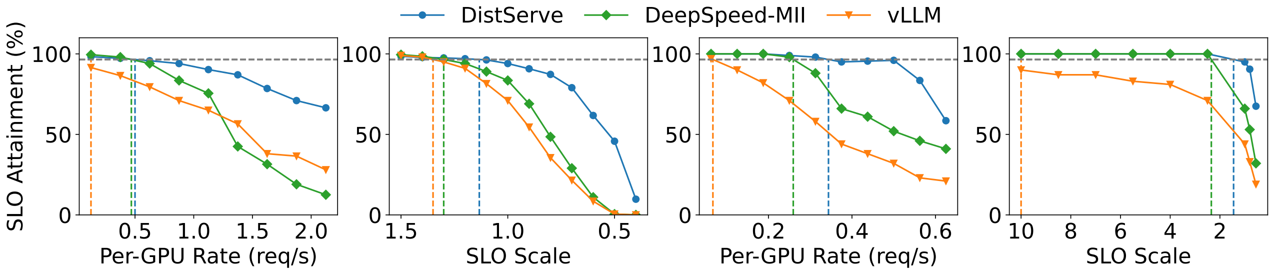


(a) OPT-13B

(b) OPT-66B

(C) OPT-175B

Figure 13: Chatbot application with OPT models on the ShareGPT dataset.



(a) Code Completion

(b) Summarization

Figure 14: Code completion and summarization tasks with OPT-66B on HumanEval and LongBench datasets, respectively.

B DistServe Placements in End-to-end Experiments

Table 3 shows the tensor parallelism (TP) and pipeline parallelism (PP) configurations for prefill and decoding instances chosen by DistServe in the end-to-end experiments §6.2.

Model	Dataset	Prefill		Decoding	
		TP	PP	TP	PP
OPT-13B	ShareGPT	2	1	1	1
OPT-66B	ShareGPT	4	1	2	2
OPT-66B	LongBench	4	1	2	2
OPT-66B	HumanEval	4	1	2	2
OPT-175B	ShareGPT	3	3	4	3

Table 3: The parallelism strategies chosen by DistServe in the end-to-end experiments.

C End-to-end Results under 99% SLO attainment

Figure 13 and Figure 14 show the end-to-end performance between DistServe and baselines with the same setup in §6.2 except that the SLO attainment goal is changed to 99%. We can see that under a more stringent SLO attainment goal, compared to vLLM, DistServe can still sustain 3×–8× higher rate and 1.24×–6.67× more stringent SLO. When compared to DeepSpeed-MII, DistServe can achieve 1.32×–8× higher rate and 1.20×–1.58× more stringent SLO.

ACCL+: an FPGA-Based Collective Engine for Distributed Applications

Zhenhao He
Systems Group, ETH Zurich

Dario Korolija
Systems Group, ETH Zurich

Yu Zhu
Systems Group, ETH Zurich

Benjamin Ramhorst
Systems Group, ETH Zurich

Tristan Laan[*]
University of Amsterdam

Lucian Petrica
AMD Research

Michaela Blott
AMD Research

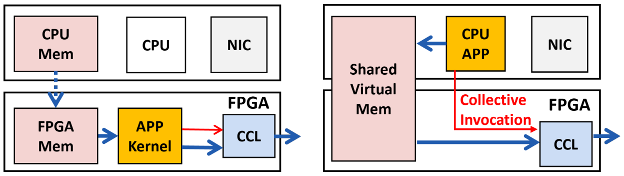
Gustavo Alonso
Systems Group, ETH Zurich

Abstract

FPGAs are increasingly prevalent in cloud deployments, serving as Smart-NICs or network-attached accelerators. To facilitate the development of distributed applications with FPGAs, in this paper we propose ACCL+, an open-source, FPGA-based collective communication library. Portable across different platforms and supporting UDP, TCP, as well as RDMA, ACCL+ empowers FPGA applications to initiate direct FPGA-to-FPGA collective communication. Additionally, it can serve as a collective offload engine for CPU applications, freeing the CPU from networking tasks. It is user-extensible, allowing new collectives to be implemented and deployed without having to re-synthesize the entire design. We evaluated ACCL+ on an FPGA cluster with 100 Gb/s networking, comparing its performance against software MPI over RDMA. The results demonstrate ACCL+'s significant advantages for FPGA-based distributed applications and its competitive performance for CPU applications. We showcase ACCL+'s dual role with two use cases: as a collective offload engine to distribute CPU-based vector-matrix multiplication, and as a component in designing fully FPGA-based distributed deep-learning recommendation inference.

1 Introduction

FPGAs are increasingly being deployed in data centers [16, 81] as Smart-NICs [29, 35, 64, 67, 103], streaming processors [31, 32, 55, 68], and disaggregated accelerators [15, 41, 61, 65, 86, 93, 115]. In scenarios where FPGAs are directly connected to the network, efficient distributed systems can be built using direct FPGA-to-FPGA communication. However, designing distributed applications with FPGAs is difficult. It requires both a network stack on the FPGA compatible with the data center infrastructure, and a higher level abstraction, e.g., *collective communication*, for more complex interaction patterns. Unlike in the software ecosystem where many such libraries exist [38, 76], there is a lack of similar resources for FPGAs. While new development platforms [53, 59] are improving FPGA programmability, and other recent efforts [10, 18, 60, 70, 72, 100] focus on virtualizing FPGA resources for abstracting data movement, they lack support for networking. This forces distributed applications on FPGAs to rely on the CPU for communication [22, 92, 116], thereby increasing the latency of data transfers between FPGAs. It has not been until recently that native networking support [14, 45, 56, 87, 95] has become available for FPGAs. But these systems lack collective communication, limiting their applicability in larger distributed use cases.



(a) For distributed accelerator

(b) For Smart-NIC

Figure 1: Collective communication library (CCL) in different FPGA-accelerated systems, where the blue line indicates application data flow and the red line indicates collective invocation commands.

Implementing high-performance and versatile collective abstractions for FPGAs poses several challenges:

Challenge 1: Support of Diverse Transport Protocols. This requirement stems from the need for application-specific solutions and to ensure interoperability in mixed environments where FPGAs coexist with CPUs and accelerators. The ability to adapt to various communication protocols is crucial for integrating FPGA-based components seamlessly with other parts of a system. Existing work [25, 26, 36, 44, 73, 84, 85, 101, 104, 112] is often tailored to scenarios where FPGAs are directly connected to each other rather than connected through a data center packet-switched network. In these approaches, communication is through low-level link-layer protocols, leading to scalability and integration challenges at a data-center scale.

[*] Work done during internship at AMD Research

Challenge 2: Flexibility for Collective Implementation. A key challenge is to provide a flexible design in hardware allowing the selection of different collectives and algorithms at runtime. Current solutions [26,45] often integrate all possible collective modules directly into FPGA hardware, resulting in static primitives that necessitate hours-long recompilations for any changes. Alternative approaches, e.g., collectives using embedded micro-controllers (uC) [89] on FPGAs, offer more flexibility at the expense of performance.

Challenge 3: Portability Across Applications and Platforms. Portability stands out as a key challenge, as FPGAs are used in a wide variety of configurations. Figure 1a shows the FPGA collective communication library (CCL) as enabler for direct networking between FPGAs accelerators. It also demonstrates a *partitioned memory model* [54, 59, 60] for FPGA-centric applications, where an explicit memory copy is required if the data originates from CPU memory (dashed line). Figure 1b illustrates CCL's role as an collective offload engine for a CPU application with *shared virtual memory* [62,70,75,106]. This portability also raises questions about which *communication models* should be provided to the application. Should interfaces support message passing, i.e., MPI, where communication occurs between memory buffers, or streaming, where communication flows through continuous data streams?

In summary, the key question to address is *how to effectively design a portable, flexible, high-level collective abstraction on FPGAs that can support various memory models (e.g., partitioned and shared virtual memory model), communication models (e.g., message passing and streaming), and transport protocols (e.g., TCP and RDMA), while accommodating a broad spectrum of applications*. Achieving this objective is complex, given the significant impact of these configurations on runtime, interfaces, and data movement. Moreover, given FPGAs' extended compilation times and lengthy hardware debugging cycles, we need a parameterized approach that allows a FPGA-resident CCL to be modified without recompilation, in order for the CCL to be practical in real-world use. Table 1 summarizes existing FPGA-based solutions, and all existing solutions have their own limitations.

Our Contributions. To address these challenges, we introduce ACCL+, an **A**daptive **C**ollective **C**ommunication **L**ibrary on FPGAs. ACCL+ can be used to enable direct communication between FPGAs and can function as a collective offload engine for the CPU. ACCL+ provides MPI-like collective APIs with explicit buffer allocation and streaming collective APIs with direct channels to the communication layer. To achieve portability, we employ a modular system architecture which decouples platform-specific IO management and runtime from the collective implementation, incorporating platform and network protocol-specific adapters and drivers. For flexibility, we have developed a platform and protocol-independent collective offload engine that supports modifying the collective implementation without hardware recompilation. We test ACCL+ on two platforms, a commodity, partitioned memory platform (AMD Vitis [59]) and a shared virtual memory platform (Coyote [62]). We choose AMD Vitis for its recent integration of high-performance 100 Gb/s UDP [107] and TCP [45] hardware stacks, aligning with our goal of leveraging cutting-edge networking capabilities for optimal communication performance. Coyote is used due to its unique provisioning of unified and virtualized memory across CPU-FPGA boundaries [62], coupled with comprehensive network services, including RDMA. ACCL+ is also designed to minimize control overheads, improve scalability, and facilitate simulation.

Table 1: Comparison of ACCL+ with FPGA-based solutions in terms of bandwidth, flexibility in implementing different collectives, target application scenarios, and supported transport protocols.

Solution	BW (Gb)	Flex.	Application	Protocol
Easynet [45]	100	Low	FPGA	TCP
SMI [26]	40	Low	FPGA	Serial Link
Galapagos [97]	10	Low	FPGA	TCP
ZRLMPI [85]	10	Low	FPGA	UDP
TMD-MPI [89]	<10	High	FPGA	Serial Link
ACCL+ (Ours)	100	High	CPU/FPGA	UDP/TCP/RDMA

Key Results. We first evaluate ACCL+ using micro benchmarks. ACCL+ achieves a peak send/recv throughput of 95 Gbps, almost saturating the 100 Gb/s network bandwidth. We evaluate collective operations under two scenarios: FPGA-to-FPGA distributed applications with FPGA kernels directly interacting with ACCL+ (F2F), and CPU-to-CPU distributed applications with ACCL+ as a collective offload engine (H2H). ACCL+ exhibits significantly lower latency in F2F scenarios compared to software RDMA MPI for FPGA-generated data. In H2H scenarios, ACCL+ has comparable performance to software RDMA MPI for CPU-generated data while freeing up CPU cycles and reducing pressure on CPU caches. Then, we examine ACCL+ with two use-case scenarios. First, distributed vector-matrix multiplication with CPU computation and ACCL+-based reduction, where ACCL+ improves performance compared to software MPI. Secondly, we show that ACCL+ enables the distribution of an industrial recommendation model across a cluster of 10 FPGAs, achieving more than two orders of magnitude lower inference latency and more than an order of magnitude higher throughput than CPU solutions. The use case study not only highlights ACCL+'s effectiveness in different scenarios but also paves the way for future research opportunities in investigating hybrid CPU-FPGA co-design for distributed applications.

2 Background

FPGA Programming. In the past, hardware description language (HDL - Verilog, VHDL) was the sole method to program FPGAs. With High-Level-Synthesis (HLS) [21], the programmability of FPGAs is enhanced by allowing the developers to program in C-like code with hints (pragmas) to infer parallel hardware blocks. Unfortunately, existing HLS-based libraries lack networking and collective abstractions.

Communication Models. Message passing, e.g., MPI, is a communication model for distributed programming on CPUs, whereby communicating agents exchange messages, i.e., user buffers, typically resulting from previous computation. This model can be applied to FPGAs [46, 84, 85], but a more common communication model for FPGAs is the streaming model. FPGA kernels support direct streaming interfaces, into which data can be pushed in a pipelined fashion during processing. Kernels executing on the same FPGA can stream data to each-other through low-level latency-insensitive channels, such as AXI-Stream [9]. The streaming model can be applied for communication across FPGAs, however, existing streaming communication framework [26, 33, 34] often do not have transport protocols or collective abstractions.

FPGA Development Platforms. Modern FPGA platforms adopt various virtualization methodologies [13, 83, 99] for FPGA resources. Most simplify development with a static *shell* for resource management and data movement, with some offering additional *services* like transport layer networking, and the host-device interaction relies on *runtime* libraries. This approach allows developers to concentrate on designing the application kernel. Many commodity platforms [54, 59] implement a partitioned memory model, which permits data movement from FPGA applications to FPGA memory while restricting direct access to host CPU memory. In contrast, shared virtual memory platforms, such as Coyote [62] and Optimus [70], offer a virtualized and unified memory space between CPU and FPGA.

Network-Attached FPGAs. FPGAs today feature 100 Gb/s transceivers, enabling direct processing of network data [81, 105]. FPGA-based Smart-NICs [2, 14, 28, 35, 66, 67, 98, 103, 119] perform programmable packet filtering but often leave the network stack to the CPU software, limiting their applicability to FPGA applications. Distributed machine learning [3, 12, 19, 77, 118], and data processing [24, 61, 65] applications capitalize on network-attached FPGAs. Following this trend, there is an increasing effort to develop hardware network stacks on FPGAs, such as UDP [47, 107], TCP [4, 27, 56, 87, 94], and RDMA [65, 69, 91, 95]. With the growing demands and the increasingly distributed nature of these applications, their communication patterns have become more complex, motivating the need for high-level collective abstractions on FPGAs. Therefore, simply offloading the network stack is often insufficient for complex applications in distributed settings.

3 Related Work

Collective for Accelerators. MPI implementation of collective communication are becoming more accelerator-aware, e.g., GPU-aware MPI [80, 102, 111] or FPGA-aware MPI [22]. In GPU-Direct RDMA collective libraries, e.g., NCCL [74] and RCCL [5], the network data can be directly forwarded to the GPU memory from the commodity RNIC via the shared PCIe switch, bypassing the CPU memory. However, FPGAs can connect directly to the network, and as such, the RDMA stack in ACCL+ resides completely in FPGA, eliminating the need for an external NIC. ACCL+ can provide streaming interfaces directly to FPGA kernels, in addition to the standard READs/WRITEs to memory used by commodity RNICs, therefore reducing latency by bypassing the memory hierarchy. Finally, the commodity RNICs typically lack offload capabilities, with collectives implemented on GPU cores, leading to computation and communication contention on the GPU which affects performance [51, 79]. Thus, ACCL+ could serve as a collective offload engine for GPUs in the future.

FPGA-based Collectives. Projects such as Galapagos [30, 97] and EasyNet [45] provide in-FPGA communication stacks for data exchange within a cluster, serving as a foundation for collectives without an external NIC. TMD-MPI [88, 89] orchestrates in-FPGA collectives using embedded processors, yet its bottleneck lies in control due to sequential execution in low-frequency FPGA microprocessors. Collective offload with NetFPGA [6–8] has been explored, but static collective offload engines limit flexibility and often rely on software-defined network switches for orchestration. SMI [26] proposes a streaming message-passing model, exposing streaming collective interfaces to FPGA kernels. While SMI enables kernels to initiate collectives directly, it employs dedicated FPGA logic for collective control, limiting flexibility for post-synthesis reconfiguration. In an earlier prototype, ACCL [46], we focused primarily on message-passing collectives for FPGA applications. However, the coordination of collectives required CPU involvement, it lacked significant streaming support, and was not tested at scale.

Collective Offload for CPUs. BluesMPI [11, 96] offloads collective operations to a BlueField DPU, demonstrating comparable communication latency to host-based collectives, but it does not target accelerator applications. The latency of ACCL+ targeting host data matches BluesMPI, even with BluesMPI ARM cores working at ten times the frequency.

Multi-FPGA Frameworks. Frameworks like ViTAL and its successors [112–114] propose FPGA resource virtualization and compilation flows for mapping large designs onto multiple FPGAs through latency-insensitive channels. OmpSs@cloudFPGA [25] introduces a multi-FPGA programming framework that partitions large OpenMP programs with domain-specific programs into smaller distributed parts for execution on FPGA clusters, providing communication through static, compile-time-defined send/recv and collective opera-

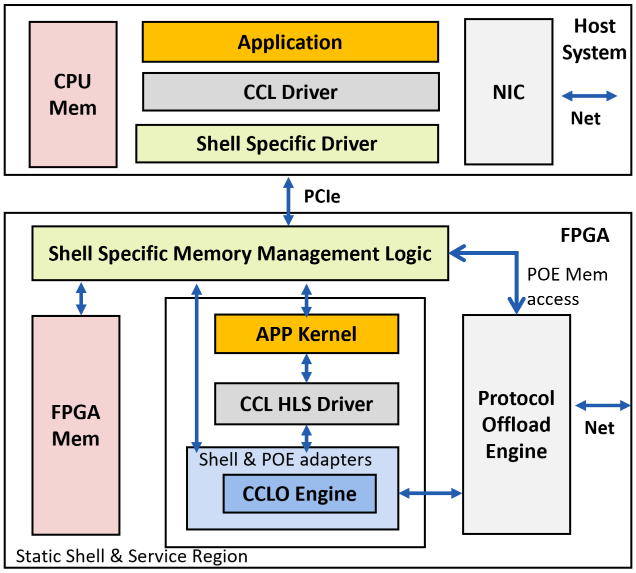


Figure 2: System overview of the FPGA-based collective communication library.

tions supporting only the unreliable UDP protocol. ElasticDF [3] and FCsN [40] present domain-specific frameworks for automatically distributing large neural network model inference across FPGAs with hardware UDP/TCP send/recv for FPGA-to-FPGA data movement. These projects are complementary to our work, and integrating ACCL+ will enhance their flexibility and performance.

4 ACCL+: An FPGA Collective Engine

ACCL+ is an FPGA-based collective abstraction designed for both FPGA and CPU applications, focusing on versatility and adaptability. Its primary goals include:

G1: Offering a standard collective API that abstracts different platforms and protocols from the application layer.

G2: Providing flexibility to dynamically select collectives and their algorithms at runtime, and to modify them without major architectural changes.

G3: Ensuring portability across various FPGA platforms and communication models for a wide range of applications.

G4: Supporting multiple transport protocols under a high-level collective abstraction.

G5: Providing high-throughput and low-latency performance for various collectives.

To achieve these goals, ACCL+ features a modular design that separates platform-specific and transport layer components from the core collective design. Its architecture includes layers of abstraction in both software and hardware, as shown in Figure 2, enabling a central CCL offload engine (CCLO) to adapt to diverse platforms and communication protocols. In this section we will describe each layer.

4.1 Application Interface

To satisfy **G1**, ACCL+ provides standard APIs for both CPU and FPGA applications. ACCL+ implements two drivers that offer similar, platform- and protocol-agnostic collective APIs for these scenarios. The ACCL+ drivers expose an MPI-like API, catering to the message-passing paradigm and facilitating the porting of existing MPI-based applications to ACCL+ collectives, and a streaming collectives API to overlap communication and computation in hardware.

ACCL+ Drivers. The host-side CCL driver allows initialization and runtime management of platform and ACCL+ data structures and hardware, as well as protocol offload engine (POE) initialization, i.e., setting up sessions for TCP or queue-pairs for RDMA. The CCL HLS driver is not capable of such initialization, and therefore the application must perform host-side initialization before any FPGA application kernels are started. We provide a more detailed description of ACCL+ initialization in Appendix A.

Listing 1: Reduce collective API in C++.

```
CCLRequest *reduce(BaseBuffer &buf, unsigned int
      count, unsigned int root, reduceFunction func,
      communicatorId comm_id, flagType flags);
```

MPI-like Collective API. This API require the application to store data in memory before invoking collectives. Listing 1 shows the MPI-like collective API, including arguments like datatype, buffer pointer, and element count, along with flags indicating buffer location (host or FPGA memory) and the option for synchronous calls. To facilitate portability, message passing collectives operate on an ACCL+ specific buffer class which can wrap normal C++ arrays with additional platform-specific information. Common collectives, such as reduce, broadcast, and barrier, are supported. Each MPI-like collective call in the host CCL driver has a corresponding HLS API call with a similar syntax for direct invocation from FPGA kernels.

Streaming Collective API. This API allows data to originate and terminate at the stream interfaces between the FPGA application kernels and the ACCL+ hardware, instead of in memory buffers.

Listing 2: Example kernel using streaming send in HLS.

```
// set up command and data interfaces
cclo_hls::Command cclo(cmd, sts, communicator);
cclo_hls::Data data(data_to_cclo, data_from_cclo);
// issue streaming send command without buffer argument
cclo.send(type, count, dst_rank);
// push data in streams to network without buffering
for (int i = 0; i < N; i++) {
    data.push(/* generate data */); }
cclo.finalize(); // wait for send completion
```

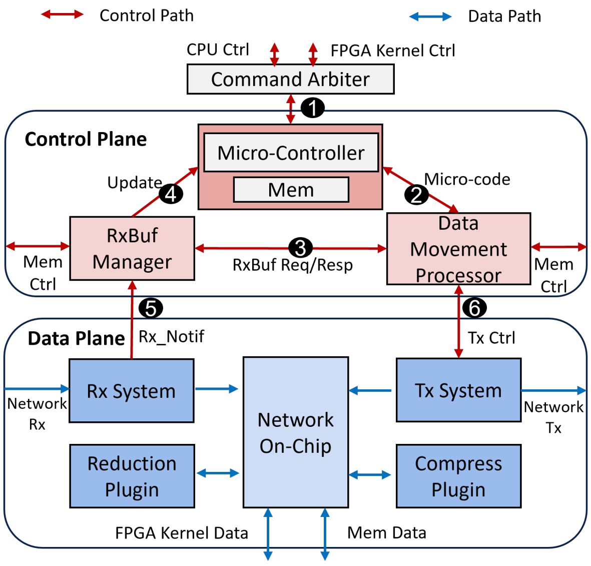


Figure 3: Hardware architecture of CCLO engine.

Listing 2 demonstrates an example FPGA kernel issuing a streaming send command to the CCLO engine (line 5) and subsequent pushes to the CCLO streaming data interface, 64B per cycle (line 8), followed by a wait for CCLO completion. The HLS-based streaming APIs are tailored for FPGA applications running in a streaming fashion and this code is synthesizable with HLS tools. HDL-based FPGA kernels can interact with the collective engine directly, through the same interfaces. Additionally, the host can also call streaming collectives via the host-side CCL driver.

4.2 CCLO Engine

Our approach to satisfying **G2**, i.e., achieving flexible collective implementation in hardware, differs from related work. One method deploys all collective modules in FPGA fabric simultaneously [26, 45], consuming extensive resources and not allowing modifications to the collective algorithms without recompiling the entire design. Another method pursues flexibility by implementing the collectives in embedded micro-controllers (uC) on FPGAs [89], which are often limited by a low clocking frequency, e.g., 200 MHz, and the sequential execution nature, thus sacrificing performance.

Our Approach. We utilize a hybrid approach that leverages the strengths of both methods. To ensure flexibility, low latency, and high throughput, the key design principle is to *decouple the CCLO logic into the flexible control plane and the parallel data processing plane*. The CCLO control plane is flexible, centered around an embedded uC [108], which enables the implementation of different collective algorithms through firmware updates without needing to refactorize the entire design and re-synthesize. The CCLO data plane contains independent latency-optimized hardware modules with wide data path for concurrent execution. Moreover, to further reduce the load on the uC, *we minimize its code footprint by offloading tasks* such as packet assembling and tag matching to hardware. Additionally, interactions with memory controllers are offloaded to dedicated hardware, preventing the uC from stalling during memory accesses. As a result, the uC handles a set of high-level data movement primitives that facilitate the implementation of the actual collective algorithms.

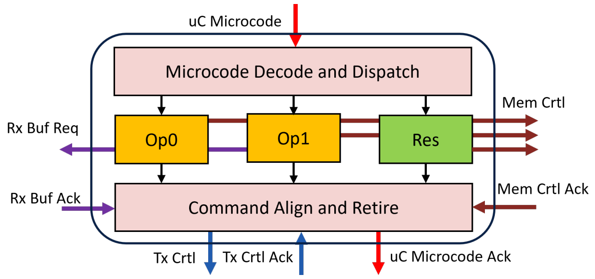


Figure 4: Architecture of the Data Movement Processor.

Figure 3 shows the overall architecture of the CCLO engine, which orchestrates the collective data movement through a set of standardized CCLO interfaces to interact with the application, the memory and the network. The CCLO accepts communication requests from the host or application kernels, communicates with the protocol offload engine, manages buffers in FPGA memory (HBM, DDR, BRAM), and manage data streams from other kernels.

4.2.1 Flexible Control Plane

The CCLO control plane contains a uC that issues high-level data movement commands to a hardware-accelerated data movement processor (DMP). The CCLO control plane also contains a RxBuf Manager (RBM), which manages temporary Rx buffers. The uC, DMP, and RBM store states in a small configuration memory implemented as FPGA BRAM. The configuration memory is also accessible by the CPU through MMIO and includes information about the communicator, e.g., session or queue pair IDs, pool of allocated Rx buffers. Besides, FIFO queues are incorporated into all command paths, allowing multiple in-flight instructions. Currently, these FIFO queues are set to a depth of 32, which can be further increased at compile time.

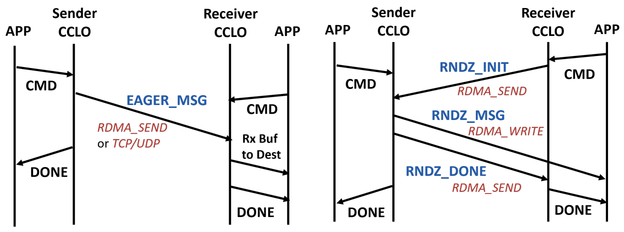
Collective Programming with Primitives in uC. The uC firmware implements various collective algorithms and synchronization protocols, such as eager and rendezvous, using high-level primitives. Each primitive instruction consists of three slots: two for operands (data entering CCLO) and one for the result (data exiting CCLO). This design aligns with common collective operations, e.g., reduce, which processes two inputs to produce one output. Unary operations like send can disregard one operand slot. Operand slots include opcodes and flags that define the data movement specifics, dictating *when* and *where* data should move. For instance, data can be moved immediately or upon arrival, sourced from a memory buffer when using the MPI-like API, or from the data interface of the FPGA kernel when using the streaming API. Additionally, the data can either be sent to a remote node through the network or remain local as intermediate results. These elements can be combined to cover nearly all data movement needs in collective operations.

Data Movement Processor. The primary purpose of the DMP is to conceal memory access latency from the uC, ensuring that the uC does not stall for memory accesses or wait for data streams, as shown in Figure 4. Upon the receipt of the microcode generated by the uC ❷, the DMP first decodes the microcode and dispatch the code to different compute units (CUs). The DMP primarily consists of three CUs, aligning with the structure of the primitive, each responsible for controlling one or more components in the datapath. If the microcode indicates to fetch data from memory and forward it to the network, the CU issues memory requests to the target address and then issues the Tx control ❻ to the data plane, ensuring the data plane waits for incoming memory streams to forward to the network. If the operand is expected to come over network and buffered in temporary buffers, the DMP also sends out requests periodically to the RBM to check if the message has arrived ❸. The DMP operates in a pipelined fashion, and each operand slot independently interprets its instruction fields, emitting commands for corresponding datapath blocks. Upon receiving acknowledgements from datapath blocks, the DMP signals instruction completion to the uC.

RxBuf Manager. The RBM alleviates uC load by autonomously managing temporary Rx buffers and reassembling messages from network packets, especially under the eager protocol. It uses a state table in FPGA on-chip memory and a set of finite-state machines (FSMs) to handle Rx buffers. Upon notification of incoming messages ❺, RBM checks the state table using the message ID. If the message is new, it identifies a free Rx buffer from the configuration memory and issues requests to store the message there. Since messages are often split into packets that may arrive interleaved, RBM uses the state table to piece together packets into complete messages in the appropriate Rx buffer. When a full message is assembled, RBM updates the exchange memory's buffer list ❹, marking it ready for retrieval, and stores essential metadata (source ID, tag, Rx buffer address) for tag matching, facilitating buffer identification by the DMP.

4.2.2 Parallel Data Plane

Rx and Tx System. In ACCL+, we implement a message protocol that includes a signature for each message. This signature contains metadata such as message type, destination rank, length, tag, and a sequence number to track the order of messages. The Tx and Rx systems feature a 512-bit wide data path and are responsible for packetizing and depacketizing the signature along with the user payload. They also issue commands to interact with the POEs. The processes of issuing commands, inserting signatures, and parsing can vary across different synchronization protocols. Both the Rx and Tx systems incorporate a finite state machine to manage these variations appropriately.



(a) Eager protocol (b) Rendezvous protocol

Figure 5: CCLO eager and rendezvous with send/recv.

Network On Chip. All the data streams internal to the CCLO can be routed in the granularity of packets based on the dest field that comes along with the data.

Streaming Plugins. The plug-ins are utilized for applying unary and binary operations to in-flight data and can be enabled at compile time. Binary operations are typically utilized to implement reductions - sum, max, etc. Unary operators may implement compression or encryption. Each of the plug-ins is a streaming kernel and may implement more than one function, in which case the control plane will specify the desired function by setting its dest field of the plugin input stream.

4.2.3 Message Synchronization Protocol

The CCLO supports two distinct message synchronization protocols: eager and rendezvous. Thanks to its flexible design, both protocols can be tuned dynamically at runtime.

The eager protocol allows the sender CCLO to immediately send data upon receiving a command, and the receiver buffers the data in the CCLO Rx buffer before moving it to its destination (either in memory or in FPGA kernel streams depends on runtime configuration), as shown in Figure 5a. This protocol is preferred for small messages to minimize latency since there is no handshake phase and small message sizes incur little overhead. We implement the eager protocol using UDP/TCP or two-sided RDMA verb.

In contrast, the rendezvous protocol requires resolving the result buffer address before transmission, as shown in Figure 5b. Once resolved, data is directly placed into the destination, eliminating the need for temporary buffering. We use two-sided RDMA `SEND` for rendezvous handshake messages, and we use one-sided RDMA `WRITE` for actual message transmission bypassing the intervention from the receiver uC. Given that one-sided RDMA operations are transparent to the receiver uC, a key design decision is how the uC should detect message arrival. One approach is to make the uC pe-

Table 2: Algorithms used for example collectives.

Collective	Eager	Rendezvous
Bcast	One-to-all	One-to-all;Recursive doubling
Reduce	Ring	All-to-one;Binary tree
Gather	Ring	All-to-one;Binary tree
All-to-all	Linear	Linear

riodically poll the destination buffers in memory. However, this approach increases uC overhead and latency, especially since buffers may be located in various memory systems like CPU or FPGA memory. Additionally, if the destination is a streaming interface rather than a buffer, polling is not feasible. Therefore, we choose an alternative method: the sender uC dispatches a small control message using two-sided RDMA `SEND` immediately after the one-sided RDMA `WRITE`. This control message is processed by the receiver uC to confirm the completion of the data transfer. Though not depicted in Figure 3, the uC contains specific ports that directly interact with the data plane for rendezvous handshake and control messages, bypassing the RBM and DMP. These command paths also incorporate FIFO queues.

4.2.4 Collective Algorithms

We provide different implementations for various collectives, and users can define their own. Collectives are realized by specifying a communication pattern as a C function in uC firmware, and then executing this pattern through instructions in DMP and Tx/Rx System on each FPGA in the communicator. Table 2 summarizes the algorithms and communication patterns used to implement collectives. For eager protocols with unreliable transmission (e.g., UDP), we currently use simple algorithms like ring and one-to-all to minimize the chances of packet loss. Future firmware improvements can enhance POE awareness for finer-grained algorithm selection. In contrast, when using RDMA, the rendezvous protocol employs more advanced algorithms like tree or recursive doubling. The token-based flow control in RDMA makes it well-suited for these sophisticated algorithms in the rendezvous protocol. For broadcast, we implement a simple one-to-all algorithm with small rank size, while with large rank size, we adopt more advanced recursive doubling such that the data transmission is not bottlenecked at the root rank. For gather and reduce, we apply a similar strategy. With small message size, we adopt an all-to-one approach to reduce the number of intermediate hops needed. On the other hand, with larger message sizes, to avoid a potential in-cast problem with the all-to-one approach, we adopt a tree-based algorithm. Tuning of the algorithms for specific collectives can be done at runtime through configuration parameters to the CCLO engine.

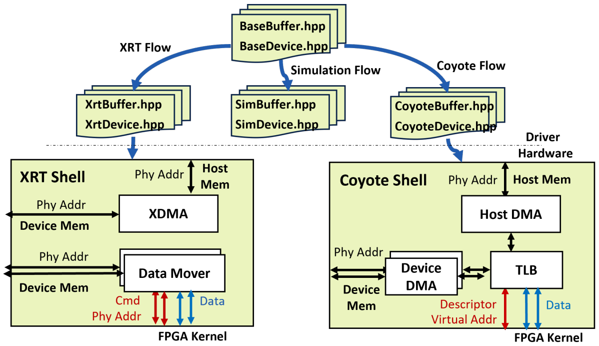


Figure 6: CCL driver for different memory managements.

4.3 ACCL+ Platform Support

A platform is defined by a software interface specification, defining how FPGA memory is allocated and manipulated, and how FPGA kernels are called, and a hardware interface specification, i.e., how FPGA kernels, including the CCLO, plug into hardware services in the FPGA. To facilitate portability between platforms and to satisfy **G3**, the ACCL+ host CCL driver layers the APIs on top of generic class types, such as BaseBuffer for memory allocation and data movement between host and FPGA, and BaseDevice for CCLO invocation. These are specialized to individual platforms through class inheritance, as illustrated in Figure 6. Each specific CCL class interfaces with platform-native drivers and employs distinct processes for handling data movement. ACCL+ supports both the commercial AMD Vitis platforms and the open-source Coyote platform [62], as well as a virtual simulation platform. New platforms can be added easily.

Integration with Coyote. Coyote utilizes a shared-memory model with a central memory management logic governed by a software-populated translation lookaside buffer (TLB). This TLB records allocated pages and facilitates virtual-physical address translation. The FPGA kernel issues memory requests through a descriptor interface, using virtual addresses directed to either host or device memory. The TLB interprets these requests, interacting with host DMA or device DMA based on the physical location of the memory page, forwarding the data to FPGA applications in a streaming manner. If a memory page is not registered during TLB lookup, it triggers an interruption to the CPU, resulting in a page fault and introducing a performance penalty. Therefore, the CCL driver, specifically the CoyoteBuffer class, eagerly maps pages to the Coyote TLBs when instantiating buffers. We modified Coyote, during integration, to increase the associativity of the TLB cache and expand the number of streaming interfaces Coyote provided to a single application region, from a single interface to three interfaces which is required by the CCLO engine. We also implemented a Coyote-specific adapter to convert from CCLO (R)DMA request syntax to Coyote-specific syntax, as indicated in Figure 7.

Integration with Vitis. Vitis platforms implement a partitioned memory model and the Xilinx Runtime (XRT) [59] is utilized by the CCL driver for low-level interaction with the platform. A XRT-controlled XDMA IP core [110] moves data between host and FPGA memory, while FPGA memory is accessed by FPGA kernels through Data Movers [109]. The CCLO memory interfaces align with the Data Mover interfaces, eliminating the need for dedicated memory interface adapters for the Vitis platform. As a result of the partitioned memory, the CCL driver explicitly migrates buffers between host and FPGA memory prior to or after the collective execution if the data originally resides in host memory - a process denoted staging. Staging creates performance penalty when ACCL+ collectives target host memory, as observed by related work on collective offload on DPUs [96]. Therefore, Vitis platforms favor distributed FPGA applications where data is streamed or resides in FPGA memory.

Simulation Platform We implemented an additional simulation platform for debugging and performance optimization. This simulation platform roughly models a Vitis platform, whereby FPGA chip interfaces (XDMA, Ethernet) are replaced by ZMQ [48] interfaces. A stand-alone simulated FPGA node is compiled to include memory and one ACCL+ CCLO Engine. The ACCL+ host driver includes dedicated buffer and device abstractions capable of connecting to the simulated node via ZMQ. ACCL+ provides convenient launch scripts to set up a simulated cluster of such simulation nodes.

The simulated nodes connect to each other through ZMQ rather than real Ethernet. While the simulated ZMQ network may lack realistic features like packet loss and reordering, it serves as a valuable functional simulation.

ACCL+ provides two simulation levels of the CCLO engine: functional simulation using compiled ACCL+ HLS source code and C firmware, and cycle-accurate (but slow) simulation using Verilog HDL generated from compiling the CCLO HLS code and firmware. For FPGA applications requiring streaming data exchange between FPGA kernels and the CCLO, we provide a bus functional model of the CCLO that connects via ZMQ to the simulated node.

4.4 Protocol Offload Engine

To satisfy **G4**, ACCL+ supports several 100 Gb/s protocol offload engines (POE) in hardware: UDP [107] and TCP [45] on Vitis platforms, and all the network services provided by Coyote. Notably, ACCL+ supports collectives with RDMA by leveraging the unified and virtualized memory space across the FPGA and the CPU provided by Coyote. All the POEs expose streaming control and data interfaces to other modules and some POEs (e.g., TCP) require direct memory access for packet buffering for re-transmission. For portability, the CCLO Engine has a set of POE-independent internal interfaces - two pairs of meta and data streaming interfaces (one for Tx and one for Rx). The meta interfaces contains various sub fields to indicate the op code, data length, communication session IDs, etc. The meta interfaces are then adapted to the POE interfaces with dedicated FPGA components as exemplified in Figure 7. The selection of the POE and its adapters is a compile time parameter of the CCLO Engine.

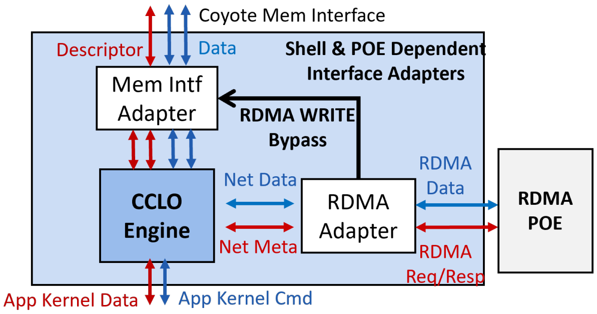


Figure 7: ACCL+ with Coyote-RDMA data path with corresponding POE and memory adapters.

Coyote RDMA POE. It supports standard RDMA verbs, including one-sided operations like `WRITE` and two-sided operations like `SEND`. The RDMA POE incorporates various streaming interfaces for RDMA commands, memory commands, and data. *Default Configuration:* On the initiating side of a `WRITE` operation, the RDMA POE issues memory requests directly to the Coyote memory management logic, fetching data from either host or device memory and streaming it through the network. On the passive side of `WRITE`, the data is directly written to virtualized memory. *ACCL+ Integration:* In the ACCL+-enabled configuration, the CCLO engine acts as a "bump-in-the-wire" engine between the memory management unit and the RDMA POE, as shown in Figure 7. On the initiating side of a `WRITE`, the CCLO engine issues RDMA commands and is responsible for data preparation, either fetching from memory or obtaining it from the application kernel in the form of streams. On the passive side, data bypasses the CCLO and is directly forwarded to the memory management unit. For single-sided `WRITE`, streaming into the application kernel is also possible by configuring the datapath at compile time. The CCLO engine consistently manages data and metadata streams from two-sided `SEND`. For CCL driver with RDMA, a queue pair needs to be exchanged between each node and needs to be registered to the RDMA POE.

TCP POE. The TCP POE supports up to 1,000 connections and can be configured to support window scaling and out-of-order packet processing. As a reliable transmission protocol, the TCP POE also needs to access protocol-internal buffers for re-transmission. The CCLO engine prepares and accepts all the data streams with the TCP POE. For CCL driver with TCP POE, a TCP session needs to be established between each node to construct the communicator.

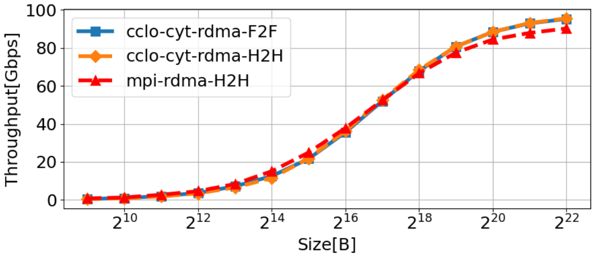


Figure 8: Send/Recv throughput comparison.

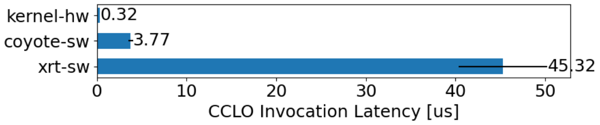


Figure 9: CCLO invocation latency from different parts.

5 Microbenchmark Evaluation

We evaluate ACCL+ on a heterogeneous cluster with 10 AMD EPYC CPUs and 10 attached FPGA cards (Alveo-U55C). Each CPU is equipped with a 100 Gb/s Mellanox NIC, while each FPGA features a 100 Gb/s Ethernet interface. All devices are connected to Cisco Nexus 9336C-FX2 switches. Evaluation scenarios consider data residing on the FPGA for distributed FPGA application (suffix *F2F*) and on the CPU for distributed CPU applications (suffix *H2H*). For F2F, the FPGA application data traverses the network directly through ACCL+. As a baseline, the FPGA data initially is moved to the CPU memory and then is transmitted via a commodity NIC. In H2H, the CPU application data is transferred to the FPGA and then transmitted with ACCL+. This is compared to transmitting the CPU data directly with a commodity NIC. We use the notion of *cclo* with different suffixes to indicate different configurations of ACCL+. The focus of these experiments is evaluating RDMA running with Coyote (suffix *cyt*) due to space limitations. We nevertheless present some results with ACCL+ running TCP on top of the Vitis XRT (suffix *xrt*) platform to compare it to ACCL [16], which utilizes an embedded micro-controller to orchestrate collective operations. Experiments configure both MPI-like collectives with memory pointers and streaming collectives. For the H2H experiments, MPI-like collectives are mandatory, while the F2F experiments are configured to run with streaming collectives. ACCL+ operates at 250 MHz in micro-benchmarks, with varying frequency in the use-case study due to the design complexity. The comparisons involve MPICH 4.0.2 with TCP and OpenMPI 4.1.3 compiled with RDMA using OpenUCX 1.13.1 on the cluster CPUs and Mellanox 100 Gb/s NICs. MPI libraries self-configure for collective algorithms and synchronization protocols. Each micro benchmark experiment is averaged over 250 runs.

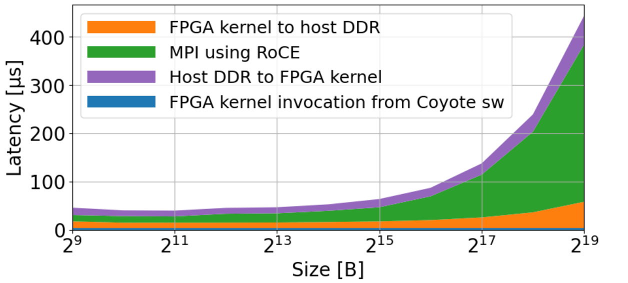
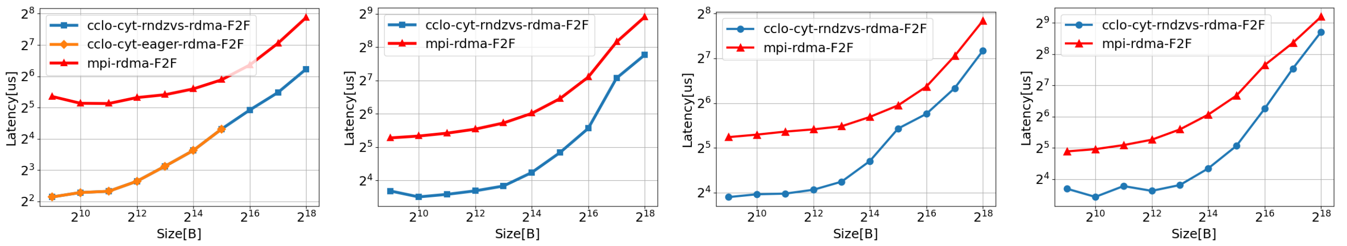


Figure 10: Latency breakdown of broadcasting FPGA produced data using software MPI with eight ranks with Coyote.

Send/Recv Throughput. We first evaluate pure throughput using send/recv. Figure 8 shows the throughput comparison of ACCL+ with Coyote RDMA and software MPI with RoCE backend. Notably, ACCL+ with RDMA achieves a peak throughput of 95 Gb/s, nearly saturating the network bandwidth. Compared to software MPI variants, ACCL+ exhibits comparable and slightly higher peak throughput. This is attributed to the FPGA network stack's ability to process network packets at line-rate in a pipelined fashion. Moreover, there is minimal distinction between F2F and H2H for ACCL+, thanks to the unified memory space provided by Coyote and both host memory access through PCIe and FPGA memory access offer higher bandwidth than the network.

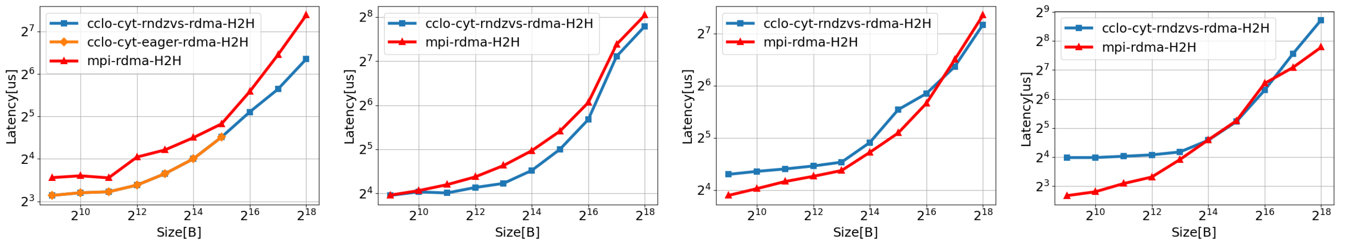
Invocation Latency. Figure 9 shows the invocation latency of the CCLO engine to execute a dummy NOP operation, which includes the time from receiving request untill the acknowledgement. For FPGA kernels that can directly interact with the CCLO engine, the invocation latency is minimal compared to software invocation from the host, showing a clear benefit of bypassing host control with FPGA-based applications. Coyote software driver contains a thin and optimized layer for invocation and scheduling and the resulting CCLO invocation time mainly consists of a PCIe write and a PCIe read latency. In contrast, the XRT invocation latency is significantly higher as it is not intended for fine-grained data movement.

FPGA-to-FPGA with Software MPI. To enable a more direct ACCL+ vs. software MPI comparison for executing collectives between kernels on FPGA, we model the execution time for MPICH- and OpenMPI-based device-to-device data movement, which includes: (1) moving data from FPGA HBM/kernel to host DDR through the PCIe, (2) executing the collective using software MPI, (3) moving data from host DDR to FPGA HBM/kernel, and (4) invoking the next computation kernel. We use the CCLO host invocation time as an approximation of the invocation time of other computation kernels. We measure the duration of each of the above steps and present a break-down of execution time of the collective with Coyote platform in Figure 10. We could observe that the PCIe transfer time is dominant for small messages while the collective time is dominant for large messages. Such



(a) Broadcast (b) Gather (c) Reduce (d) All-to-all

Figure 11: Collective latency comparison between ACCL+ RDMA and software MPI RDMA with eight ranks and device data



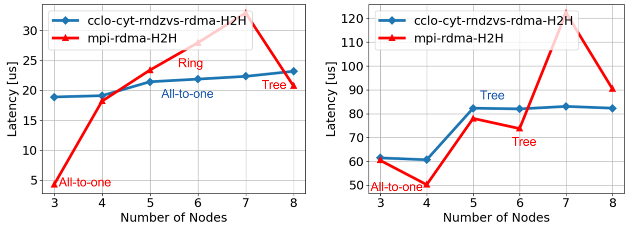
(a) Broadcast (b) Gather (c) Reduce (d) All-to-all

Figure 12: Collective latency comparison between ACCL+ RDMA and software MPI RDMA with eight ranks and host data

breakdown for XRT platform can be derived by changing the Coyote invocation latency to XRT invocation latency.

F2F Collective Latency RDMA. Figure 11 illustrates the latency of ACCL+ RDMA collectives with various message sizes on eight Alveo-U55C boards. This is compared to FPGA-to-FPGA data movement with software MPI over RDMA. For clarity, we present experiments showcasing better performance between eager and rendezvous collectives. The algorithms for each collective in ACCL+ are detailed in Table 2. Notably, ACCL+ exhibits significant performance benefits compared to its software counterpart. This advantage stems from the hardware's efficient execution of collectives and the direct network access within the FPGA device, eliminating the need for data copying to CPU memory.

H2H Collective Latency RDMA. Figure 12 illustrates a latency comparison between ACCL+ and software MPI targeting host data. The performance gains with ACCL+ vary across different collectives. Notably, for operations like broadcast and gather, ACCL+ consistently outperforms software MPI across a range of message sizes. However, for other collectives such as reduce and all-to-all, ACCL+ shows only marginal benefits and, in some cases, falls short of software MPI performance. One reason is that software MPI adapts its algorithms more finely to different configurations, whereas ACCL+ currently supports only a limited set of options. However, by offloading collective to hardware, CPU cycles could be freed for other computation tasks. Besides, by comparing ACCL+ F2F and H2H performance, we could observe that the ACCL+ collective latency has minimal difference because Coyote with unified memory allows direct memory access to both host and FPGA memory.

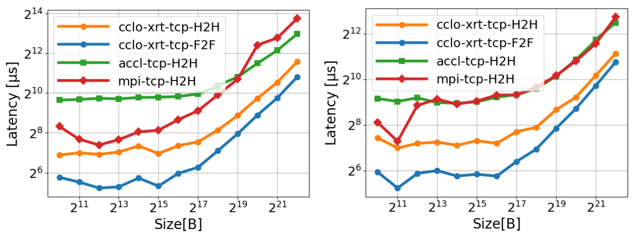


(a) 8KB message size. (b) 128KB message size.

Figure 13: Latency vs. rank sizes (Reduce).

Effect of Synchronization Protocol. Despite the simpler algorithms used by most eager collectives, such as one-to-all or ring, we observe that eager collectives can sometimes outperform rendezvous collectives with small message sizes, as seen in broadcast. This is because eager collectives do not require a handshake to resolve addresses.

Collective Algorithm and Scalability. Figure 13 illustrates the impact of algorithm selection and scalability on both ACCL+ and software MPI during collective executions. For an 8 KB message size, ACCL+'s reduce operation adopts an all-to-one algorithm, resulting in minimal latency increase across nodes. However, recognizing potential bottlenecks at the root node with this approach, ACCL+ switches to a binary tree algorithm for larger message sizes, such as 128 KB. In this case, an increase in latency is observed after



(a) Gather.

(b) Reduce.

Figure 14: Comparison of collective performance between ACCL+ TCP with XRT, software MPI TCP and ACCL TCP.

four nodes, stabilizing until eight nodes due to a consistent tree depth. On the other hand, software MPI exhibits a more fine-grained approach to algorithm selection based on the scale of the message size and the number of nodes. For instance, it deploys three distinct algorithms within the 8 KB range: an all-to-one algorithm for fewer than four nodes, a ring protocol for four to eight nodes, and an optimized binomial algorithm for 8 nodes. Additionally, for larger messages, software MPI switches between an all-to-one algorithm below three nodes and a binomial tree algorithm between four and eight nodes. This fine-grained algorithmic tuning contributes to its superior performance in certain H2H scenarios. While software MPI's approach involves detailed algorithmic tuning, ACCL+'s flexible design allows for potential future enhancements through additional fine-grained tuning to further optimize performance.

XRT Platform and TCP. In Figure 14, we evaluate ACCL+ TCP with the XRT platform and compare it against software MPI with TCP. We also include a comparison with ACCL [46] collectives, which employs a similar embedded processor to orchestrate collectives and supports TCP on the XRT platform. Notably, ACCL+ TCP consistently outperforms its software counterpart across all configurations, benefiting from the line-rate processing capabilities of a hardware TCP POE. Furthermore, ACCL+ demonstrates superior performance compared to ACCL. While both ACCL+ and ACCL utilize embedded microprocessors for collective orchestration in hardware, ACCL+ distinguishes itself by offloading more tasks to the hardware data plane, such as utilizing the RxBuf Manager for packet assembling. In contrast, ACCL relies more on the microprocessor, leading to lower performance. When comparing ACCL+ TCP for serving host applications and device applications, a significant overhead is observed for host applications. This is attributed to the limitation of XRT platform, which prohibits direct data movement from the FPGA kernel to host buffers, resulting in a memory-copy overhead. Additionally, the XRT software invocation latency is notably higher, as indicated in Figure 9.

Table 3: Parameters of the target recommendation model.

Tables	Concat Vec Len	FC Layers	Embed Size
100	3200	(2048, 512, 256)	50GB

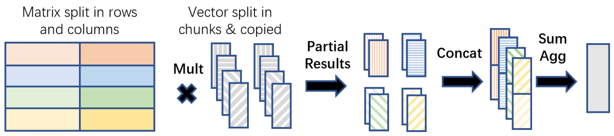

Matrix split in rows and columns
Vector split in chunks & copied
Mult
Partial Results
Concat
Sum Agg

Figure 15: Checkerboard block decomposition.

6 Case Study: Deep Learning Recommendation Model

Deep Learning Recommendation Models (DLRM) are widely used in personalized recommendation systems for various scenarios [23, 37, 117]. The structure of a DLRM includes two major components: memory-bound embedding layers and computation-bound fully-connected (FC) layers. These models handle both dense and sparse features, with the latter stored as embedding vectors in tables. In inference, these vectors are accessed via indexes, resulting in multiple random memory accesses. The retrieved embedding vectors are then concatenated with dense features and passed through several FC layers to predict the click-through rate, incurring heavy computational loads due to vector-matrix multiplication.

DLRM has been a focal point for acceleration on GPUs and FPGAs, given that CPU solutions are generally constrained by both random memory access and computation [43, 49, 63]. GPU-based solutions [42, 49, 52, 58] mostly accelerate the computation-bound FC layers to gain high throughput. However, the large batch sizes required for efficient GPU computation, coupled with random memory access, often lead to increased latency (tens of milliseconds). FPGA-based techniques [57, 71] overcome the embedding lookup bottleneck by distributing tables across memory banks and enabling parallel accesses, leveraging high-bandwidth-memory (HBM) and on-chip memory (BRAM/URAM). However, this approach is constrained by the requirement for embedding tables to fit within a single FPGA's memory (e.g., 16 GB HBM on AMD Alveo-U55C), limiting the size of embedding layer. Additionally, the finite computational resources on a single node pose restrictions on overall throughput for all FC layers.

6.1 Distributed DLRM Inference

We aim to demonstrate that ACCL+ can facilitate distributing DLRM inference across FPGAs to accommodate larger embedding layers, as in many large-scale industrial settings, while at the same time achieving low latency and high throughput. Table 3 shows the detailed configuration of such an industrial-level recommendation model [58]. In such a use

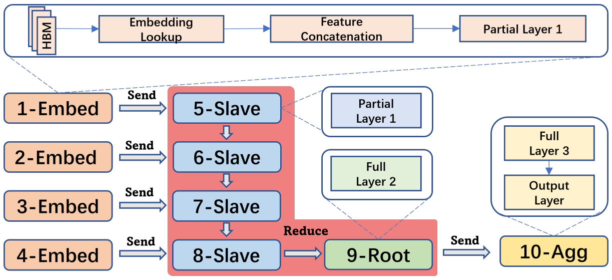


Figure 16: Conceptual design of partitioned DLRM, with $FC1$ decomposed and $FC2$, $FC3$ pipelined across nodes.

case, the embedding table does not fit into a single FPGA HBM and therefore both the embedding lookup and the computation are distributed across the network. This poses significant challenges for performance, scalability, and networking.

Vector-Matrix Multiplications Decomposition for DLRM. The computation pattern in DLRM inference involves a chain of three vector-matrix multiplications, with the inference output vector computed as a sequence of operations involving three matrices of FC layers ($FC1$, $FC2$, $FC3$) and a concatenated embedding vector. The concept of distributed vector-matrix multiplication has been extensively studied in literature [90] across CPUs and the same principle can be applied to an FPGA cluster. One common approach is *checkerboard block decomposition of matrix*, as shown in Figure 15. This method involves partitioning the matrix in terms of both rows and columns, while partitioning the vector ensures that processes associated with the same matrix row partition share the same sub-embedding vector. Each process can then perform partial computations, and the results belonging to the same row partition are concatenated and subsequently aggregated.

Decomposed and Pipelined Distributed DLRM. The partitioning strategy for the DLRM considers the need for balanced resource utilization, ensuring that the overall throughput is not limited by any process among all nodes. Typically, the computation load of the $FC1$ is significantly larger than subsequent layers like $FC2$ and $FC3$. To accommodate this, resource distribution should reflect the varying computation requirements. Additionally, for modern FPGAs with HBM, the capacity requires a minimum number of FPGAs to effectively store the embedding layer. A conceptual partitioned DLRM is illustrated in Figure 16. In this scenario, $FC1$ is decomposed and distributed across multiple FPGAs using the checkerboard block decomposition, and $FC2$ and $FC3$ are assigned to one FPGA each. The embedding tables are evenly distributed across nodes 1-4, with partial vectors transmitted to nodes 5-8, leveraging the network's low latency. Similarly, partial results computed on nodes 1-4 are forwarded to corresponding nodes 5-8, where an overall reduction of all partial $FC1$ results is conducted. The aggregated $FC1$ results are then forwarded to node 9 for $FC2$ computation, followed by node 10 for $FC3$ computation and final processing. Scaling resources according to the computation distribution requirements of each layer could lead to improved performance. For example, increasing the allocation of FPGAs for different layers based on their computational load. Such partitioning method requires diverse communication patterns by each node, such as send-only, send/recv, and reduction and ACCL+ provides a unified design supporting all the communication requirements of the DLRM through a standard interface. Additionally, for nodes that do not require reduction, the streaming reduction plugins of ACCL+ can be removed with a compilation flag, reducing resource consumption and improving routing and timing. Furthermore, the cross-node simulation provided by ACCL+ can facilitate the development process, reducing hardware debugging cycles.

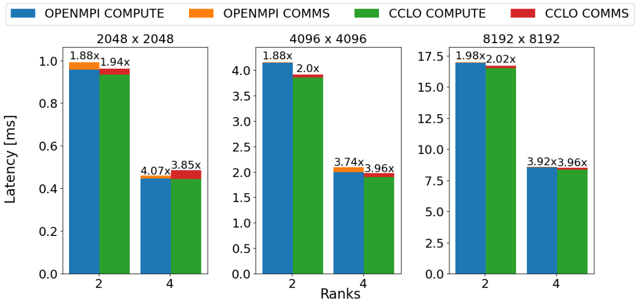
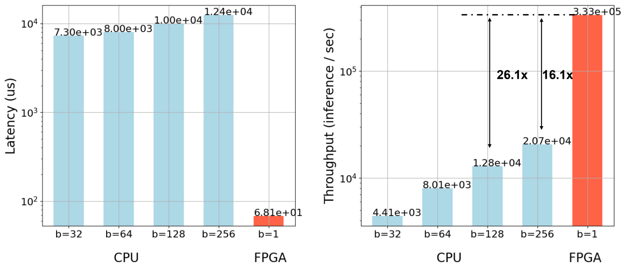


Figure 17: Speedup comparison and latency breakdown of distributed vector-matrix multiplication.

6.2 Use Case Evaluation

Distributed FC Layer Execution on CPU. We use an illustrative example to demonstrate how ACCL+ can be utilized to improve the efficiency of distributed work executing on CPU. In this use case, we distribute an FC layer workload (matrix-vector multiplication) by partitioning the weight matrix column-wise, with each rank receiving part of the input vector and a subset of the weight matrix columns. The matrix-vector product is obtained by summing the partial rank products using the reduce collective. For the implementation, we use the highly optimized Eigen library [39], distributing it with both ACCL+ RDMA and MPI RDMA. In this experiment we do not overlap computation and communication.

The overall execution time of the distributed FC layer is compared to its single-node execution, as depicted in Figure 17, where top-of-bar numbers indicate the speed-up compared to single-node execution. We observe that utilizing ACCL+ instead of MPI for the reduction generally results in lower matrix-vector computation time. This performance increase is most likely due to reduced pressure on the CPU cache, as ACCL+ utilizes FPGA memory for all intermediate reduction data structures. The figure indicates two instances of super-linear scaling, attributed to the weight matrix partitions fitting into either L2 (8 MB) or L3 (128 MB) caches



(a) DLRM latency.

(b) DLRM Throughput.

Figure 18: ACCL+ DLRM performance comparison.

on the CPU after partitioning, whereas the entire matrix did not fit in caches during single-node execution. The reduction time itself is higher in most cases due to an additional copy required to move data between Eigen result buffers and ACCL+ buffers, which can be eliminated with further optimization. Overall, distributing work with ACCL+ achieves lower latency, especially for specific configurations of FC size and number of ranks.

Distributed FPGA-based DLRM. We distribute an industrial DLRM model, as in Table 3, with ACCL+ on 10 U55C FPGAs following the same design principle as shown in Figure 16. The communication between the embedding node and the reduce slave node during each inference requires the transmission of a 3.2 KB partial embedding vector and a 4 KB partial result. Additionally, the reduction process spanning nodes 5 to 9 operates with a message size of 8 KB per inference. The achieved operating frequency is 115 MHz. We utilize 32-bit fixed-point precision for computation. All the application kernels utilize streaming collective APIs to interact with ACCL+. ACCL+ DLRM is configured with the TCP backend from XRT. Though the communication latency could be further optimized with ACCL+ RDMA, it is not on the critical path of overall latency as it is overlapped with computation. We also compare with CPU implementation [58], where the DLRM inference is run on an Intel Xeon Platinum 8259CL CPU @ 2.50 GHz (32 vCPU, Cascade Lake, SIMD supported) and 256 GB DRAM with TensorFlow Serving enabled. Figure 18(a) shows the latency comparison between ACCL+ and the CPU baseline. We evaluate various batch sizes on the CPU. On the other hand, ACCL+ works with streaming data without batching. The hardware implementation demonstrates two orders of magnitude lower latency compared to the CPU. This substantial latency reduction in the hardware implementation is attributed to the parallel arithmetic units in hardware and the significant latency introduced by random memory accesses. Figure 18(b) shows the throughput comparison. ACCL+ shows more than an order of magnitude higher throughput compared to CPU baseline.

Table 4: Resource utilization.

Component	CLB kLUT	DSP	BRAM	URAM
U55C(100%)	1303	9024	2016	960
CCLO	12.1%	1.6%	5.7%	0
TCP POE	19.8%	0	10.6%	0
RDMA POE	13.0%	0	5.3%	0
DLRM FC1	278.1%	580.1%	186.3%	798.3%
DLRM FC2	29.6%	85.1%	34.2%	97.9%
DLRM FC3	6.2%	16.1%	2.2%	20.8%

6.3 Resource Consumption

The resource utilization of ACCL+ components and the overall utilization of DLRM across nodes are summarized in Table 4. In the ACCL+ subsystem, the majority of resources are allocated to POEs, with the TCP POE being the most resource-intensive, while the CCLO engine utilizes comparatively fewer LUT and memory resources. DLRM utilization is categorized by different layers, and the presented utilization values represent the sum across multiple FPGAs after decomposition. Note that DLRM $FC1$ utilization exceeds 100%, reflecting the decomposition across 8 FPGAs (max 800%). The primary resource bottlenecks for DLRM are URAM, serving as fast on-chip memory for storing small embedding tables, and DSP, essential for matrix computations.

7 Discussion

In this paper we have explored the design of ACCL+ targeting efficient and high-speed offload of MPI-like collective operations. However, due to its flexible and portable design, ACCL+ can be utilized in various applications and scenarios beyond the demonstrated use cases. This section explores how ACCL+ can be extended for a broader range of users applications.

Integrating ACCL+ with Machine Learning Frameworks. While in HPC it is commonplace to develop distributed applications utilizing MPI collectives explicitly, in the field of Machine Learning, codes are often written by Data Scientists who reason about distributed execution in high-level terms such as data, model, or expert parallelism [20]. Integrating ACCL+ into popular machine learning frameworks like TensorFlow [1] and PyTorch [78] is therefore essential to enable its use in ML. Our ongoing work focuses on integrating ACCL+ into PyTorch's Distributed Data Parallel (DDP) [82] module. DDP supports various communication backends for collective operations, which are invoked automatically by the PyTorch execution orchestrator to distribute work to a cluster. We aim to add ACCL+ as a new communication backend to PyTorch DDP, enabling the use of FPGA-based smartNICs to enhance collective operations in AI training and inference.

Additionally, we plan to extend ACCL+ support to other machine learning frameworks.

ACCL+ for Streaming Applications. ACCL+ can also be used for distributed applications that do not require bulk synchronous parallel collective communications, such as streaming applications. In such a scenario, one could use ACCL+ as a transport layer for model-parallel, multi-FPGA streaming accelerators, e.g., Elastic-DF [3]. ACCL+ has existing streaming primitives and collectives which could be utilized for this purpose, as demonstrated in the implementation of the DLRM in Section 6. A more flexible transport based on ACCL+ would, for example, enable higher flexibility in partitioning DNNs to multiple FPGAs.

Implementing Other Distributed Programming Models with ACCL+. The shared memory (SHMEM) programming model [17] is gaining in popularity as it becomes evident that it enables finer-grained overlap between compute and communication on GPU-accelerated systems [50]. SHMEM libraries include MPI-like collectives but add asynchronous one-sided operations (put/get) and signals. These additional operations could be implemented easily into ACCL+ with minimal firmware modifications and no hardware recompilation. Utilizing ACCL+ could reduce the latency of one-sided SHMEM operations, especially where these are used to implement complex communication sequences such as halo exchanges in stencil computations.

8 Conclusion

In this paper, we introduce ACCL+, an open-source FPGA-based collective library designed for portability across diverse platforms and communication protocols. ACCL+ offers flexibility in implementing collectives without the need for FPGA re-synthesis and demonstrates high performance as collective abstractions for FPGA-distributed applications and as a collective offload engine for CPU applications. With ACCL+, there is potential for exploring new possibilities by extending collectives across CPU and FPGA boundaries and orchestrating them for a unified application.

Acknowledgements

We would like to thank AMD for the donation of the HACC FPGA cluster at ETH Zurich on which the experiments were conducted. We thank our shepherd Ming Liu and the anonymous reviewers for their helpful feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2023.

[3] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021.

[4] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the Network for Scatter-Gather Workloads. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2020.

[5] AMD. RCCL's documentation. https://rccl.readthedocs.io/en/rocm-4.3.0/, 2021.

[6] Omer Arap, Lucas R.B. Brasilino, Ezra Kissel, Alexander Shroyer, and Martin Swany. Offloading Collective Operations to Programmable Logic. *Journal of IEEE Micro*, 2017.

[7] Omer Arap, Geoffrey M. Brown, Bryce Himebaugh, and D. Martin Swany. Software Defined Multicasting for MPI Collective Operation Offloading with the NetFPGA. In *European Conference on Parallel Processing*, 2014.

[8] Omer Arap and Martin Swany. Offloading Collective Operations to Programmable Logic on a Zynq Cluster. In *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, 2016.

[9] ARM. AMBA 4 AXI4-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/a/, 2010.

[10] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *Journal of IEEE Access*, 2017.

[11] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek,

editors, *High Performance Computing*, Cham, 2021. Springer International Publishing.

[12] Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. Toward Multi-FPGA Acceleration of the Neural Networks. *ACM Journal on Emerging Technologies in Computing Systems*, 2021.

[13] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

[14] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4T: A Fast and Flexible FPGA-Based Full-Stack TCP Acceleration Framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[15] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[16] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[17] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010.

[18] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF)*. Association for Computing Machinery, 2014.

[19] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *Journal of IEEE Micro*, 2018.

[20] Colossal.AI. Paradigms of Parallelism. https://colossalai.org/docs/concepts/paradigms_of_parallelism.

[21] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[22] Nicholas Contini, Bharath Ramesh, Kaushik Kandadi Suresh, Tu Tran, Ben Michalowicz, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar Panda. Enabling Reconfigurable HPC through MPI-Based Inter-FPGA Communication. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*. Association for Computing Machinery, 2023.

[23] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016.

[24] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. An FPGA-Based Network Intrusion Detection Architecture. *IEEE Transactions on Information Forensics and Security*, 2008.

[25] Juan Miguel de Haro, Rubén Cano, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Francois Abel, Burkhard Ringlein, and Beat Weiss. OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[26] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. Streaming Message

Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2019.

[27] Li Ding, Ping Kang, Wenbo Yin, and Linli Wang. Hardware TCP Offload Engine based on 10-Gbps Ethernet for low-latency network communication. In *2016 International Conference on Field-Programmable Technology (FPT)*, 2016.

[28] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. FlexDriver: A Network Driver for Your Accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[29] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, 2019. USENIX Association.

[30] Nariman Eskandari, Naif Tarafdar, Daniel Ly-Ma, and Paul Chow. A Modular Heterogeneous Stack for Deploying FPGAs and CPUs in the Data Center. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2019.

[31] Yuanwei Fang, Chen Zou, and Andrew A. Chien. Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture. *Proceedings of the Very Large Data Base Endowment (VLDB)*, 2019.

[32] Clément Farabet, Cyril Poulet, and Yann LeCun. An FPGA-based stream processor for embedded real-time vision with Convolutional Networks. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, 2009.

[33] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The LEAP FPGA operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[34] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging Latency-Insensitivity to Ease Multiple FPGA Design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2012.

[35] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An Open-Source 100-Gbps Nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[36] Shanyuan Gao, Andrew G. Schmidt, and Ron Sass. Hardware implementation of MPI_Barrier on an FPGA cluster. In *2009 International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[37] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 2015.

[38] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*. Springer-Verlag, 2005.

[39] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[40] Anqi Guo, Tong Geng, Yongan Zhang, Pouya Haghi, Chunshu Wu, Cheng Tan, Yingyan Lin, Ang Li, and Martin Herbordt. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[41] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[42] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[43] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[44] Pouya Haghi, Anqi Guo, Tong Geng, Justin Broaddus, Derek Schafer, Anthony Skjellum, and Martin Herbordt. A Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[45] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[46] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. ACCL: FPGA-Accelerated Collectives over 100 Gbps TCP-IP. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2021.

[47] Fernando Luis Herrmann, Guilherme Perin, Josue Paulo Jose de Freitas, Rafael Bertagnolli, and Joao Baptista dos Santos Martins. A Gigabit UDP/IP network stack in FPGA. In *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, 2009.

[48] Pieter Hintjens. *ZeroMQ: messaging for many applications*. "O'Reilly Media, Inc.", 2013.

[49] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-Stack Workload Characterization of Deep Recommendation Systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020.

[50] Chung-Hsing Hsu and Neena Imam. Assessment of nvshmem for high performance computing. *International Journal of Networking and Computing*, 2021.

[51] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[52] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[53] Intel. Intel FPGA Add-on for oneAPI Base Toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html.

[54] Intel. Intel Quartus Prime Standard Edition User Guide: Getting Started. https://www.intel.com/content/www/us/en/programmable/documentation/yoq1529444104707.html.

[55] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[56] Yong Ji and Qing-Sheng Hu. 40Gbps multi-connection TCP/IP offload engine. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, 2011.

[57] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions. In *2021 4th Conference on Machine Learning and Systems (MLSys)*, 2021.

[58] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. Association for Computing Machinery, 2021.

[59] Vinod Kathail. Xilinx Vitis Unified Software Platform. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[60] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[61] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. In *12th Annual Conference on Innovative Data Systems Research Proceedings (CIDR)*, 2022.

[62] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[63] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[64] Joshua Lant, Emmanouil Skordalakis, Kyriakos Paraskevas, William B. Toms, Mikel Luján, and John Goodacre. DiAD – Distributed Acceleration for Datacenter FPGAs. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[65] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[66] Junnan Li, Zhigang Sun, Jinli Yan, Xiangrui Yang, Yue Jiang, and Wei Quan. DrawerPipe: A Reconfigurable Pipeline for Network Processing on FPGA-Based SmartNIC. *Electronics*, 2020.

[67] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[68] Junyi Liu, Aleksandar Dragojević, Shane Fleming, Antonios Katsarakis, Dario Korolija, Igor Zablotchi, Ho-Cheung Ng, Anuj Kalia, and Miguel Castro. Honeycomb: ordered key-value store acceleration on an FPGA-based SmartNIC. *IEEE Transactions on Computers (TC)*, 2023.

[69] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[70] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[71] Chinmay Mahajan, Ashwin Krishnan, Manoj Nambiar, and Rekha Singhal. Hetero-Rec: Optimal Deployment of Embeddings for High-Speed Recommendations. In *Proceedings of the Second International Conference on AI-ML Systems (AIMLSystems)*, New York, NY, USA, 2023. Association for Computing Machinery.

[72] Joel Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[73] Kenji Mizutani, Hiroshi Yamaguchi, Yutaka Urino, and Michihiro Koibuchi. OPTWEB: A Lightweight Fully Connected Inter-FPGA Network for Efficient Collectives. *IEEE Transactions on Computers (TC)*, 2021.

[74] NVDIA. NVIDIA Collective Communications Library (NCCL). https://docs.nvidia.com/deeplearning/nccl/index.html, 2021.

[75] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011.

[76] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science*, 2021.

[77] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out Acceleration for Machine Learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, 2017.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

[79] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives. *arXiv preprint arXiv:2401.16677*, 2024.

[80] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, and Dhabaleswar K Panda. Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012.

[81] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, 2014.

[82] PyTorch. DistributedDataParallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html.

[83] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. A Survey of System Architectures and Techniques for FPGA Virtualization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2021.

[84] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. ZRLMPI: A Unified Programming Model for Reconfigurable Heterogeneous Computing Clusters. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[85] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020.

[86] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. A Case for Function-as-a-Service with Disaggregated FPGAs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[87] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[88] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. MPI as a Programming Model for High-Performance Reconfigurable Computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2010.

[89] Manuel Saldana and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *2006 International Conference on Field Programmable Logic and Applications (FPL)*, 2006.

[90] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. Parallel Matrix Multiplication: A Systematic Journey. *SIAM Journal on Scientific Computing*, 2016.

[91] Niklas Schelten, Fritjof Steinert, Anton Schulte, and Benno Stabernack. A High-Throughput, Resource-Efficient Implementation of the RoCEv2 Remote DMA Protocol for Network-Attached Hardware Accelerators. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[92] Junnan Shan, Mihai T. Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R. Casu. CNN-on-AWS: Efficient Allocation of Multikernel Applications on Multi-FPGA Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[93] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a Fully Disaggregated and Programmable Data Center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. Association for Computing Machinery, 2022.

[94] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency TCP/IP stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[95] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2020.

[96] Kaushik Kandadi Suresh, Benjamin Michalowicz, Bharath Ramesh, Nick Contini, Jinghan Yao, Shulei Xu, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[97] Naif Tarafdar, Nariman Eskandari, Varun Sharma, Charles Lo, and Paul Chow. Galapagos: A Full Stack

Approach to FPGA Integration in the Cloud. *Journal of IEEE Micro*, 2018.

[98] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[99] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A Survey on FPGA Virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[100] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[101] John Paul Walters, Xiandong Meng, Vipin Chaudhary, Tim Oliver, Leow Yuan Yeow, Bertil Schmidt, Darran Nathan, and Joseph Landman. MPI-HMMER-boost: distributed FPGA acceleration. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 2007.

[102] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2014.

[103] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, 2022.

[104] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015.

[105] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, 2016.

[106] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, 2016.

[107] AMD Xilinx. XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example.

[108] AMD Xilinx. Quick Start Guide: MicroBlaze Soft Processor for Vitis 2019.2. https://www.xilinx.com/support/documentation/quick_start/microblaze-quick-start-guide-with-vitis.pdf, 2019.

[109] AMD Xilinx. AXI DataMover v5.1 LogiCORE IP Product Guide. https://docs.xilinx.com/r/en-US/pg022_axi_datamover/AXI-DataMover-v5.1-LogiCORE-IP-Product-Guide, 2023.

[110] AMD Xilinx. DMA for PCI Express (PCIe) Subsystem. https://www.xilinx.com/products/intellectual-property/pcie-dma.html, 2023.

[111] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 2011.

[112] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020.

[113] Yue Zha and Jing Li. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[114] Yue Zha and Jing Li. When Application-Specific ISA Meets FPGAs: A Multi-Layer Virtualization Framework for Heterogeneous Cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2021.

[115] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, and Zeke Wang. SmartDS: Middle-Tier-Centric SmartNIC Enabling Application-Aware Message Split for Disaggregated Block Storage. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[116] Wentai Zhang, Jiaxi Zhang, Minghua Shen, Guojie Luo, and Nong Xiao. An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[117] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[118] Yu Zhu, Zhenhao He, Wenqi Jiang, Kai Zeng, Jingren Zhou, and Gustavo Alonso. Distributed Recommendation Inference on FPGA Clusters. In *2021 31th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021.

[119] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *Journal of IEEE Micro*, 2014.

Appendices

A ACCL+ Initialization

ACCL+ is specifically designed to deliver a high-speed collective communication solution tailored for FPGAs, or to function as a specialized NIC for CPUs. To simplify the initialization process, we choose not to generalize the network stack in the hardware for general-purpose communication. Instead, we utilize the conventional NIC in the CPU system for launching ACCL+ applications in a distributed environment, such as through *mpirun*, or for establishing RDMA queue pairs for ACCL+ communicators. This NIC only involves a lower-speed connection to other ranks.

Other than the *collective API*, the CCL driver also exposes a *housekeeping API* which enables CCLO configuration and monitoring, and a *primitive API* consisting of simple data movement operations (send, receive, copy). Listing 3 illustrates the three APIs - the code initializes ACCL+, invokes the ACCL+ send/receive primitives to exchange data between ranks 0 and 1, and executes an reduce collective on all ranks.

In this example, we utilize the MPI library to determine the local rank ID (lines 6-8) when the application has been launched with mpirun. Then ACCL+ is initialized by calling the constructor function and passing the Coyote device object (line 11). Similar approach is applied for Vitis device object. Within the constructor, it also allocates and configures a set of CCLO-managed Rx buffers for collective operations in the FPGA memory, e.g., for the eager protocol. The code then constructs the communicator according to rank information and protocol type (line 15). If the protocol is TCP, the code will issue commands to open connections between each rank in the communicator via the protocol offload engine. If the protocol is RDMA, the code utilizes the commodity NIC to change queue pair information. The TCP connections and the RDMA queue pairs are generalized to session IDs in the communicator. All configuration information is offloaded to the FPGA so that the CCLO can rapidly access them. Just like MPI, ACCL+ can be configured with multiple communicators of different sizes. While not pictured here for brevity, each ACCL+ collective can specify the communicator it operates on, with COMM_WORLD being the default.

Lines 21-25 implement data movement from the buffer of rank 1 to the buffer of rank 0 utilizing the primitives API. Lines 27 execute collectives on the entire communicator using the collectives API.

```
1   #include "accl.hpp"
2   #include <mpi.h>
3   using namespace ACCL;

5   int main(int argc, char **argv) {
6       int mpi_rank;
7       MPI_Init(&argc, &argv);
8       MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

10      CoyoteDevice* device = new CoyoteDevice();
11      ACCL* accl = new ACCL(device);

13      std::map<int, std::string> ranks_dict = /*
            Populate rank vector*/;
14      Protocol protocol = TCP; // or RDMA
15      accl->configure_communicator(ranks_dict, mpi_rank,
            protocol);

17      const int bufsize = 64;
18      auto opbuf = accl->create_buffer<int>(bufsize);
19      auto resbuf = accl->create_buffer<int>(bufsize);

21      if (mpi_rank == 0) {
22          accl->send(opbuf, bufsize, 1); // Send to rank 1
23      } else if (mpi_rank == 1) {
24          accl->receive(opbuf, bufsize, 0); // Receive
                from rank 0
25      }

27      accl->reduce(opbuf, resbuf, bufsize, 0); // Root
            rank 0

29      opbuf->free_buffer();
30      resbuf->free_buffer();
31      delete accl;
32      delete device;
33      MPI_Finalize();
34  }
```

Listing 3: Initialization and invocation of collectives from CPU.

Beaver: Practical Partial Snapshots for Distributed Cloud Services

Liangcheng Yu[♡] Xiao Zhang[♠] Haoran Zhang[♡] John Sonchack[♣] Dan Ports[♢] Vincent Liu[♡]
[♡]*University of Pennsylvania* [♠]*Shanghai Jiao Tong University* [♣]*Princeton University* [♢]*Microsoft / University of Washington*

Abstract

Distributed snapshots are a classic class of protocols used for capturing a causally consistent view of states across machines. Although effective, existing protocols presume an isolated universe of processes to snapshot and require instrumentation and coordination of all. This assumption does not match today's cloud services—it is not always practical to instrument all involved processes nor realistic to assume zero interaction of the machines of interest with the external world.

To bridge this gap, this paper presents Beaver, the first practical partial snapshot protocol that ensures causal consistency under external traffic interference. Beaver presents a unique design point that tightly couples its protocol with the regularities of the underlying data center environment. By exploiting the placement of software load balancers in public clouds and their associated communication pattern, Beaver not only requires minimal changes to today's data center operations but also eliminates any form of blocking to existing communication, thus incurring near-zero overhead to user traffic. We demonstrate the Beaver's effectiveness through extensive testbed experiments and novel use cases.

1 Introduction

The ability to capture a consistent, global view of a system is a powerful tool. For many tasks—deadlock detection, checkpoints and failure recovery, network telemetry, debugging of distributed software, and many others [3–5, 8, 9, 17, 33, 37, 39, 52, 54, 55, 58, 60]—a global view, and particularly a consistent one, is essential for correct operation. Without consistency, results are unreliable, and the value of associated tools is questionable.

The classic method for capturing consistent global states is the Chandy-Lamport snapshot algorithm that was proposed almost four decades ago and its subsequent variants [11, 26, 33–35, 41, 57, 60]. At a high level, these protocols flood snapshot initiation messages throughout the system, triggering local captures of state at every node they pass in a manner that guarantees causal consistency of the recorded values. Some versions (including the original) also include support of capturing messages that are in-flight at the time of the snapshot, i.e., channel state.

While these protocols have been simple, effective, and widely used for decades, they all rely on the fundamental assumption that the set of participants in the protocol is closed under causal propagation. In other words, if any node can both send and receive messages from participants in the protocol, it can propagate Lamport's 'happened-before' relation [35] and must also be a participant in the snapshot. For systems operating in isolation, ensuring full participation is trivial; however, modern cloud deployments are not so utopian.

Today's cloud services are often modular, e.g., structured as microservices, each of which might be developed and maintained by a different user, team, or organization or hosted on otherwise inaccessible infrastructure. Take, for instance, a managed pub/sub messaging layer like Amazon's Simple Notification Service (SNS). As a proprietary and black-box service, users cannot directly propagate snapshot initiation markers through the service. Further, while they might be able to add markers to the application-level content manually, with concurrency, replication, and reordering (e.g., due to prioritization), content-based markers are unlikely to track causal relationships accurately. Even when developers fully control all relevant servers, the clients of the service can also introduce hidden causal relationships, for example, when the user of a generative AI chatbot sends a follow-up message based on the response to the previous prompt. Ultimately, the nature of causal consistency means that a single non-participant can render all snapshots useless.

Observing this gap between classical assumptions and the practicalities of real-world deployments, we ask the question: Can we make distributed snapshots practical in modern cloud data centers, i.e., is it possible to capture a causally consistent snapshot when only a subset of the broader system participates? At first glance, this goal seems far-fetched: With partial participation, we cannot control the messaging behaviors nor instrument any coordination logic for machines external to those of interest. Complicating the issue is the fact that, to be practical, the protocol cannot block, e.g., by buffering or delaying user packets during a snapshot. In essence, this means that hidden causal relationships between participants and external communication partners are unavoidable.

This work presents Beaver[1], the first 'partial' snapshot protocol that extends the capability of distributed snapshots to cloud services with external interactions. Beaver provides the same basic abstraction as other snapshot protocols—for any event whose effects are observed in the snapshot, all other

[1] The animal species known for their engineering expertise in constructing dams using locally available materials such as rocks and tree branches.

events that ‘happened-before’ are also included. It achieves this even when the target service communicates with an arbitrary number of external, black-box entities, regardless of their scale, semantics, or placement, and despite potential multi-hop propagation of causal dependencies. Beaver does all of this without blocking or delaying user requests. Beaver tackles this seemingly impossible problem by:

1. Relying on two features found in all of today’s largest cloud data centers: (a) Layer-4 Software Load Balancers (SLBs) that interpose on a subset of inbound traffic [15, 28, 49, 50] and (b) servers with low time strata or otherwise stable clocks [13, 25, 29, 36, 38, 42, 44, 45].
2. Eschewing the enforcement of causal consistency in favor of simply *detecting* when violations may have occurred, a mechanism we call Optimistic Gateway Marking (OGM).

Note that for (1b), Beaver does not rely on the traditional notion of clock synchronization that other recent systems [13, 38, 42] are founded upon, which requires that the clocks of distinct machines have bounded drift. Instead, it uses a much weaker property [25, 40] over the frequency drift of a single machine[2]. Also note that (2) implies a tradeoff: snapshots are not always successful, but users can be assured of their correctness when they are and retry when they are not.

At a high level, Beaver’s approach is based on the observation that when examining the causal consistency of a snapshot, only inbound traffic is relevant and only a small subset therein. More specifically, we can divide inbound traffic into messages that are ‘causally irrelevant’ (e.g., triggered asynchronously and, thus, are not a part of any transitive causal relationships) and messages that are ‘causally relevant’ (e.g., triggered by post-snapshot outbound traffic but may not carry any markers of that fact). Beaver’s OGM mechanism is an approximate but full-recall detector of causally relevant traffic.

Our prototype[3] of Beaver demonstrates that not only is it possible to build an OGM mechanism, but by leveraging the aforementioned features of today’s cloud data centers, we can render the possibility of rejected snapshots minimal (near-zero in many cases). To summarize, this paper makes the following contributions:

- To the best of our knowledge, we are the first to detail the gap between classical assumptions of distributed snapshots and the practicalities of real-world clouds.
- We propose Beaver, the first partial distributed snapshot primitive for modern cloud services. Beaver presents a unique design point by tightly coupling the protocol with the regularities of the underlying data center environment.
- We evaluate Beaver through end-to-end implementation on a real-world testbed aligned with the production data center settings. We also show that the causally consistent view provided by Beaver enables a spectrum of use cases.

[2] Bounded clock drift to a low-stratum reference server is sufficient to guarantee bounded local frequency drift, but not necessary.

[3] The prototype is available at https://github.com/eniac/Beaver.

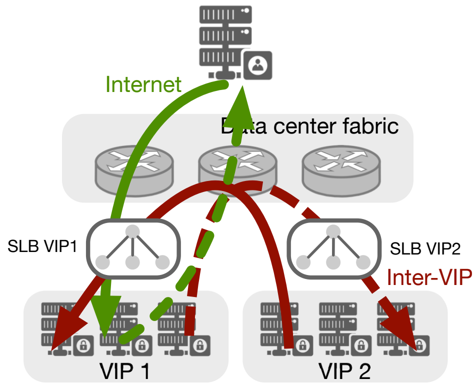


Figure 1: Today’s public cloud services place SLBs to handle the external traffic to its VIP in the inbound direction (solid lines to VIP 1). The response to inbound messages (dotted lines from VIP 1) typically bypasses its SLB to minimize the SLB traffic load.

2 Background and Motivation

We begin by describing the structure of today’s cloud services and the data centers in which they reside before we discuss the application of distributed snapshots to these services.

2.1 Communication in Public Cloud Data Centers

Today’s cloud data centers are massive collections of servers connected by a network fabric that host user services of diverse sizes and scopes. In this context, we can abstract user services as a set of virtual or bare metal machines managed as a single logical entity. Each service is typically assigned a public Virtual IP (VIP) address, and each physical machine a private Direct IP (DIP) address [15, 50].

Software Load Balancers (SLBs). A set of dedicated servers or programmable devices is responsible for translating between VIPs and DIPs. We refer interested readers to prior work [15, 50] for full details, but at a high level, these layer-4 devices act similarly to traditional Network Address Translators (NATs), allocating a new mapping for every new connection and rewriting the headers of every passing packet according to the mapping. In cloud systems, these devices are distributed, replicated, and serve an additional purpose as software load balancers that spread requests over available backend servers. A single service/VIP typically has a dedicated set of SLBs based on its scale (e.g., ∼7–20, including replication).

The path of packets in public clouds. In the presence of SLBs, packets can take different paths depending on the relationship between their source and destination (Figure 1):

Internet traffic: Incoming packets from the Internet are always routed through an SLB to translate from the service’s publicly visible VIP to a relevant internal DIP [15, 21, 22, 43, 50, 63]. Unlike most other NAT-like mechanisms, response packets are usually sent back directly, bypassing the SLB using techniques like Direct Server Return (DSR) [15, 50].

Inter-service traffic: Inbound traffic from other services within the same provider also passes through SLBs [15, 21, 22, 43, 50, 63], which still need to perform the same VIP-to-DIP translation. This is true even if the two service's servers are physically adjacent. Note that, like with Internet traffic, outbound traffic can bypass the responder's SLB; however, even in this case, the packet will still need to pass through the SLB responsible for the destination VIP(s), as shown in Figure 1. Note that while cached DIPs have been suggested to bypass inbound SLBs on the fast path [50], this optimization is currently disabled for major classes of production traffic due to load imbalance and cache management issues. The implication is that, at least for public clouds, this need to interpose on all inbound traffic is ubiquitous [10, 15, 63].

Intra-service traffic: Finally, messages between sources and destinations belonging to the same VIP are sent directly, bypassing the SLBs entirely.

Typical service communication patterns. In parallel to the above, we note that modern cloud services rarely operate in isolation. Frontend services typically rely on a wide array of backend services, e.g., to handle storage, analytics, and learning, thus triggering inter-service traffic. The rise of managed cloud service offerings and microservice design patterns have further encouraged modularity and the associated growth in the number of distinct services involved in processing a single user request. At a more basic level, most cloud services take requests from and return responses to external clients, each with its own internal, causality-carrying logic.

2.2 Revisiting the Chandy-Lamport Snapshot

The ability to capture a consistent snapshot of a cloud service's global state is a powerful tool. Indeed, many problems in distributed systems boil down to determining the global state across machines, including distributed logging and debugging, network telemetry, checkpointing and recovery, and deadlock detection [3, 11, 17, 33, 56, 60].

Intuitively, a snapshot is a collection of local states captured from the processes of a system. For simplicity, we omit channel states in our definitions, but the analysis is similar. The snapshot is deemed consistent if the captured states at each process 'cut' the timeline of events in a way that respects the following definition:

Definition 1. *(Consistent Snapshot [11, 56]). For a snapshot, let C be the set of events on every process that occurs before the 'cut'. C is causally consistent iff $\forall e \in C$, if $e' \rightarrow e$, then $e' \in C$, where $x \rightarrow y$ denotes that x 'happened before' y.*

The seminal Chandy-Lamport algorithm was the first to present a solution for this problem. We refer the interested readers to the original paper [11] or a distributed systems textbook [33, 56] for complete details, but we give a simplified description of the model and the protocol below:

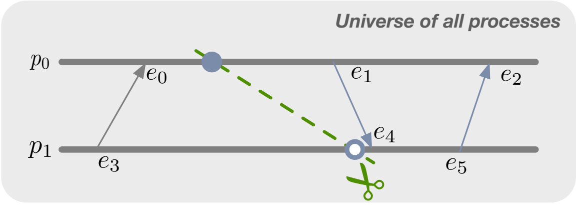


Figure 2: A minimal example of a consistent cut for 2 processes p_0, p_1 and 6 events $e_{0,1,...,5}$. The global snapshot formed from the collection of ○ and ● is a 'causal cut' of the event timelines for all processes, where ● and ○ indicate snapshot initiations triggered out-of-band or by receiving marker messages, respectively.

- *Model:* A system involves a set of asynchronous processes $P = \{p_0, p_1, \ldots, p_{N-1}\}$ that interconnect with each other through FIFO message channels. Each process p_i holds state of interest, s_i, that may change in response to local events (e.g., local computation, message sends or receives, etc.). A global snapshot involves a union of states $\{s_i\}$ recorded at different times for all processes.
- *Protocol state machine:* The protocol requires coordination in *all* processes $p \in P$. An initiator process first records its local state and then sends a marker message to all others. The captured state is application-dependent and can range from a single bit representing the state of a lock to all of local memory. When any other process p_i receives a marker message for the first time, it records its state s_i and, to ensure consistency, sends marker messages immediately through all other channels.

Later variants refine the basic algorithm to generalize channel assumptions, allow for concurrent initiation, or reduce message complexity [26, 33, 34, 41, 57, 60]. In particular, the Lai-Yang algorithm permits non-FIFO and lossy channels by having processes piggyback a single marker bit in every sending message [34] rather than sending separate marker messages as in the original protocol. Upon receiving a message with a marker bit set, the receiving process *first* records the local state, processes the payload, and sets the bit for future sending messages. Additional bits can be used to support concurrent snapshots. Figure 2 shows a consistent cut with the Lai-Yang algorithm.

2.3 A Case for Partial Snapshots

The above snapshot algorithm makes a fundamental and implicit assumption that *all* processes that can communicate with processes in P are themselves in P. Unfortunately, as previously mentioned, today's cloud services are frequently interconnected, with efforts toward modular design and managed solutions promoting increasing complexity in the dependency graph over time. As a rough indication of severity,

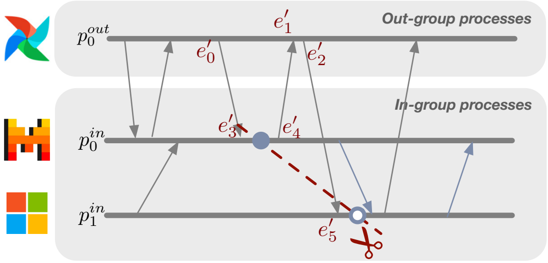


Figure 3: An application where a distributed serving system is accessed by an external user (e.g., an Apache Airflow workflow). The out-group process p_0^{out} imposes a hidden causal relationship $e'_4 \rightarrow e'_5$ between events e'_4 and e'_5, rendering a traditional snapshot of only the serving system inconsistent.

previous studies have shown that inter-service traffic comprises 10–50% of total traffic in the data center, and Internet traffic accounts for 5–25% [21, 22, 50].

Consider, for instance, a HuggingFace-like ML inference service [2] that hosts a collection of models that can be accessed from external clients. As they are externally visible, the models are frequently used in larger jobs, e.g., as part of an interactive chatbot (where clients submit requests based on prior responses) or more complex Apache Airflow workflows.

The inference service might want to capture a service-wide statistic (e.g., tracking the maximum number of in-flight requests) to decide on the number of servers to provision. Any analysis of the developer's application that does not consider the potential dependencies introduced by external services or clients will miss important causal dependencies.

Figure 3 shows a simple example of this, where a single external Airflow job makes requests to multiple models hosted by the inference service such that only one request is outstanding at any given time. Occasional internal messages are for monitoring and coordination. Although there is at most one outstanding request at any given time, a traditional distributed snapshot that only considers the inference service will not respect that bound.

For example, in Figure 3, the depicted cut 'observes' two inflight messages because it fails to capture the external interactions ($e'_5 \in C$, yet $e'_4 \notin C$). In fact, for a single client that issues a single request at a time, an n-server snapshot can 'observe' any number of in-flight requests [0, n]. These arbitrary results can cause the developer to waste money and resources on redundant provisioning. More broadly, while the frequency and consequences of consistency violations are application-dependent, there is often a meaningful difference between 'correct' and 'incorrect'.

Although converting all cloud services into participants of the snapshot protocol might be possible given either (a) a well-resourced developer who can implement and manage everything (even if machines are geo-distributed or on the broader Internet) in-house or (b) support from the cloud provider to propagate snapshot markers on all packets, these approaches are not always feasible. For (a), the popularity of managed services demonstrates their importance to low-cost and agile development. For (b), forced instrumentation can lead to overhead and fragmentation for users not involved in the snapshot. Even worse, if the external source of dependencies is a human (e.g., accessing your service through a browser), incorporating her into the snapshot is impractical.

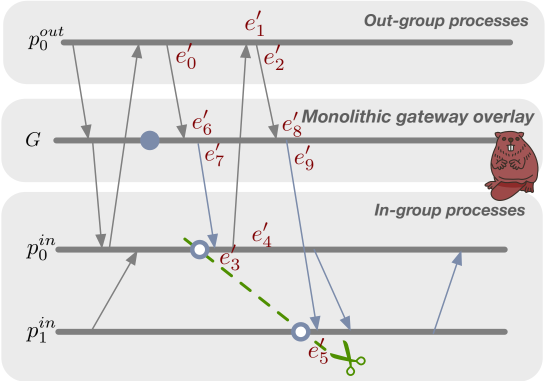


Figure 4: With the gateway indirection, Beaver's MGM results in a new frontier at the in-group process p_1^{in} that precedes rather than succeeds the event e'_5 (as in the scenario of Figure 3), converging to a consistent partial snapshot.

A formal definition of partial snapshots. We seek the design and implementation of a partial snapshot. In a partial snapshot, processes are divided into two groups. The first, *in-group processes* P^{in}, are the machines of the VIP(s) of interest. The second, *out-group processes* P^{out}, includes all other machines, whether in the same data center or the broader Internet.

Given these sets, we refine Definition 1 to obtain a definition of consistent partial snapshots:

Definition 2. *(Consistent Partial Snapshot). Consider a universe of processes* $P = P^{in} \cup P^{out}$, $P^{in} \cap P^{out} = \emptyset$. *Let* C_{part} *be the set of pre-snapshot events for* P^{in}. C_{part} *is causally consistent iff* $\forall e \in C_{part}$, *if* $e'.p \in P^{in} \wedge e' \rightarrow e$, *then* $e' \in C_{part}$.

Similar to traditional snapshots, for a set of in-group processes P^{in}, if a consistent partial snapshot includes the effect of an event e, it must include any event e' at $p \in P^{in}$ that leads to it. Like traditional snapshots, the 'happened before' relation, → is transitive and defined over events in the universe of processes. Unlike traditional snapshots, however, the included events only account for in-group events.

3 Gateway Marking

This paper introduces Beaver, a partial snapshot primitive that captures a causally consistent collection of state for cloud services sitting behind one or more operator-specified VIPs.

Fundamentally, the nodes in P^{out} are uncontrollable and, as a result, can introduce arbitrary hidden causal relationships, disrupting the consistency of traditional snapshots. At

Symbol	Description
P	Set of all processes.
P^{in}	Set of in-group processes with states of interest.
P^{out}	Set of out-group processes without any control.
G	Set of gateways handling inbound traffic for P^{in}.
C	Set of pre-snapshot events for a snapshot 'cut'.
e	Event tuple $e = (p, m, t)$.
$e.p$	The process at which an event e occurs.
$e.m$	The message involved in an event e, if any.
$e.t$	Global wall clock time, for ease of discussion.
e^{ss}_{gmax}	The event when the last gateway is in a new snapshot.
e^{ss}_{gmin}	The event when the first gateway is in a new snapshot.
e^{ss}_{g}	The event when $g \in G$ enters a new snapshot.
e^{ss}_{p}	The event triggering $p \in P^{in}$ to enter a new snapshot.
$d(p,q;V)$	One way delay from p to q with intermediate nodes $v \in V$ $(p, q \in (P \cup G), V \subseteq (P \cup G))$ in sequence.
τ_{min}	Min time for an external causal chain to occur.

Table 1: Summary of notations in Beaver.

the core of Beaver is a primitive called Optimistic Gateway Marking (OGM), which allows Beaver to detect when such causality violations may have occurred. As we show later in §5, by combining this primitive with common-case features of today's cloud data centers, Beaver can provide:

- Partial deployability where *only* the in-group machines for the target VIP(s) participate while ensuring high-rate, consistent partial snapshots for the target service(s).
- Minimal cost for data center infrastructure, for example, without switch reconfiguration or additional SLB replicas.
- Near-zero impact on existing data center service traffic.

In this section, we first introduce a strawman version of the primitive before discussing practicalities and how Beaver addresses them with OGM in §4.

Strawman: Monolithic Gateway Marking (MGM). Beaver starts with a simple idea: for all packets originating from out-group nodes and destined for in-group nodes, route them through a gateway. The gateway is responsible for two tasks:

1. Tagging incoming packets to in-group nodes with snapshot markers.
2. Initiating snapshots by tagging all subsequent inbound messages accordingly.

After the gateway initiates a snapshot, the protocol proceeds as a traditional snapshot among the in-group nodes. For the strawman, assume that the gateway is implemented by a single monolithic node. Figure 4 shows an example execution using the above protocol and the same application-level communication pattern as Figure 3. In contrast to Figure 3, indirection and marking via a gateway cause p^{in}_1 to take the snapshot at the correct time. In a way, the gateway node in this protocol can be seen as a stand-in for all nodes in P^{out}. We can prove that MGM produces a consistent partial snapshot.

THEOREM 1. *With MGM, a partial snapshot C_{part} for $P^{in} \subseteq P$ is causally consistent, that is, $\forall e \in C_{part}$, if $e'.p \in P^{in} \wedge e' \to e$, then $e' \in C_{part}$.*

PROOF. *Let $e.p = p^{in}_i$ and $e'.p = p^{in}_j$. There are 3 cases:*

1. *Both events occur in the same process, i.e., $i = j$.*
2. *$i \neq j$ and the causality relationship $e' \to e$ is imposed purely by in-group messages.*
3. *Otherwise, the causality relationship $e' \to e$ involves* at least *one* $p \in P^{out}$.

In cases (1) and (2), the theorem is trivially true using identical logic to proofs of traditional distributed snapshot protocols. We prove (3) by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. With (3), $e' \to e$ means that there must exist some e^{out} (at an out-group process) satisfying $e' \to e^{out} \to e$. Now, because $e' \notin C_{part}$, we know $e^{ss}_{p^{in}_j} \to e'$ or $e^{ss}_{p^{in}_j} = e'$, that is, p^{in}_j's local snapshot happened before or during e'. Combined with the fact that the gateway is the original initiator of the snapshot protocol, we know that $e^{ss}_g \to e' \to e^{out} \to e$.

We can focus on a subset of the above causality chain: $e^{ss}_g \to e$. From the properties of the in-group snapshot protocol, $e^{ss}_g \to e$ implies $e \notin C_{part}$.

This contradicts our original assumption that $e \in C_{part}$! □

Theorem 1 implications: Beyond correctness, the strawman exhibits several valuable properties:

1. *Obliviousness to out-group semantics:* The proof treats the internals of the out-group processes as a black box. In fact, the protocol remains correct, even if the causal dependency results from multiple network hops through distinct out-group nodes or if an element of the out-group chain is a human.
2. *Obliviousness to outbound messages:* The gateway only needs to observe messages inbound to in-group processes without requiring any visibility or tagging of outbound messages. MGMs achieve this by initiating the snapshot at the gateway, which—as a stand-in for P^{out}—obviates the need to track dependencies carried to the out-group.

SLBs as a candidate for gateway marking. The SLBs described in §2.1 are a convenient candidate for implementing gateway marking as VIPs are a natural granularity for service-specific partial snapshots, and SLBs already interpose on all incoming traffic to a VIP—regardless of whether it is from the Internet or a different service. MGM's obliviousness to outbound messages helps here as well, making the system amenable to DSR.

Of course, assuming that a single server can handle all incoming traffic to a service is not feasible. The scale of modern SLBs serves as proof that even for simple gateway processing incoming requests for a single service, multiple servers are necessary to handle typical data volumes, load balance among SLBs, and provide fault tolerance.

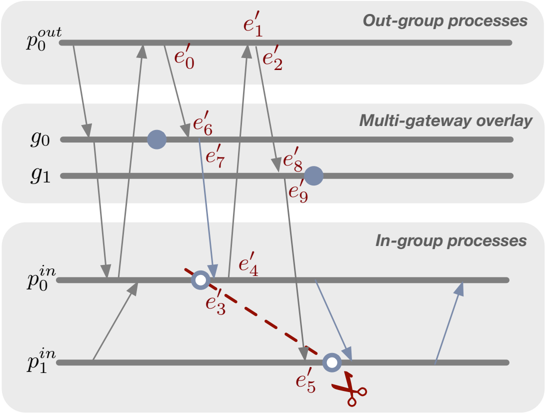


Figure 5: An inconsistent partial snapshot using two asynchronous SLBs g_0, g_1. When $e'_8.m$ arrives at g_1, g_1 has not initiated the new snapshot mode to mark the message, thus triggering the violation.

4 Optimistic Gateway Marking (OGM)

Beaver extends gateway marking to practical, distributed environments using OGM. First, to see why asynchronous SLBs could break the consistency guarantee, consider a simple scenario in Figure 5 where two SLBs, signaled by an out-of-band controller, initiate a new snapshot. When g_0 initiates snapshot mode and marks $e'_6.m$, it triggers a snapshot at p_0^{in}. However, a new message $e'_2.m$ from p_0^{out} is routed to a different gateway g_1[4], which has not yet entered snapshot mode. This leads to inconsistency: while $e'_5 \in C$ and $e'_4 \to e'_5$, $e'_4 \notin C$.

To block or not to block? An obvious solution would be to block inbound packets at SLBs during a snapshot and only resume forwarding them after all SLBs have 'committed' to the new snapshot. Unfortunately, this method introduces large overheads—not only to the applications, whose response times will spike while the SLB is blocking requests but also to the cloud providers, where the SLB would require large buffers and overprovisioned capacity to drain said buffers after a snapshot.

Rather than trying to enforce consistency, Beaver seeks a method to (a) detect *in*consistency, (b) reject snapshots when they are potentially inconsistent, and (c) minimize the rejection rate. It seeks to do this with near-zero overhead for applications and cloud infrastructure.

4.1 Causal Relevance and Irrelevance

A key idea in Beaver is that, even among the incoming traffic to the in-group, only a subset of that traffic is causally relevant. Using Figure 5 to illustrate, an incoming message, m, is **causally relevant** only when (1) an initiated SLB (g_0) sends a marked message to an in-group node (e.g., p_0^{in}), (2) that node interacts directly or indirectly with an out-group node (e.g., p_0^{out}), and (3) that out-group node sends m back to a different in-group node via an uninitiated SLB (e.g., g_1). Other communication patterns, e.g., an m triggered by an uninitiated process, are **causally irrelevant**.

In essence, causally relevant messages are only produced if the message loop: $GW_A \to IN_A \to OUT \to GW_B$ all occurs within the window of time in which the gateways are propagating snapshot initiation. More formally:

THEOREM 2. *In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway initiating snapshots be $e^{ss}_{gmin}.t = \min_{e^{ss}_g}(e^{ss}_g.t)$ and $e^{ss}_{gmax}.t = \max_{e^{ss}_g}(e^{ss}_g.t)$, $\forall g \in G$, respectively. Also let $\tau_{min} = min(d(g,g';\{p,q\}))$, $\forall g,g' \in G$, $p \in P^{in}$, and $q \in P^{out}$. If $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$, then the partial snapshot is causally consistent.*

PROOF. *We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction.*

Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. As before, there must be some chain $e' \to e^{out} \to e^g \to e$. Because $e' \notin C_{part}$, we have $e^{ss}_{p_j^{in}} \to e'$ or $e^{ss}_{p_j^{in}} = e'$, that is, p_j^{in} must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as $e^{g'}$. By the definition of τ_{min}, we have $e^g.t - e^{g'}.t \geq \tau_{min} > e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. Thus, at event e^g, the gateway must have already initiated the snapshot and will mark $e^g.m$ before forwarding. This results in $e \notin C_{part}$, a contradiction! □

Theorem 2 implications: Informally, this theorem suggests that if the time gap between the first and last SLB snapshot initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) is sufficiently small, or the minimum time for a message to revisit a gateway (τ_{min}) is long enough, causally relevant messages are impossible and the concerned partial snapshot is provably consistent[5].

Causally relevant messages are rare in the real world. Intuitively and with anecdotal evidence, the inequality $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$ can be satisfied with an exceedingly high probability in real-world contexts:

For the LHS ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$): This time gap is essentially the difference in one-way delays between the controller and each of the SLBs. As SLBs share a region with the target service, a well-placed initiator (e.g., equidistant from all target SLBs or one whose messages are forced to travel to the root of the data center fabric) can simultaneously ensure reactive snapshot initiation and $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ of near zero.

For the RHS (τ_{min}): This value includes multiple network hops, extending from an SLB to in-group nodes, then to out-group nodes, and back to an SLB. Particularly when out-group nodes are in other data centers or are end-host clients,

[4] This is typical in ECMP routing, where connections even from the same source may reach different SLBs.

[5] In principle, another sufficient condition is when the in-group snapshot completes quickly enough. We do not rely on this because it has worse scaling properties than SLB convergence, but it can be added as an optimization.

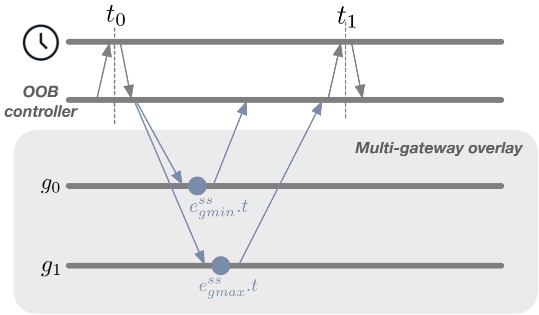


Figure 6: The time difference $t_1 - t_0$ as a safe upper bound for $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ by querying a single hardware clock source with bounded frequency drift.

this value can be orders of magnitude higher than typical values for the LHS—on the order of milliseconds or tens of milliseconds. However, even when the out-group nodes are in the same region or data center as the in-group, we can still expect that this value is higher than any observed delta between initiator-to-SLB one-way delays as it includes *at least three trips* through the data center fabric in addition to processing time at the in/out-group network stacks[6].

4.2 Efficiently Verifying Causal Irrelevance

The primary technical challenge of OGM is minimizing the LHS of the above inequality and efficiently/confidently verifying that the resulting inequality held for a given snapshot, even in the presence of message drops, delays, and other sources of unexpected latency. The cloud provider can compute the two sides of the inequality separately.

4.2.1 Computing a Lower Bound for the RHS

For the RHS, the value can be determined statically, as dynamic network conditions like failures and congestion can only add to the latency of the message sequence. The latency is then equivalent to the sum of each hop's minimum propagation, transmission, and processing delays. These values depend on the relative placements of the in-group nodes, SLBs, and out-group communication partners, but all of those are known at runtime. To ensure a conservative lower bound, operators can and should assume that application-level processing and transmission delays are zero (Figure 13).

4.2.2 Determining an Upper Bound for the LHS

The LHS is harder to compute statically as failures and congestion mean a true upper bound may not exist[7]. Instead, we need to measure an upper bound online for the observed difference between gateway timestamps ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) for the snapshot in question.

	Rubidium	JILA Sr	Quartz	Quartz (calibrated)
Δf	±0.05 ppb	$\pm 2.1 \times 10^{-18}$	±100 ppm	±100 ppb

Table 2: Frequency drift (Δf) uncertainty range of today's clocks, ppb (parts per billion) $= 10^{-9}$, ppm (parts per million) $= 10^{-6}$.

The typical method of measuring time gaps on different machines is via clock synchronization. Although today's clock synchronization techniques can achieve microsecond or sub-microsecond precision, fundamentally, they rely on frequent cross-machine messaging to correct the offset, which is sensitive to congestion and failures, thus impacting the bound on clock drift in the worst case [23, 62]. Data center services like TrueTime provide a reliable interface to query time points and calculate their differences. However, a general timing service incurs higher overhead and a typical clock uncertainty range of 1–7 ms [13], much greater than the timescales relevant for Beaver detection.

Synchronization-free approach. Beaver adopts an alternative, customized approach using a single hardware clock to calculate the elapsed time. As depicted in Figure 6, the controller queries the start time at t_0 from this clock source with a read t^r_0 before initiating a new snapshot. Once the final ACK from the SLBs arrives, it reads the end time t^r_1 at t_1 from the source, where t_0, t_1 represents the global wall clock time, and t^r_0, t^r_1 the actual clock reads. This hardware clock can be a local hardware clock from either a COTS PCIe NIC [46] or from one equipped with an atomic clock, which are increasingly deployed in production data centers [42, 48].

Note that $t_1 - t_0$ is an upper bound on the LHS as $t_1 > e^{ss}_{gmax}.t$ and $t_0 < e^{ss}_{gmin}.t$. Thus, if $t_1 - t_0 < \tau_{min}$, the partial snapshot under examination is consistent. In practice, the time difference $t^r_1 - t^r_0$ is adjusted to account for the maximum frequency drift Δf according to the clock data sheet, to determine an upper bound estimate for the corresponding elapsed time $t_1 - t_0$, thus the detection criteria $(t^r_1 - t^r_0) \times (1 + \Delta f) < \tau_{min}$. This method, which relies solely on a single hardware clock to calculate time differences, eliminates issues common in traditional clock synchronization approaches, such as cross-machine message congestion and errors stemming from delays in clock readings due to software interrupts. The frequency drift of a single clock is relatively low and is mainly deterministically affected by temperature, which has low variance in modern data centers [38, 44, 59]. Standard quartz crystal oscillators in production data centers typically drift by ±100 ppm, or 0.01% error [13, 25, 36, 38, 44]; recent studies are able to reduce this drift of quartz clocks in commodity data center servers to ±100 ppb (10^{-7} error) by calibrating the offset due to temperature variations. More advanced oscillators (e.g., atomic clocks) can reduce this frequency drift by further orders of magnitude [29, 45] (Table 2).

Snapshot invalidation. While ensuring correctness (i.e., no

[6] Even with the detection criteria later described in §4.2, LHS entails only 2 trips and encompasses a simpler data path with SLB stacks that are heavily optimized for minimal processing latency and jittering [15, 18, 22, 63].

[7] Beyond the heat death of the universe or at least the life of a data center.

false negatives), our proposed upper bound adds an additional margin to the original time gap. This margin comprises the clock query latency and the RTT between the controller and the SLBs, which may lead to false positives. In practice, however, we note that many devices support precise hardware timestamping along with the packet data path (i.e., when sending the first notification and when receiving the last notification). Our evaluations on a cloud data center in §7 reveal that the resulting snapshot invalidation rate is $< 5\%$ for typical SLB scales today, even in worst-case scenarios when the out-group nodes are in the same data center and under stressed snapshot operation frequencies.

In the end, false positives—while leading to the invalidation of potentially consistent snapshots—are of little concern due to our system's efficient snapshot operations and its ability to achieve a high snapshot rate.

5 Beaver's Partial Snapshot Protocol

As mentioned previously, Beaver's snapshot 'quantum' is a single VIP—Beaver can provide snapshots for one or more such VIPs within a single region.

Operation. At a high level, Beaver's partial snapshot protocol distinguishes itself from traditional snapshots in two aspects: (1) its lightweight SLB marking logic for inbound traffic and (2) the snapshot verification process at the controller.

In-group processes: Among in-group processes, Beaver inherits its coordination logic (and the omitted, optional recording of in-flight messages) from prior snapshot algorithms [34, 60] that piggyback 'marker' information per message to handle non-FIFO and lossy channels[8]. Figure 7 depicts the core logic: upon receiving a packet, either from an SLB or another in-group process, the current in-group process evaluates if $pkt.sid > csid$. If true, it signals a new snapshot operation: it records the relevant state, updates the local $csid$, and asynchronously notifies the controller of completion. For outgoing packets, if the destination address falls within the scope of in-group processes, the process updates $pkt.sid$ to its current $csid$.

SLBs: As discussed in §4, Beaver instantiates the gateway overlay with the SLBs. For the set of SLBs handling the target in-group process traffic, Beaver embeds logic for marking inbound messages. On receiving an inbound packet, an SLB first checks if the destination VIP is for the in-group [line 16]—since operators may multiplex a single SLB server for multiple VIPs—and modifies the snapshot ID field accordingly. On the control path, the SLB initializes a new snapshot upon receiving an 'INIT' notification from the controller and subsequently sends the acknowledgment to the controller. This process happens out-of-band to avoid biases in the snapshot verification process. Combined, Beaver's gateway logic requires minimal processing and can be incorporated into existing SLB data planes at line rate, including hardware-accelerated ones.

```
• $csid$: Current snapshot ID state for $p \in P^{in}$ or $g \in G$.
– $pkt.sid$: Snapshot ID ($N$b) in SLB encapsulation header.
– $pkt.dst$: Destination address of a user packet.
– $pkt.src$: Source address of a user packet.

1  function IN-OnReceive (pkt):
2      /* Signaled a new snapshot */
3      if $pkt.sid > csid$ then
4          Record the state of interest;
5          Send FIN for $csid + 1, \ldots, pkt.sid$ to the controller;
6          $csid \leftarrow pkt.sid$;

7  function IN-OnSend (pkt):
8      if $pkt.dst \in P^{in}$ then
9          $pkt.sid \leftarrow csid$;

10 function SLB-OnReceive (INIT):
11     if $INIT.sid > csid$ then
12         $csid \leftarrow INIT.sid$;
13         ACK for $csid + 1, \ldots, pkt.sid$ to the controller;

14 function SLB-OnReceive (pkt):
15     /* Mark inbound packet from out-group */
16     if $(pkt.dst \in P^{in}) \wedge (pkt.src \notin P^{in})$ then
17         $pkt.sid \leftarrow csid$;
18     Forward packet to $pkt.dst$;
```

Figure 7: Logic for partial snapshots at in-group processes and SLBs. All control plane operations are asynchronous.

Controller: With Beaver, operators can designate any server with direct or indirect access to a stable clock source, preferably located near the pertinent SLBs, as the controller. The core logic to initiate snapshots, shown in Figure 8, involves continuously sending INIT commands to SLBs to initiate new snapshots. The protocol maintains the number of snapshots in flight and controls the snapshot frequency. The detection of invalid snapshots follows the methodology outlined in §4.2: The controller queries the clock read for t_0 before sending notifications [line 5] and uses the clock reads upon receiving the last ACK to determine the snapshot's validity [line 20]. It the local NIC supports hardware time-stamping capabilities, `queryClock()` can occur along the data path during the send of the first INIT notification and the receive of the last ACK response.

Handling packet loss, delay, and reordering. Beaver is robust to faults in data- and control-plane communications.

Data plane: Unlike the original Chandy-Lamport protocol, which relies on separate marker messages, Beaver draws inspiration from subsequent variants [34, 60] to incorporate marker information by piggybacking it into existing traffic. This piggybacking makes Beaver inherently resilient to 'marker' losses and reordering on the data path, whether these occur within the network core or the host networking stacks.

Control plane: Although timely and reliable delivery of control messages can be beneficial (e.g., through an alternate port that is dedicated to control tasks) Beaver does not de-

[8] Optional broadcast of marker messages from SLBs to in-group processes may accelerate the snapshot convergence when service traffic is infrequent.

```
• $csid$: The next snapshot ID to initiate at the controller.
• $receivedFIN[sid][p]$: If received FIN from $p \in P^{in}$ for $sid$.
• $receivedACK[sid][g]$: If received FIN from $g \in G$ for $sid$.
• $t_0[sid]$: Timestamp $t_0$ for $sid$.
– $FIN.p$: The source process sending the FIN.
– $FIN.sids$: The associated $sid$(s) of the FIN.
– $ACK.g$: The source SLB sending the FIN.
– $ACK.sids$: The associated $sid$(s) of the ACK.
1  function Controller-OnSnapshot():
2      $num\_inflight\_ss = 0$, $csid = 0$;
3      while $num\_inflight\_ss < 2^{N-1} - 1$ do
4          /* Optional rate-limiting for less greedy snapshots */
5          $t_0[csid] = queryClock()$;
6          Send $INIT$s ($INIT.sid = csid$) to all $g \in G$;
7          $num\_inflight\_ss$ += 1, $csid$ += 1;
8  function Controller-OnReceive(FIN):
9      for $sid \in FIN.sids$ do
10         $receivedFIN[sid][FIN.p] = 1$;
11         /* Check all FINs received with bitwise negation */
12         if $\sim receivedFIN[sid][\cdot] == 0$ then
13             $num\_inflight\_ss$ −= 1;
14             $receivedFIN[sid][\cdot] = 0$;
15 function Controller-OnReceive(ACK):
16     for $sid \in ACK.sids$ do
17         $receivedACK[sid][ACK.g] = 1$;
18         /* If all ACKs received */
19         if $\sim receivedACK[sid][\cdot] == 0$ then
20             if $(queryClock() - t_0[sid])(1 + \Delta f) < \tau_{min}$ then
21                 /* Accept the snapshot */
22             else
23                 /* Invalidate the snapshot */
24             $receivedACK[sid][\cdot] = 0$;
```

Figure 8: Main controller logic for continuous snapshots.

pend on it for its core functionality. It operates effectively even with unreliable transport protocols such as UDP and it requires only a negligible number of control messages: $|P^{in}|$ FIN messages (or less as members of the in-group, P^{in}, can batch updates in a single ACK on the increments in prior snapshots), $|G|$ INIT commands, and $|G|$ ACK responses for each snapshot.

While delays or losses of the above messages might slow down the snapshot rate—a minimal impact as observed in our evaluation—they do not compromise the correctness of Beaver. The controller, in response to any delays or losses, simply invalidates the affected snapshot.

Handling failures. One important problem is how to handle failures of the SLBs and backend servers. Fortunately, most public clouds today already apply central management mechanisms that ensure fault tolerance and state consistency during changes in membership of machines for each VIP[9] [10, 15, 22, 50]. Operating on top of the abstraction, Beaver's controller coordinates with the SLBs and backend servers belonging to the requested VIP (as indicated by the current central state), incurring minimal additional costs and deployment complexity. To handle failure events during a snapshot, Beaver incorporates a single ACK mechanism (Figure 8): if the controller does not receive the ACK from an SLB or an in-group process, Beaver simply invalidates the snapshot or drops affected states while guaranteeing correctness.

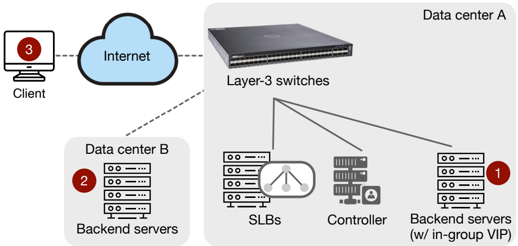


Figure 9: Evaluation setup considering three different out-group locations: within the same data center, data center of a different region, or on the Internet (from a local laptop).

Supporting parallel snapshots. Many cases, such as event-driven or telemetry tasks, require higher-frequency state capture [60, 61]. Rather than waiting for the completion of one snapshot before initiating another, limiting the snapshot rate to the slowest component in the snapshot convergence process, Beaver can initiate snapshots concurrently. The controller ensures that the number of packets in flight remains within $2^N - 1$ [line 3 in Figure 8], the maximum concurrent snapshots supported by the header field sid. The extra $^{-1}$ in the exponent is to eliminate ambiguities in comparator operations at in-group processes [line 3 in Figure 7] under worst-case wrap-around conditions.

Beaver also supports parallel snapshots for distinct groups of VIPs without needing extra metadata. This is facilitated by the SLBs' ability to naturally segregate operations based on VIP information. Consequently, the same sid header space can be utilized for simultaneous snapshots across groups with non-overlapping VIPs.

6 Implementation

We implement a Beaver prototype on a cloud data center [14] (Figure 9) that aligns with a production setup [15, 28, 50].

Supporting SLB-associated functionalities. We implement an end-to-end workflow to mirror the behaviors associated with SLBs in production data centers [15, 28, 50]. Additionally, our system facilitates automated service discovery operations through an out-of-band controller server.

SLB implementation: Our setup configures DELL EMC PowerSwitch S4048-ON [1] for layer-3 ECMP forwarding based on service VIPs to SLBs. Emulating prior work [15, 28], we implement the core SLB functions with DPDK [19], involving around 1860 lines of C/C++ code. Each SLB maintains an in-memory connection flow state, employs consistent hashing on the 5-tuple of each packet to determine the appropriate backend server, and caches the decision for future

[9] Unlike the DIP caching feature in §2.1, the consistency mechanism was originally absent in [50], but later incorporated as an essential component.

decisions. Then, the SLBs encapsulate the inbound packet's header and forward it to the backend server with the destination DIP. To maximize utilization of SLB servers, we perform load balancing across different CPU cores using RSS.

Backend servers: To maintain transparency for the upper-layer applications, we implement the re-computation of checksums, NAT caching in a shared eBPF map, and the de-encapsulation of incoming packets from the SLB via XDP [27]. For outbound packets, we instrument the Linux `tc` to look up the NAT entries and perform the header transformations to replicate Direct Server Return (DSR). In total, they involve 1040 lines of C/C++ code.

Topology. Our testbed supports typical communication patterns, encompassing a variety of out-group positions, including other VIPs within the same data center, VIPs in other data centers, and Internet clients—all through the layer-3 switches and SLBs, along with DSR on the return path. We scale up to 16 SLB servers, each capable of supporting 64 in-group processes, due to limits in resource availability. Our current testbed servers are equipped with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and dual-port ConnectX-4 Lx NICs.

Integrating the Beaver protocol. We implement Beaver's partial snapshot protocol from §5. The SLBs append a snapshot ID to inbound packet headers that encapsulate the destination DIP and the source SLB IP. The in-group processes and SLBs embed Beaver's snapshot logic from Figure 7 through XDP and DPDK. The additional logic involves 68 lines of C++ for SLB data-path logic and 102 lines of C codes for eBPF at in-group processes. The controller server, following Figure 8, automates the initiation, control, collection, and verification of snapshots. We use UDP for bi-directional control messages with SLBs and unidirectional messages from in-group servers. The controller currently exploits local NIC hardware timestamping (`SOF_TIMESTAMPING_RAW_HARDWARE`) for precise timing of INIT and ACK messages on their data path [47].

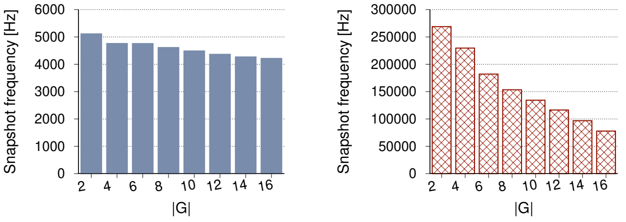
7 Evaluation

Our evaluation focuses on exploring the following questions.

- Can Beaver sustain fast snapshot rates? How does the scale of the in-group nodes and SLBs affect? (§7.1)
- What about effective snapshot rates? How often do Beaver invalidate snapshots in cloud data centers? (§7.2)
- Does Beaver's distributed coordination affect the existing service traffic? (§7.3)
- How does Beaver help real-world services? (§7.4)

7.1 Beaver Supports Fast Snapshot Rates

To stress-test Beaver, unless otherwise specified, our evaluation runs Beaver at very high snapshot frequencies. To further ensure that our performance/overhead results are conservative, state capture in the snapshots are NOPs. Real local record operations (which are application-dependent and orthogonal to the study of distributed snapshot protocols) will only result in less contention and overhead.



(a) w/o parallelism (b) w/ parallelism

Figure 10: Beaver's sustained snapshot frequency versus a strawman approach with blocking operations at varying scales of SLBs and backend processes.

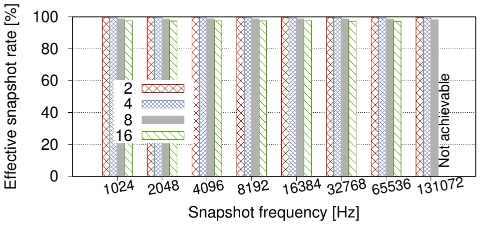


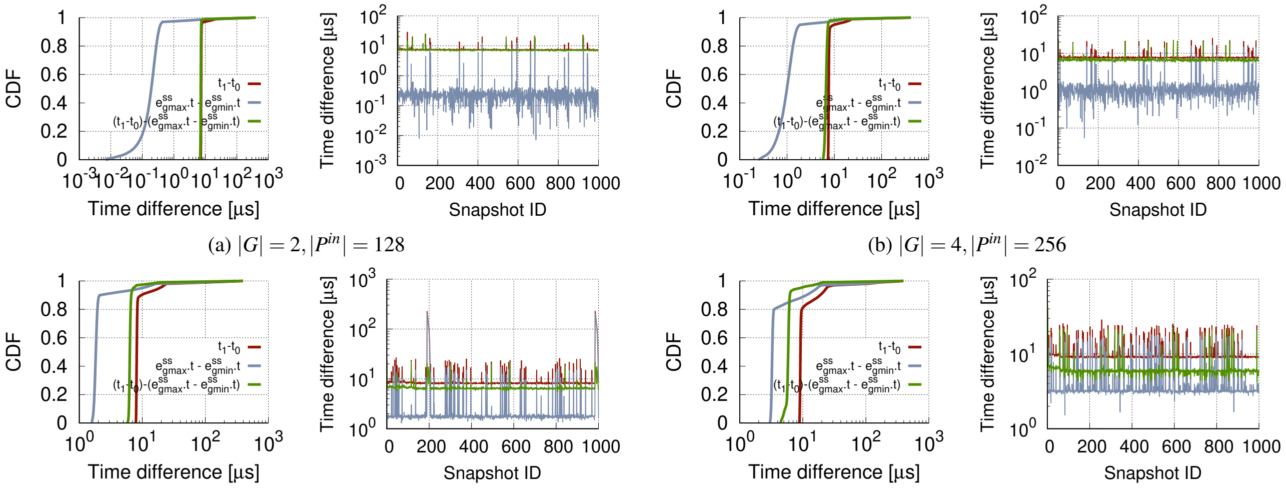
Figure 11: Beaver's effective snapshot rates under varying snapshot frequencies and in-group process scale.

As a measure of Beaver's efficiency and scalability, even at these high rates, Beaver exhibits good performance. Figure 10 shows the maximum snapshot rate compared to a strawman approach, which waits for completion before initiating another. The maximum rate is determined by increasing the snapshot frequency until we observe backlogs in the ACK and INIT message notification queue. We vary the number of gateways ($|G|$) up to 16, aligning with typical values for SLBs assigned to a VIP.

The baseline is limited by the snapshot convergence time, which depends on factors such as scale, traffic pattern, and topology. In contrast, Beaver's parallel snapshot capability significantly enhances the rate and shifts the bottlenecks to the processing power of the controller's CPU. Even at the maximum scale, Beaver reaches a snapshot rate of $>$ 77000 Hz, $>$ 18× that of the strawman. In practical applications, leveraging a more powerful processor or scaling the controller server could further improve its speed.

7.2 Beaver Invalidates Snapshots Infrequently

With a high snapshot frequency, how does Beaver perform in terms of effective snapshot rates? Recall in §4.2, Beaver uses an upper bound $t_1 - t_0$ for the time gap between SLB initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) to eliminate the need for time synchronization, it invalidates a snapshot if the bound is greater than τ_{min}, the minimum time to for an external causal chain to occur. While this upper bound ensures correctness, it may reject snapshots and reduces the effective snapshot rate.



(a) $|G| = 2, |P^{in}| = 128$

(b) $|G| = 4, |P^{in}| = 256$

(c) $|G| = 8, |P^{in}| = 514$

(d) $|G| = 16, |P^{in}| = 1024$

Figure 12: CDF of Beaver's upper bound $t_1 - t_0$ with the ground truth ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) for $>$ 10M snapshots and a zoom in to its snapshot series, under stressed scenario with 65536 Hz snapshot frequency and varying number of SLBs/processes.

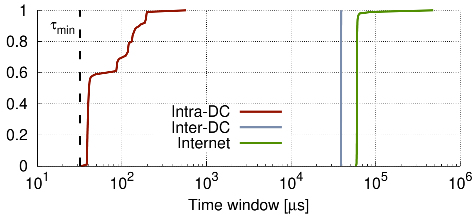


Figure 13: Measurement of the minimum time window for a external causal chain to occur under worst case conditions.

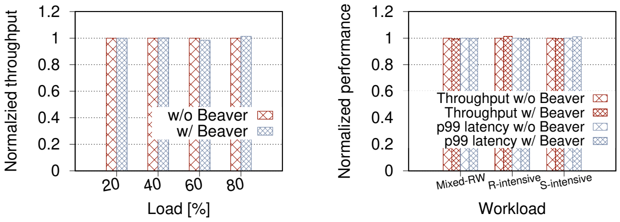
To measure the time for an external causal chain to occur, we consider three distinct scenarios for out-group process locations in Figure 9. In each scenario, we set up a worst-case condition where, immediately following an SLB's snapshot initiation, the SLB forwards an inbound packet to the closest in-group node. The in-group node then loopbacks an immediate message to out-group node with the shortest path, which bounces the packet back to any SLB. Figure 13 shows that the intra-DC scenario results in the shortest time window, resulting in τ_{min} as $33\mu s$. This value is robust because, even though varying cloud conditions often cause latency spikes, they primarily affect the tail rather than the minimum.

To stress test Beaver's performance, we focus on the worst-case scenarios with out-group processs located within the same data center. For other scenarios, τ_{min} is significantly greater, leading to 100% effective snapshot rates across 10M snapshot operations. We execute Beaver in various experimental settings, including scale and snapshot frequencies. For each configuration, we calculate the effective rate based on more than 10 million snapshots. The results, as in Figure 11, reveal that the proportion of snapshots invalidated by Beaver is remarkably low even under the maximum operating frequencies and scales of our testbed.

To better understand the results, we compare the recorded upper bound estimation of $t_1 - t_0$ with the true ground truth $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. As the two events e^{ss}_{gmax} and e^{ss}_{gmin} occur on separate SLB machines, we synchronize the clocks of all SLBs to controller's PTP master clock over symmetric paths without contending traffic, which reports maximum 50 ns offsets during the ground truth measurement. This step, meant solely to understand the behavior, should not be confused with Beaver's clock-synchronization-free approach. Figure 12 shows the comparison over $>$ 10M snapshots when Beaver operates at a frequency of 65536 Hz. Overlapping tails of $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ and the heads of $t_1 - t_0$ are expected—the cdf of the pairwise calculation of $(t_1 - t_0) - (e^{ss}_{gmax}.t - e^{ss}_{gmin}.t)$ for each snapshot clearly demonstrates that the upper bound is strictly higher than the ground truth SLB initiation time gap. The observed outliers in $t_1 - t_0$ are typically due to queueing in our manager's processing queue at high rates or asynchrony in SLB initiations. Furthermore, the margin introduced by $t_1 - t_0$ over $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ is due to the RTT between the controller and the SLBs, which is used to ensure the theoretical upper bound without clock synchronization.

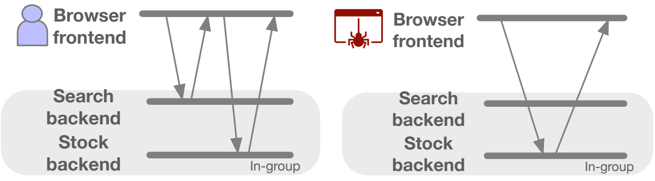
7.3 Beaver Incurs Near-zero Impact

We also stress test the overhead of Beaver on user traffic. Figure 14a compares throughput with and without Beaver under the 65536 Hz snapshot frequency and the max scale of our testbed. `iperf` clients send traffic with varying degree of the total consumed bandwidth capacity of the 16 SLBs. We also run YCSB benchmark workloads [12] with varying mix of read, update, and scan operations, as shown in Figure 14 for backend servers running CassandraDB [6]. The requests follow zipfian distributions, and the scan length adheres to the uniform distribution.



(a) Stressed workloads (b) YCSB benchmarks

Figure 14: Performances with and without Beaver's overhead, normalized to the value without Beaver.



(a) Benign (b) Bot access

Figure 15: Example benign and bot access patterns.

The results of various performance metrics are almost identical, confirming that Beaver has a near-zero effect on service traffic. This is because Beaver, by design, eliminates any delay or blocking operations on the data path for distributed coordination, and the lightweight control path messages are orthogonal.

7.4 Use Cases

We also examine several use cases of Beaver. These examples are intended as instruments through which we can understand its potential utility, differences versus traditional snapshots, and the semantics of its causal consistency guarantee under partial deployments that were previously impossible.

7.4.1 Detecting Anomalous Access

Web applications often feature a JavaScript browser frontend for user interaction and a backend providing service APIs. Consider a legitimate user access in an e-commerce application (Figure 15a). The frontend calls a Search API `fetch("example.com/api/v1/search")`, followed by a Stock API `fetch("example.com/api/v1/get_stock")` for product details. However, malicious traffic, such as web scrapers, might bypass the initial search stage and directly query the stock backend, potentially overwhelming the server. This type of traffic can be challenging to detect as it differs from legitimate traffic in intent rather than content [16, 31].

Beaver can help detect such anomaly patterns, as its partial snapshot can capture the external dependency of these requests, even though it occurs through communication with the Internet. To illustrate, we run a varying mixture of benign and illegal bot clients on our testbed. The backend servers maintain per-client request count in a BPF map through double buffering, so as to 'freeze' the current state through a single switch of the pointer and minimize the impact of blocking local record calls. Table 3 shows detection results calculated against the ground truth. We find that Beaver can accurately recognize the interdependence between the accesses. For example, when all clients are benign, Beaver consistently results in true negatives, aligning with the ground truth. However, a polling-based approach and traditional snapshots (L-Y) can result in false positives due to interpretations of erroneous capture of higher counts at the Search backend than at the Stock backend.

Method	Bot ratio = 0% TP, FP, TN, FN	Bot ratio = 5% TP, FP, TN, FN	Bot ratio = 10% TP, FP, TN, FN
Polling	0, 0.005, 0.995, 0	0.005, 0.062, 0.874, 0.059	0.069, 0.136, 0.666, 0.129
L-Y	0, 0.005, 0.995, 0	0.001, 0.058, 0.886, 0.055	0.011, 0.105, 0.783, 0.101
Beaver	0, 0, 1, 0	0.053, 0, 0.947, 0	0.113, 0, 0.887, 0.001

Table 3: Beaver's detection accuracy versus (1) polling-based approach using time synchronization, and (2) Lai-Yang algorithm, a state-of-the-art global snapshot protocol.

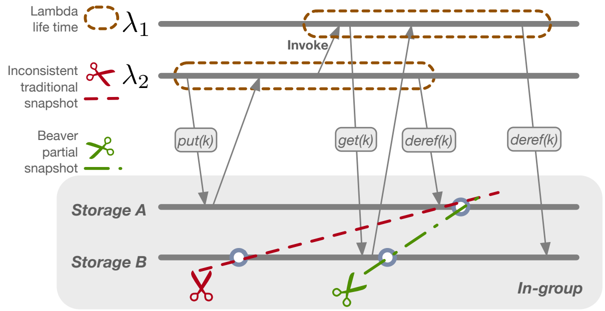


Figure 16: Garbage collection for the ephemeral storage for serverless analytics.

7.4.2 Serverless Garbage Collection

Backend services that support serverless applications are also a natural fit, as requests to serverless functions rely on schedulers and logic that are not visible to the backend services or the serverless functions themselves. Consider an application that provides storage for a serverless analytics job and uses reference counting for garbage collection [32]. The storage service deploys multiple servers for scalability and supports three primary APIs: `get()`/`put()`, which fetch/upload the object and increment the reference counter, and `deref()`, which indicates that the previously fetched object is no longer in use and decrements the reference counter.

Beaver's consistent partial snapshots can support safe garbage collection decisions. To illustrate, we instantiate two serverless functions through [30] that follow the workflow of Figure 16 on our testbed. The backend storage maintains an in-memory state of reference counters for each KV object. When a reference counter reaches 0 in a snapshot, the controller informs the backends to recycle the correspond-

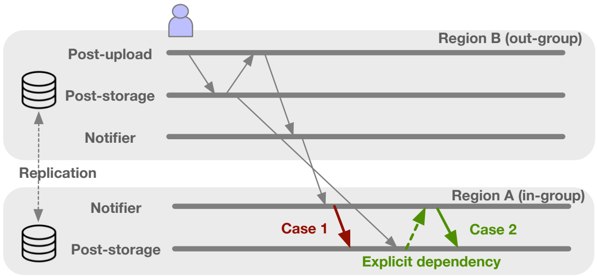


Figure 17: A simplified example of geo-distributed social media application [17] which includes distinct services such as post-upload, post-storage, and notifier.

ing object. During invocations, we also record the incident counts of invalid `get()` access or `deref()` calls. We find that, across invocations, the Lai-Yang algorithm may produce inconsistent snapshots (shown in Figure 16) that indicate *no* open references to the object—however, λ_1 is still keeping a reference to it. This leads to unsafe decisions to recycle the object associated with the key and results in an observed invalid call percentage of 23–29%. In contrast, Beaver's partial snapshots guarantee causal consistency even in the presence of external communication that ensures safe reclamation of the object and consistently results in 0 invalid calls.

7.4.3 Integration Testing

Integration testing, commonly used in CI/CD pipelines [24], extends the coverage of testing to inter-service logic. Unfortunately, applying it to distributed applications can be challenging. Consider the example shown in Figure 17, a violation of the application specification occurs when followers in a region receive a notification and request the storage DB (case 1) before the cross-region protocol actually replicates the post data. Recent solutions [17, 53] address the inconsistencies by forming explicit dependencies (case 2). However, the involvement of auxiliary services and additional dependencies make it difficult to capture a holistic snapshot.

Beaver offers a practical abstraction to test distributed applications by enabling partial deployment and capturing causal dependencies relevant to the local service. By snapshotting states in post-storage and notifier services, developers can write test cases to verify the crucial invariant above: the presence of a post in the storage must always precede its corresponding notification in the notifier service. In particular, Beaver's guarantee of causal consistency means that if a canary solution is correct, a partial snapshot observing a log in the notifier must have captured the data entry of the corresponding version in post-storage. Therefore, a single violating test case will suggest the presence of bugs.

7.4.4 In-flight Message Tracking

We also revisit the example in Figure 3. As mentioned in §2.2, a useful query is to estimate the number of concurrent requests, which can inform resource provision decisions. Figure 18a illustrates a scenario with only one active request. In theory, traditional snapshots, which fail to capture the causality between the client's follow-up request and the prior response, can give an overestimation of 2 in-flight messages (indicated by the cut in red). Beaver, in contrast, can capture the external causality and results in an estimation of no more than 1 message in flight (indicated by the cut in green).

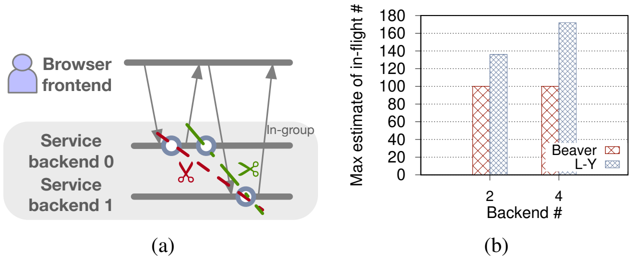


Figure 18: (a) Example snapshots for in-flight message tracking. (b) Comparison of estimated number of in-flight requests with and without Beaver.

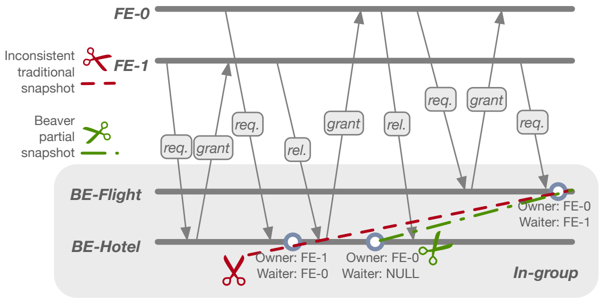
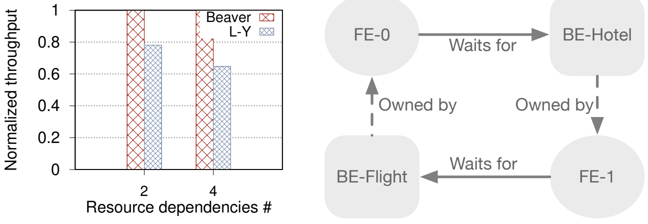


Figure 19: An example of deadlock detection using distributed snapshots.

To validate the behavior in practice, we run 100 clients concurrently that conform to the poisson arrival pattern on our testbed. Each backend process maintains a total request and response count value using a BPF map. Thus, the difference between the two counters indicates the number of messages in flight. The controller then collects the snapshot of counter values and then obtains the aggregate estimate. Figure 18b shows that traditional snapshots can overestimate the number of concurrent requests by more than 30%, while Beaver's result consistently matches the ground truth. Worse, a higher number of backends will lead to an overestimation further divorced from reality.

7.4.5 Distributed Deadlock Detection

A classic use of distributed snapshots is deadlock detection, a fundamental problem in distributed systems. Consider the scenario in Figure 19, where the machines of a frontend service interact with a reservation microservice to book flights and hotels on behalf of its clients. Here, a frontend server acquires a lock from the backend server for a target resource ID and



(a) (b)

Figure 20: (a) Comparison of transaction throughput (normalized to Beaver). (b) WFG for the inconsistent snapshot in Figure 19.

releases it after completing its transaction. A deadlock may occur when a client requests resources that are held by others, forming a directed cycle in the resource dependency graph (known as Wait-For Graphs or WFGs). As these systems (such as those used by Airbnb and Uber) encompass thousands of microservices, each with its own sovereignty [20,64], global snapshots are challenging and expensive to enforce.

Beaver, however, is amenable to only taking partial snapshots of the reservation service. To illustrate, we run backend processes that maintain the ID list of client(s) currently owning/waiting for the local resources in memory. When the controller detects a deadlock based on a snapshot, it informs backend processes to abort the current transaction. We emulate clients that request backend resources in random order and measure the resulting transaction throughput. Figure 20a shows that the traditional snapshot algorithm can suffer from more than 20% throughput drops compared to Beaver. This is because, without accounting for the external message dependencies, it can render a snapshot that is inconsistent (Figure 19), which leads to false deadlocks (Figure 20b) and the unnecessary costs of deadlock resolution operations. Beaver, on the other hand, guarantees safe detection.

8 Discussion

Instantiating Beaver gateways. Beaver focuses on public clouds, which already contain SLBs, imposing minimal changes and costs to integrate its functionality. We argue that these are where partial snapshots are most important as smaller private clouds are easier to modify wholesale [51]. Without cloud providers' support, cloud tenants could also deploy their own Beaver-compatible gateways on virtual machines (e.g., Network Virtual Appliances (NVAs) [7]) to ensure consistency under external communication with clients and human users. This involves additional costs and complexities and can be suitable if NVAs are already in use, e.g., to provide firewall functionality.

Optimizing local record operations. Similar to classic distributed snapshot protocols (§2.2), Beaver is agnostic to the semantics of local record operations. An interesting problem—orthogonal to the core mechanism of Beaver—is to enable efficient local-state capturing mechanisms, especially when the user desires a large target state or a high snapshot frequency. Besides application-specific practices in §7.4, we postulate that a more generic and opportunistic approach may minimize their online impacts by focusing on state changes during IDLE times of the application. We leave a complete exploration for future work.

9 Related Work

Distributed snapshots. This work builds on the large array of classic distributed snapshot algorithms [11,26,33,34,41,56,57,60]. To the best of our knowledge, Beaver formalizes, designs, and implements the first partial snapshot primitive that extends their capabilities for practical usage.

Cloud data centers. Beaver is also related to works on various facets of cloud data centers, including layer-4 load balancers [10,15,22,43,50,63] and its clock services [13,23,25,36,38,42,44,48,59,62]. For the former, Beaver integrates its gateway marking logic based on the behaviors of SLBs fundamental to cloud data center services and implements a practical prototype aligned with today's setups. Meanwhile, Beaver builds on extensive measurement studies that highlight the reliable properties of frequency drifts of a single clock. Combined, Beaver presents a unique design without making any assumptions about clock synchronization that ensures consistent, high-rate partial snapshots under external interactions while incurring minimal changes and impacts to current operations and service traffic.

10 Conclusion

This paper rethinks the classic distributed snapshots and observes the mismatch of their assumptions with today's cloud services. With it, we present Beaver, the first partial snapshot primitive that advances the capabilities of existing snapshots for practical usage in distributed cloud services. Central to Beaver is the design and instantiation of a novel optimistic gateway marking primitive. Beaver presents a unique design point by tightly integrating the protocol with the regularities of data center networks. Our evaluation demonstrates that Beaver not only can capture partial snapshots at high speed, but it also incurs near-zero costs to existing service traffic.

Acknowledgments

We gratefully acknowledge our shepherd Nitin Agrawal and the OSDI 2024 reviewers for all their help and constructive comments. We also thank Behnaz Arzani, Linh Thi Xuan Phan, Wyatt Lloyd, and João Ferreira Loff for the valuable feedback. This work was funded in part by NSF grants CNS-1845749, CCF-2124184, and CNS-2321726.

References

[1] Dell emc powerswitch s4048-on support page. https://www.dell.com/support/home/en-us/product-support/product/force10-s4048-on/docs.

[2] Hugging face models. https://huggingface.co/models.

[3] Apache kafka. https://kafka.apache.org/, 2024.

[4] Remzi Can Aksoy and Manos Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 385–398, 2019.

[5] Apache Flink. Apache flink: Stateful computations over data streams. https://flink.apache.org/, 2024.

[6] Apache Software Foundation. Apache cassandra. https://cassandra.apache.org/, 2021.

[7] Aviatrix. Azure network virtual appliance. https://aviatrix.com/learn-center/cloud-security/azure-network-virtual-appliance/, 2024.

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[9] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.

[11] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[12] Brian Frank Cooper et al. Yahoo! cloud serving benchmark (ycsb), 2021. Available at https://github.com/brianfrankcooper/YCSB.

[13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[15] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[16] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE, 2009.

[17] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing cross-service causal consistency in distributed applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 298–313, 2023.

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[19] Linux Foundation. Data plane development kit (DPDK), 2015.

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[21] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang (Harry) Liu, and Ming Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 473–485, Santa Clara, CA, July 2015. USENIX Association.

[22] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[23] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[24] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, et al. Conveyor: One-tool-fits-all continuous software deployment at meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[25] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 550–563, 2023.

[26] Jean-Michel Helary. Observing global states of asynchronous distributed applications. In *Distributed Algorithms: 3rd International Workshop Nice, France, September 26–28, 1989 Proceedings 3*, pages 124–135. Springer, 1989.

[27] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[28] Facebook Incubator. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2023.

[29] IQD Frequency Products. Iqrb-1 high-performance rubidium oscillators. https://www.digikey.com/en/product-highlight/i/iqd-frequency-products/iqrb-1-high-performance-rubidium-oscillators.

[30] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 287–300, 2005.

[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[33] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering*, 2(4):224, 1995.

[34] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[36] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[37] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

[38] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 1171–1186, 2020.

[39] Yunhe Liu, Nate Foster, and Fred B Schneider. Causal network telemetry. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 46–52, 2022.

[40] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and control*, 62(2-3):190–204, 1984.

[41] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of parallel and distributed computing*, 18(4):423–434, 1993.

[42] Meta. Precision time protocol at meta. https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/, November 2022.

[43] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[44] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, 2022.

[45] Travis L Nicholson, SL Campbell, RB Hutson, G Edward Marti, BJ Bloom, Rees L McNally, Wei Zhang, MD Barrett, Marianna S Safronova, GF Strouse, et al. Systematic evaluation of an atomic clock at 2× 10- 18 total uncertainty. *Nature communications*, 6(1):6896, 2015.

[46] NVIDIA Corporation. Connectx-6 dx ethernet adapter cards datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[47] NVIDIA Corporation. Time stamping - nvidia networking docs. https://docs.nvidia.com/networking/display/ofedv502180/time-stamping.

[48] Open Compute Project. Time appliance project. https://opencomputeproject.github.io/Time-Appliance-Project/.

[49] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[51] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[52] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

[53] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423, 2020.

[54] Ticket News. Ontario man 'heartbroken' after ticketmaster cancels 'double-sold' seat. `https://www.ticketnews.com/2018/03/paul-simon-fan-scored-floor-seats-had-them-revoked-by-ticketmaster-after-seat-was/`, 2018.

[55] Peter Van Roy and Angel Bravo Gestoso. Saturn: A distributed metadata service for causal consistency. In *EuroSys 2017 Conference*, 2017.

[56] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3 edition, 2017.

[57] S Venkatesan. Message-optimal incremental snapshots. In *Proceedings. The 9th International Conference on Distributed Computing Systems*, pages 53–54. IEEE Computer Society, 1989.

[58] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 701–718. USENIX Association, November 2020.

[59] Nofel Yaseen, Behnaz Arzani, Krishna Chintalapudi, Vaishnavi Ranganathan, Felipe Frujeri, Kevin Hsieh, Daniel S Berger, Vincent Liu, and Srikanth Kandula. Towards a cost vs. quality sweet spot for monitoring networks. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 38–44, 2021.

[60] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[61] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the 2020 ACM SIGCOMM 2021 Conference*, pages 296–309, 2020.

[62] Liangcheng Yu, John Sonchack, and Vincent Liu. Orbweaver: Using idle cycles in programmable networks for opportunistic coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1195–1212, 2022.

[63] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358. USENIX Association, 2022.

[64] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. Crisp: Critical path analysis of large-scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

Fast and Scalable In-network Lock Management Using Lock Fission

Hanze Zhang[1,2,4] Ke Cheng[1,3] Rong Chen[1,2,3] Haibo Chen[1,3,5]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University [2]Shanghai AI Laboratory
[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[4]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University
[5]Key Laboratory of System Software (Chinese Academy of Sciences)

Abstract

Distributed lock services are extensively utilized in distributed systems to serialize concurrent accesses to shared resources. The need for fast and scalable lock services has become more pronounced with decreasing task execution times and expanding dataset scales. However, traditional lock managers, reliant on server CPUs to handle lock requests, experience significant queuing delays in lock grant latency. Advanced network hardware (e.g. programmable switches) presents an avenue to manage locks without queuing delays due to their high packet processing power. Nevertheless, their constrained memory capacity restricts the manageable lock scale, thereby limiting their effect in large-scale workloads.

This paper presents FISSLOCK, a fast and scalable distributed lock service that exploits the programmable switch to improve (tail) latency and peak throughput for millions of locks. The key idea behind FISSLOCK is the concept of lock fission, which decouples lock management into grant decision and participant maintenance. FISSLOCK leverages the programmable switch to decide lock grants synchronously and relies on servers to maintain participants (i.e., holders and waiters) asynchronously. By using the programmable switch for routing, FISSLOCK enables on-demand fine-grained lock migration, thereby reducing the lock grant and release delays. FISSLOCK carefully designs and implements grant decision procedure on the programmable switch, supporting over one million locks. Evaluation using various benchmarks and a real-world application shows the efficiency of FISSLOCK. Compared to the state-of-the-art switch-based approach (NetLock), FISSLOCK cuts up to 79.1% (from 43.0%) of median lock grant time in the microbenchmark and improves transaction throughput for TATP and TPC-C by 1.76× and 2.28×, respectively.

1 Introduction

Distributed lock services are essential building blocks for coordinating concurrent access to shared resources in numerous distributed systems, such as OLTP databases [23, 62, 67], file systems [18, 51], and rich cloud-based systems [1, 22, 25, 52]. Modern distributed systems commonly rely on fine-grained locks to concurrently access near-billion-scale datasets, such as files and directories [57, 58, 61], database tuples [10, 12, 14], and knowledge graphs [2, 4, 15].

With the prevalence of affordable high-performance networks (e.g., RDMA) and high-capacity persistent memory (e.g., Intel Optane) in modern datacenters, it is not uncommon to see microsecond-scale execution time in distributed in-memory systems [20, 60, 66, 71–73, 78] (see Table 1). As a result, the overhead of granting locks (10–100 µs) becomes non-trivial (e.g., comparable to task execution time) and even dominates the end-to-end performance [62, 75, 76].

Distributed lock managers [6, 13, 30, 54, 55, 65, 76] are commonly designed in a centralized manner to handle lock requests and grant locks. This makes it easy to enable powerful features such as latency predictability [31, 38, 46], starvation freedom [36], and performance isolation [76]. Specifically, before and after accessing a set of objects, the corresponding locks must be acquired from and released to the lock manager, which acts as a central point for granting and managing locks. Traditional lock managers rely on commodity servers to serve lock requests, which imposes one network round-trip overhead in granting locks and often incurs significant queueing delay due to limited request processing throughput of the server CPUs.

To overcome these drawbacks, it has recently been proposed to use the programmable switch as a centralized lock manager to host part of locks [76], as it offers lower latency and higher throughput than servers for packet processing. Further, using the switch to handle lock requests halves the network overhead due to its central network location. However, due to the limited switch memory (typically just a few MB), only a small fraction of locks (e.g., less than 10,000 [76]) can be hosted on a programmable switch for large-scale workloads. This is mainly because the variable-size metadata of a lock—a set for holders and a queue for waiters—consumes several hundred bytes of switch memory. Moreover, the performance of existing switch-based approaches is heavily dependent on the workloads, which must be both highly skewed and predictable to achieve significant improvement. It is also difficult, or even impossible, to dynamically update the data plane model of a programmable switch for exchanging locks.

Key insight. The centralized lock manager can be divided into two phases: *synchronous* grant decision and *asynchronous* participant maintenance. Making a grant decision is based solely on the *fixed-size* metadata (lock mode),

Table 1: *Task execution time of distributed in-memory systems.*

Systems	Workload	Exec. Time
Txn. Processing [37, 73]	TPC-C / TATP	7 / 2.8 µs
File System [18, 51]	Read / Write / Mkdir	1 / 10 / 20 µs
Key-value Store [71, 78]	Search / Insert / Delete	8 / 15 / 12 µs
Online Trading [20]	Trade Execution	10 µs

while maintaining participants also requires the remaining *variable-size* metadata (holders and waiters).

Our approach. Armed with the above insight, we design FissLock, a *switch-centric* lock managing system that also leverages programmable switches but in a new way to offer significant performance improvement (both latency and throughput) and memory efficiency for millions of locks. The key idea is *lock fission*, which decouples grant decision and participant maintenance procedures of the lock manager and deploys the two parts on the programmable switch and commodity servers, respectively. Specifically, the switch acts as the *decider* that immediately makes a decision and replies to the requester if the lock is granted. Meanwhile, the lock request is further routed to the server, hosting the *agent* of the lock, for updating lock holders and waiters with rich semantics. The main advantages of lock fission are three-folds. First, it stores only a small, fixed-size lock mode in switch memory to accelerate millions of locks, which is two orders of magnitude larger than existing switch-based approaches. Second, it leverages high-speed, line-rate request processing of the switch to concurrently grant locks with lower latency and higher throughput than server-based approaches. Third, it delegates lock management tasks (i.e., maintaining holders and waiters) to the server of the lock holder, enhancing load balance and data locality for diverse workloads.

FissLock proposes the first design of the lock fission protocol, which splits the lock manager into a centralized, stationary decider on the switch, and migratable agents for each lock on the servers. The protocol introduces new workflows for acquiring and releasing locks, which allow for halfway responses from the switch when acquiring grantable locks and migrating agents among servers to exploit locality and balance lock management loads. FissLock further addresses the anomalies in the protocol caused by network exceptions using incarnation checks.

FissLock stores small fixed-size metadata (e.g., lock mode) for each lock in the switch memory. By carefully designing on-switch metadata structure, a single switch with a few MB of memory can host millions of locks. To implement the decider of the lock fission protocol on an ASIC-based programmable switch (e.g., Intel Tofino [16]), FissLock devises a 6-stage pipeline to process four types of packets in the protocol. Each stage employs one or more match-action units in the switch data plane to perform simple operations that a programmable switch can afford, such as metadata matching and updating, and packet destination selection.

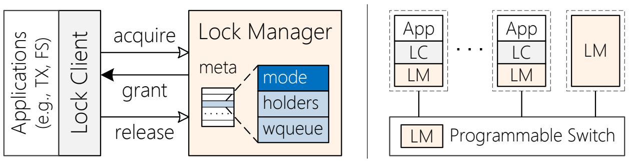


Fig. 1: *Distributed lock management.*

We implemented FissLock from scratch on a Tofino switch [16] and evaluated it using a microbenchmark, two transaction benchmarks, and a real-world application. Our experimental results show that FissLock cuts up to 79.1% (from 43.0%) of median lock grant time in the microbenchmark and improves the transaction throughput of TATP [12] and TPC-C [14] by 1.76× and 2.28×, respectively, compared to the state-of-the-art switch-based approach (NetLock [76]). We built a Redis-backed mobile banking application [11] with FissLock, which is one order of magnitude faster than Redis's official implementation (RedLock [5]).

Contributions. We summarize our contributions as follows:

- An in-depth analysis of performance issues in existing lock manager designs for modern distributed in-memory systems (§2).
- A new centralized lock management scheme, *lock fission*, which decouples grant decision and participant maintenance to embrace the best of both programmable switches and servers (§3).
- A switch-centric lock manager design that enables the lock fission protocol (§4) and implements grant decision for millions of locks on the programmable switch (§5).
- A prototype implementation and evaluation that demonstrates the efficacy of FissLock over state-of-the-art (§7).

2 Background

2.1 Distributed Lock Management

The distributed lock service commonly uses a centralized lock manager (LM) to handle all requests and grant the lock for various applications, such as transactions and file systems. As shown in the left part of Fig. 1, before reading or updating the protected data, applications—through the lock client (LC)—*acquire* the lock in shared or exclusive mode by sending the request to the lock manager, and waits until the lock manager *grants* the lock. After accessing the data, applications *release* the lock to the lock manager asynchronously. The lock manager maintains the metadata of locks (*meta*) identified by a unique lock ID. The metadata of each lock contains a mode of lock (*mode*), a set of lock holders (*holders*), and a queue for waiters (*wqueue*) [13]. The mode is a small, fixed-size flag (2 bits) that represents the current lock state (i.e., free, exclusive, or shared) and decides the grant of locks. Both holders and wqueue require large, variable-size data structures (up to hundreds of bytes) that represents the current lock participants and enables flexible locking policies (e.g., priority and fairness).

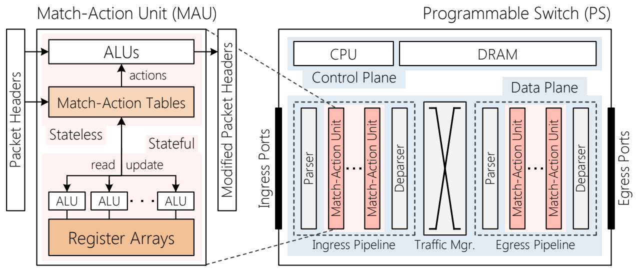


Fig. 2: *The architecture of programmable switches (PS) and the internal structure of match-action units (MAU).*

The lock manager can run on dedicated servers to avoid interference with applications [54, 55, 65] (called SrvLock). However, it may not scale well with fast-growing application workloads and datasets. Therefore, the lock manager can also be partitioned and co-located with applications to further exploit locality [6, 13, 30] (called ParLock), but it may suffer from load imbalance under skewed workloads. Recently, the programmable switch has been proposed to handle skewed workloads by managing a small fraction of hot locks (e.g., less than 10,000) directly on the switch (i.e., NetLock [76]), because it provides higher throughput than servers, halves the network latency, and saves server resources. However, due to the limited memory resources of programmable switches (a few MB), just a part of the workload can be accelerated, and the rest will be downgraded to server-based solutions. The right part of Fig. 1 illustrates the above three solutions for distributed lock management.

2.2 Programmable Switch (PSwitch)

Programmability is becoming a trend in modern network switch design, with support from major manufacturers like Cisco [3], Broadcom [33], and Intel [16]. Compared with commodity servers, programmable switches possess orders of magnitudes higher throughput (several billion packets per second) and lower delay (less than 1 µs) for packet processing [34]. Yet, only very limited memory resources (a few MB) are available. As shown in Fig. 2, modern programmable switches have a general-purpose CPU with DRAM (i.e., the *control plane*) attached to the switch ASIC (i.e., the *data plane*) via a PCIe bus. The control plane hosts a Linux-based operating system that manipulates the switch ASIC as a device. The data plane is programmable via P4 Language [19], which describes the logic of packet parsing, processing, and forwarding. Specifically, packet processing is realized as pipelines of *match-action units* (MAUs), which perform pre-defined packet modifications (*actions*) according to the value of specific fields in the packet header. Incoming packets can be either forwarded to a single egress port (*unicast*) or replicated to multiple egress ports (*multicast*). Each MAU reads and updates user-defined data stored in *register arrays* (hundreds of KB) once for each packet, and conditionally modifies packet header fields according to *match-action tables* (up to over 1 MB).

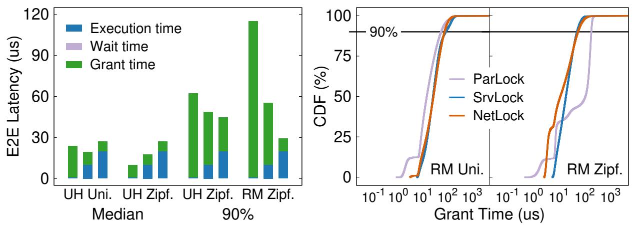


Fig. 3: *The end-to-end median and 90^{th} percentile latency breakdown with different task execution times (1 µs, 10 µs, and 20 µs) under update-heavy (UH) and read-mostly (RM) workloads (left), and the lock grant time distribution under read-mostly Uniform (Uni.) and Zipfian (Zipf.) workloads when using different LMs (right).* **Testbed:** *An 8-node cluster with an Intel Tofino switch.* **Workload:** *A microbenchmark with one million locks (see §7.1 for details).*

2.3 Performance Analysis

Grant time becomes increasingly important. The end-to-end latency of typical distributed tasks is mainly composed of execution time and lock acquire/release time. Lock release time is irrelevant as locks are typically released asynchronously. Lock acquire time consists of two parts—*wait time* and *grant time*, which denotes the duration that the request is suspended in wqueue and the rest, respectively. Nowadays, microsecond-scale execution time becomes common in distributed in-memory systems [18, 51, 60, 66, 71–73, 78]. Meanwhile, the rapidly increasing size of datasets (e.g., millions of objects per thread [29, 66]) dramatically enlarges the lock space, resulting in lower lock contention rate and less wait time. Consequently, grant time becomes non-negligible in the end-to-end task latency. As shown in Fig. 3 (left), grant time accounts for a significant portion of the median and tail end-to-end task latency under workloads with varied read-write ratio and skewness, while wait time is negligible because of the low lock contention.

Despite its importance, grant time has not drawn enough attention in existing lock manager designs, in which we observe three major performance issues (see Fig. 3 (right)).

Issue#1: Unstable latency. All existing approaches rely on server CPUs to process (partial or all) lock requests, introducing significant queueing delay that makes the grant time unstable. Since the server receives and processes packets in batches, the handling latency of requests is proportional to their positions in the batch. Due to the limited packet processing power of server CPUs, this results in non-trivial grant time variance, from a few µs to over 100 µs in our testbed. Note that requests handled by the switch do not exhibit apparent queueing delay because of line-rate processing.

Issue#2: Limited acceleration. Both ParLock and NetLock adopt fast-path request handling to accelerate a part of lock requests, but the portion of accelerated requests is rather limited. ParLock partitions the locks to handle some lock requests locally, which saves 1 RT as compared with SrvLock. However, the portion of locally handled requests is inversely proportional to the cluster size (e.g., 12.5% in our 8-node setup). NetLock manages hot locks on the programmable switch, halving the required round trips and eliminating queueing delay. However, due to limited switch memory capacity, the switch can only manage thousands of locks, which are insufficient for protecting million-scale datasets without exhibiting significant contention (see §7.7). When managing 1 million locks, even with workload profiling in advance, NetLock only accelerates 1% and 27% of grants under the Uniform and Zipfian workloads, respectively, as most requests are handled by the lock server instead of the switch.

Issue#3: Workload sensitivity. The performance of ParLock and NetLock are sensitive to workload attributes, which results in their dependence on prior knowledge of the workload. ParLock partitions locks statically, which incurs severe load imbalance problem under skewed workloads. In our experiment on the Zipfian read-mostly workload, 56.7% of requests are processed by one server, which throttles the LM and results in extremely high grant time. Due to the limited switch capacity, NetLock prefers skewed workloads and heavily relies on workload profiling for detecting hot locks. It falls back to SrvLock on occasions that the workload is uniform or has dynamic patterns (e.g., e-commerce).

3 Approach and Overview

System model and design goals. FISSLOCK is a distributed lock management system that uses programmable switches to accelerate the processing of millions of locks across diverse workloads. It is designed for distributed in-memory systems that rely on a centralized lock service to coordinate concurrent access of microsecond-scale tasks to large-scale shared datasets. Unlike lock-based coordination services like Zookeeper [32] and Chubby [21], which aim for reliable but coarse-grained coordination, FISSLOCK is not designed to achieve high availability. Instead, FISSLOCK has three high-level design goals:

- **Efficiency**: Grant locks in single-digit microseconds to meet the common needs of microsecond-scale tasks.
- **Pervasiveness**: Unleash full-scale acceleration for million-scale locks, making it feasible for large-scale systems.
- **Robustness**: Ensure good yet stable performance for diverse or dynamic workloads without prior knowledge.

Key insight. The lock management can be divided into two phases: grant *decision* and participant *maintenance*. The decision phase determines whether the lock can be granted with regard to the current lock mode, while the maintenance phase manages lock participants (i.e., holders and waiters) accordingly. We recognize two key insights that motivate the split design of a centralized lock manager. First, decision making must be *synchronous* (i.e., executed before handling other requests) to ensure the correctness of lock semantics, while participant maintenance can be *asynchronous* to shorten the critical path of granting a lock. Second, decision making relies solely on the *fixed-size* metadata (shared or exclusive), while participant maintenance also requires the remaining *variable-size* metadata (holders and waiters).

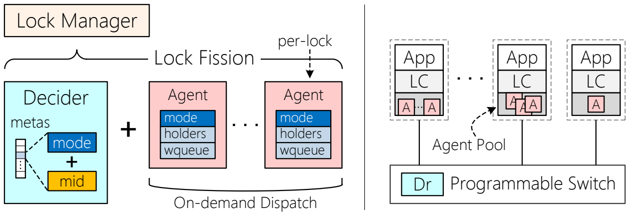


Fig. 4: *The lock fission scheme (left) and the system architecture of* FISSLOCK *(right).*

Our approach. We propose *lock fission* as a technique that decouples decision and maintenance in terms of both functionality and metadata. Specifically, the lock manager is split into a centralized *decider* and multiple per-lock *agents* (see Fig. 4 (left)). The decider records each lock's mode and makes granting decisions for lock requests accordingly, while the agent stores and maintains the remaining lock metadata (e.g., holders and waiters) of the corresponding lock. The lock acquisition request is first sent to the decider, which makes decisions and replies instantly if the lock is granted. Simultaneously, the decider forwards the request to the responsible lock agent, which updates holders and waiters according to the decision. Release requests are also sent to the decider and forwarded to the agent. When the last holder releases, the agent grants the lock to the next holder or frees the lock. In both cases, the decider is notified in advance to update the lock mode.

Lock fission provides the opportunity to accustom workload attributes without prior knowledge. By recording the resident machine ID (*mid*) of each lock's agent, the decider is always able to route requests to the latest location of agents, which supports the dynamic migration of lock agents among machines. To balance the request handling load among machines, agents are *on-demand dispatched* to the lock holder's machine. When the lock is freed or transferred to waiters, the agent is deconstructed or migrated to the waiter's machine. The migration of agents also enables most release requests to be processed by local agents, thereby shortening the network path of lock release operations.

Lock fission meets the design goals by exploiting the *packet processing strengths* of programmable switches and the *high memory capacity* of commodity servers simultaneously. First, metadata for decision is small, fixed-size data, which enables decision making for million-scale locks with limited switch memory capacity. Second, since maintenance is decoupled with decision, servers are not involved in the lock granting critical path, which eliminates the queueing delay. Third, lock fission does not require any prior knowledge of the workload and resolves the load imbalance problem by dynamic lock agent migration.

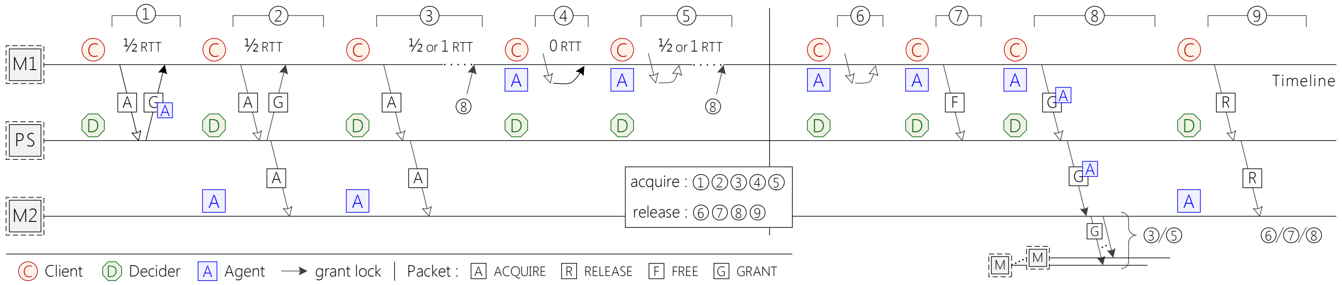


Fig. 5: *The lock acquisition and release workflow in the lock fission protocol.*

System overview. FISSLOCK is a switch-centric lock management system that applies lock fission to enable efficient and centralized management of million-scale locks. As shown in Fig. 4 (right), FISSLOCK is composed of lock clients (LC), a decider (Dr), and lock agents (A). Lock clients are libraries that encapsulate lock acquire/release requesting functionalities into APIs. The decider resides on the switch for accelerating lock grants and routing requests to agents. Each machine owns an agent pool that manages all agents on it. Lock requests forwarded from the decider are received by the agent pool, which finds the responsible agent for each request and hands over the request. The agent subsequently updates the lock metadata.

Applications acquire locks via the LC, which sends the request to the decider, or if the lock agent can be found locally, the lock agent. The decider makes decision and replies instantly, *multicasting* the request to the lock agent's resident machine at the same time. Packets arriving at machines are dispatched to the lock client (grant replies) and the agent pool (lock requests), which wakes up application tasks and calls agent functions to update the lock metadata respectively. When the lock is released, similar to the acquire case, the request is forwarded to or directly handed over to the lock agent. If the lock needs to be granted to the next holder, the agent is transferred along with the lock ownership.

4 FISSLOCK

This section describes the lock fission protocol implemented by FISSLOCK. Although the design principles of lock fission are independent of specific lock mechanisms used, we elaborate on the read-preferring design as an example, and briefly discuss the write-preferring design in §6.

4.1 Lock Operation Workflow

Fig. 5 illustrates the nine possible workflows (①–⑨) for acquiring and releasing locks in the lock fission protocol. The branches executed in each workflow are marked on the pseudocode presented in Fig. 6, Fig. 7, and Fig. 8.

```
# lid|mid|tid: lock|machine|task ID
# mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}

sslock_acquire(lid, mid, tid, mode)                         ①②③④⑤
1    if agents.find(lid) then    # local agent (rare)        ④⑤
2      | if acquire(lid, mid, tid, mode) then
3      |   | return    # local grant                           ④
4    else    # remote agent                                  ①②③
5      | net_send(ACQUIRE, {lid, mid, tid, mode})
     # wait for grant
6    pkt = net_recv(GRANT, lid, mid, tid)                   ①②③⑤
7    if pkt.agent != nil then    # add agent
8      | grant(lid, mid, tid, mode, pkt.agent)                 ①③

sslock_release(lid, mid, tid)                               ⑥⑦⑧⑨
9    if agents.find(lid) then    # local agent               ⑥⑦⑧
10     | release(lid, mid, tid)
11   else    # remote agent (rare)                             ⑨
12     | net_send(RELEASE, {lid, mid, tid})
```

Fig. 6: *Pseudocode of* FISSLOCK *client lib implementation.*

Lock acquisition workflow (①–⑤). To acquire a lock, the lock client (C) sends an ACQUIRE request to the lock decider (D) (Line 5 in Fig. 6). The decider makes a lock granting decision by examining the mode of the lock (*meta.mode*) and the requested mode in the packet (*pkt.mode*). If the lock is free (①), the decider will update the lock mode and immediately grant a lock with an empty agent (A) to the client by returning a GRANT packet (Lines 2–6 in Fig. 8). After receiving the packet, the client calls the grant function of the agent pool to initialize the agent and add it to the pool (Lines 17–19 in Fig. 7). If the lock is being held and both modes are SHARED (②), the decider will still immediately grant the lock to the client and multicasts the ACQUIRE request to the agent (Lines 8–11 in Fig. 8). The agent will add the requester to holders later (Lines 2–4 in Fig. 7). Finally, if the lock cannot be granted immediately (③), the decider will forward the request to the agent, and the agent will append the requester to the wait queue (Line 6 in Fig. 7). In rare cases (④ and ⑤), the agent is on the same machine since another client on the machine is hosting the lock, such that the client will locally acquire the lock by calling the acquire function of the agent pool (Line 2 in Fig. 6). The agent can make a decision by itself—to grant (④) or to wait (⑤)—without consulting the decider. This is because the agent always has the latest lock mode and does not need to update the decider.

Lock release workflow (⑥–⑨). In the lock fission protocol, the agent (A) is always located on the same machine as the current holder (e.g., the first holder of the shared lock).

```
# agent:
#   mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}
#   holders: a set of lock holders {mid, tid}
#   wqueue: a queue of waiters {mid, tid, mode}

acquire(lid, mid, tid, mode)                            ②③④⑤
1   agent = agents[lid]
2   if mode == SHARED && agent.mode == SHARED then      ②④
3     agent.holders.add({mid, tid})
4     return TRUE   # grant lock
5   else                                                ③⑤
6     agent.wqueue.append({mid, tid, mode})
7     return FALSE   # wait for grant

release(lid, mid, tid)                                  ⑥⑦⑧
8   agent = agents[lid]
9   agent.holders.remove({mid, tid})
10  if agent.holders.empty() then                       ⑦⑧
11    agents.remove(lid)   # remove agent
12    if agent.wqueue.empty() then                      ⑦
13      net_send(FREE, {lid})   # free agent
14    else   # transfer agent and grant lock            ⑧
15      next = agent.wqueue.pop()
16      net_send(GRANT, {lid, next.mid, next.tid,
                         next.mode, agent})

grant(lid, mid, tid, mode, agent)                       ①③
17  agent.mode = mode
18  agent.holders.add({mid, tid})   # grant lock
19  agents.add(lid, agent)   # add agent
20  if agent.mode == SHARED then   # grant others
21    ..   # pop shared waiters and add to holders
22    ..   # send grant lock (w/o agent) to them
```

Fig. 7: *Pseudocode of* FISSLOCK *agent pool implementation.*

```
# meta:
#   mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}
#   mid: machine ID

process_acquire(pkt): # pkt: {lid, mid, tid, mode}  ①②③
1   meta = metas[pkt.lid]
2   if meta.mode == FREE then   # lock is free           ①
3     meta = {pkt.mode, pkt.mid}   # alloc agent
      # grant lock and assign (empty) agent
4     agent = {pkt.mode, {}, {}}
5     grant_pkt = pkt.append(agent)
6     forward_packet_to(GRANT, pkt.mid, grant_pkt)
7     return
8   if meta.mode == SHARED && pkt.mode == SHARED then    ②
      # grant lock
9     grant_pkt = pkt.append(nil)
10    forward_packet_to(GRANT, pkt.mid, grant_pkt)
    # acquire lock on agent
11  forward_packet_to(ACQUIRE, meta.mid, pkt)            ②③

process_release(pkt): # pkt: {lid, mid, tid}            ⑨
12  meta = metas[pkt.lid]
    # release lock on agent
13  forward_packet_to(RELEASE, meta.mid, pkt)

process_free(pkt): # pkt: {lid}                          ⑦
14  metas[pkt.lid] = {FREE, nil}   # free agent
15  drop_packet(pkt)

process_grant(pkt): # pkt: {lid, mid, tid, mode, agent}⑧
16  meta = metas[pkt.lid]
17  if pkt.agent != nil then   # transfer agent
18    meta = {pkt.mode, pkt.mid}
    # grant lock and assign agent
19  forward_packet_to(GRANT, pkt.mid, pkt)
```

Fig. 8: *Pseudocode of* FISSLOCK *decider implementation.*

Therefore, when releasing a lock, its agent is highly probable to be local to the requester. The client requests the local agent to release a lock through calling the `release` function of the agent pool (Lines 9–10 in Fig. 6). If the lock is also held by other clients of the machine (⑥), the release completes immediately (Line 9 in Fig. 7). If there is no waiter (⑦), the agent pool will remove the local agent and send a `FREE` request to the decider (Lines 11–13 in Fig. 7). The decider will free the lock and drop the packet directly (Lines 14–15 in Fig. 8). If there are waiters (⑧), the lock with its agent will be transferred to the next holder, popping from the wait queue, by sending a `GRANT` packet to the decider (Lines 15–16 in Fig. 7). The decider updates the lock metadata and forwards the packet to the machine of the next holder (Lines 17–19 in Fig. 8). After receiving the packet, the client calls the `grant` function of the agent pool to maintain the agent and add it to the pool (Lines 17–19 in Fig. 7). Furthermore, if the lock could be shared with subsequent waiters, the client will pop them from the wait queue and add to the holders, sending a `GRANT` packet to each of them (Lines 20–22 in Fig. 7). If the client, without a local agent (e.g., one holder of a shared lock), releases the lock (⑨), it will send a `RELEASE` request to the decider (Line 12 in Fig. 6). The decider will then forward the request to the agent (Line 13 in Fig. 8), and the agent will call the `release` function (Lines 8–16 in Fig. 7) to release the lock, as in the above workflows (⑥–⑧).

4.2 Network Exceptions

The lock fission protocol splits the lock manager into a stationary decider on the switch and migratable agents for each lock on the servers. Since they are connected via the network, network exceptions including lost, out-of-order, and delayed packets may cause some anomalies. FISSLOCK addresses these anomalies by retransmission, rerouting, and incarnation checks, respectively. As requests for different locks do not interfere with each other, we only consider out-of-order and delayed packets pertaining to the same lock.

Lost packets. FISSLOCK uses different approaches to handle the loss of packets initially sent by the switch and servers.

Server-initiated packets. FISSLOCK addresses the loss of packets initially sent by servers through TCP-based retransmission. When a server receives a packet, it sends an acknowledgement (ACK) to the origin of the packet. The switch forwards ACKs without updating on-switch metadata. Specifically, the destination of `FREE` packets (⑦) is the switch instead of servers, in which case the switch sends the packet back as an ACK. Servers monitor the arrival of ACKs for each packet regularly and retransmit packets that are not ACKed within a certain time frame.

In cases where the ACK is lost or delayed, the retransmis-

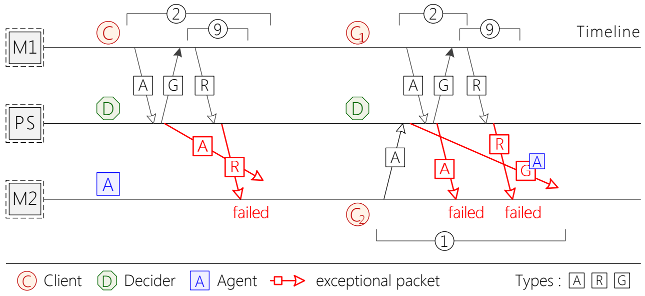


Fig. 9: *Two anomalies in the protocol due to out-of-order packets.*

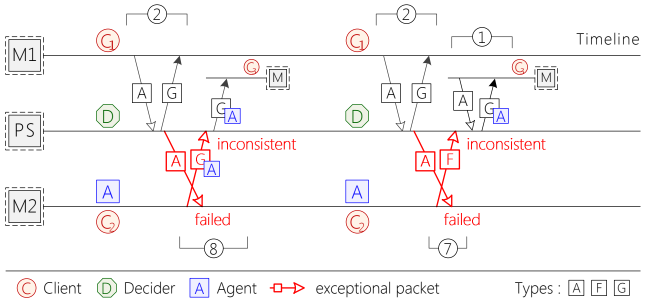


Fig. 10: *Two anomalies in the protocol due to delayed packets.*

sion mechanism can cause packet duplication, which is handled by servers through TCP. To prevent duplicated packets from corrupting on-switch metadata, the switch maintains a sequence number for each server to signify the number of processed packets. Servers track the number of packets sent to the switch, excluding retransmitted ones, and embed this number into all packets. If the incoming packet's number is not larger than the on-switch number, indicating that the packet is a duplicate, the switch does not update the lock metadata when processing the packet.

Switch-initiated packets. In workflows ① and ②, the lock request is granted by the switch instead of servers. Therefore, the GRANT packets in these workflows are considered switch-initiated. To avoid the complexities of monitoring ACK arrivals and executing retransmissions on the switch, FISSLOCK addresses the loss of switch-initiated packets in an alternative way—by setting a fixed timeout for lock acquisition operations. If an acquisition operation times out due to the absence of the GRANT packet, the client releases the lock (⑨) and retries the acquisition operation later. When processing the RELEASE packet, the switch frees the lock and drops the packet if it originates from the agent's server (①). If the RELEASE packet arrives at the agent, the agent removes the client from holders (②) or, if the client is not in holders (③), the wait queue.

Out-of-order packets. Packets with dependencies arriving out of order may lead to two anomalies. If ACQUIRE and RELEASE requests from the same requester are reordered (Fig. 9 (left)), the agent pool will fail to process the RELEASE request because the requester is not yet the holder or waiter of the lock. Further, the lock will never be released after granting it to the requester. If the ACQUIRE (and RELEASE) packet arrives before the agent is granted (Fig. 9 (right)), the agent pool will fail to process the ACQUIRE (and RELEASE) request because the agent does not exist. To avoid extra on-switch design, FISSLOCK resolves these anomalies by simply sending failed requests back to the decider. The decider then routes the request to the agent again, correcting the order.

Delayed packets. The ACQUIRE packet, which is immediately granted by the decider (②), may lead to anomalies when it arrives after the agent has been transferred (⑧) or freed (⑦). Note that the lock mode and the ACQUIRE packet must be both shared, as the lock is immediately granted. As shown in Fig. 10, the decider will incorrectly transfer the lock with its agent to the next holder (⑧) or grant the freed lock to a new client (①). The agent pool will fail to process the delayed packet.

FISSLOCK uses incarnation checks [68] to detect anomalies on the decider. Both the decider and the agent have a per-lock *incarnation* that is initially zero and is incremented when receiving a shared ACQUIRE request (②). The agent pool will include the expected incarnation in the GRANT and FREE packets. When the decider receives these packets and the lock is shared, it will check if its own incarnation matches the incarnation in the packet. If they match, the decider resets the incarnation and handles the request as normal. Otherwise, it refuses the packets, and the agent pool restores the agent to continue handling lock requests. Moreover, the failed ACQUIRE request will also be sent back to the decider, which will then route the request to the agent again.

4.3 Protocol Correctness

The lock fission protocol introduces two changes to the traditional reader-writer lock design: decoupling the decision process from the lock manager and allowing lock agents to migrate among servers. We argue the correctness of our lock fission protocol by showing that these changes preserve the reader-writer property [24, 53]. Specifically, we prove that the following two invariants always hold.

Invariant 1 (reader-writer exclusion): *Locks held in exclusive mode do not have any other holders; locks held in shared mode do not have any exclusive holders.*

In the traditional reader-writer lock design, all acquisition requests are processed by the lock manager, which maintains Invariant 1. In the lock fission protocol, requests are either processed by the local agent or sent to the decider. Both of them grant or suspend lock requests following the same criteria as traditional lock managers. Hence, Invariant 1 holds as long as the local agent and the decider always have consistent lock mode, and the lock mode correctly reflects the number of holders, which we prove as follows.

Lemma 1: *If the local agent exists, it has the same lock mode as the decider.*

Proof: The lock mode in the decider becomes shared or exclusive when a free lock is acquired for the first time (①) or the lock is transferred to a shared or exclusive waiter (⑧). In both cases, the agent is carried by the `GRANT` packet, so the local agent does not exist until the `GRANT` packet is received by the requester's machine. Moreover, the lock mode in the carried agent, which subsequently becomes the local agent, is identical to the updated lock mode in the decider. Before the local agent is removed, the decider only receives `ACQUIRE` and `RELEASE` packets, which do not change the lock mode. Hence, the lock mode in the decider remains consistent with the lock mode in the local agent.

Lemma 2: *The lock mode in the decider always correctly reflects the number of holders.*

Proof: We enumerate all possible lock mode transitions to prove Lemma 2. Initially, each lock is free and has no holders. The lock mode transits from free to shared or exclusive only when the lock is granted to a requester, which guarantees that the lock has at least one holder. If the lock mode is exclusive, subsequent `ACQUIRE` packets will not be granted. Hence, exclusive locks have at most one holder. The lock mode transits back to free only when receiving `FREE` packets whose incarnation matches the decider's incarnation, which indicates that all holders have released the lock. Similarly, the lock mode transits between shared and exclusive when receiving `GRANT` packets that have matched incarnation. In this case, all former holders have released the lock, and the destination of the `GRANT` packet becomes the new holder.

Invariant 2 (finite wait): *Waiters of the lock will be granted in finite time.*

Traditional lock managers decide to suspend a lock request and add the requester to the wait queue simultaneously. However, in the lock fission protocol, these two operations are decoupled and executed by different entities, i.e., the decider and the agent. We show that Invariant 2 still holds in the decoupled setup, which allows the agent to migrate among servers, by proving Lemma 3 and 4.

Lemma 3: *Lock acquisition requests that are suspended will eventually be added to the wait queue.*

Proof: Lock acquisition requests may be suspended by the agent locally (⑤) or remotely (③). In the former case (⑤), the local agent directly adds the requester to the wait queue when suspending the request. In the latter case (③), the decider, after deciding not to grant the lock, forwards the `ACQUIRE` request to the agent, which adds it to the wait queue. If the agent is not present due to packet reordering, the request is sent back to the decider. The decider then forwards it to the agent's latest location, guaranteeing that it will eventually be recorded in the agent's wait queue.

Lemma 4: *Waiters recorded in the wait queue will eventually be granted.*

Proof: All holders of a lock will eventually release it, either locally (⑥) or remotely (⑨). Therefore, the number of holders will eventually become zero, unless the lock is continuously granted to new shared holders (②), which FissLock averts by adopting a starvation prevention mechanism (see §6). When the number of holders reaches zero, at least one waiter is granted and becomes the new holder (⑧). Thus, according to induction, all waiters will eventually be granted.

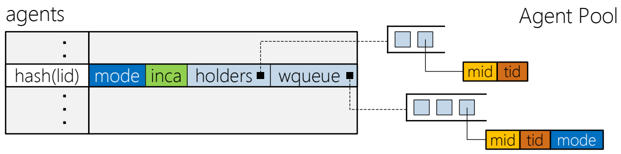


Fig. 11: *The main structures in the agent pool.*

5 Design

5.1 On-server Agent Pool

Data structures. As shown in Fig. 11, the agent pool uses a hash table to store granted agents, which is shared by all clients on the same machine. Each agent maintains complete lock metadata, including mode, incarnation, holders, and wqueue. The holders is an unordered set of current lock holders (*mid*, *tid*), and the wqueue is a FIFO queue of pending lock requests (*mid*, *tid*, *mode*). The incarnation (*inca*) is used to handle delayed packets.

Agent operations. The agent pool adds an agent when receiving a `GRANT` packet with agent information (Lines 17–19 in Fig. 7) and removes an agent when the lock is freed or granted to the next holder (Line 11 in Fig. 7). Furthermore, if the agent pool fails to process packets due to network exceptions (see §4.2), it sends such packets back to the decider. For incarnation checking, the agent pool includes the expected incarnation in the `GRANT` and `FREE` packets and restores the agent if the decider refuses these packets.

5.2 On-Switch Lock Decider

Data structures. The metadata of all locks (*metas*) is stored in register arrays of MAUs (RA in Fig. 12). When lock packets pass through the MAU, predefined actions read and update lock metadata via ALUs attached to the RA. To guarantee line-rate packet processing, each RA can only be accessed once per packet, and the ALU is allowed to perform a few simple arithmetic operations. Therefore, the lock decider functionality and metadata must be split into multiple MAUs, creating a pipeline. Each MAU stores a piece of metadata and performs the corresponding logic. All RAs are indexed by the lock ID (*lid*).

FissLock carefully selects the register size of RA for minimal memory consumption. The allowed register size includes 1 bit and a specific amount of bytes (1, 2, 4, or 8). An intuitive design is to store all of the lock *mode*, machine ID (*mid*), and incarnation (*inca*) into 1-byte RAs, while it wastes 6 bits for each lock mode. Instead, FissLock stores the lock mode with two 1-bit RAs (*free RA* and *r/w RA* in

Table 2: *Packet types used in* FISSLOCK.

Type	Contents
ACQUIRE	lock ID and mode to be acquired, machine ID and task ID of the requester
RELEASE	lock ID to be released, machine ID and task ID of the requester
FREE	lock ID to be freed, INCA, and lock mode before free
GRANT	lock ID to be granted to machine ID and task ID, INCA, and agent of the lock (optional)

Fig. 12), indicating whether the lock is held and whether the lock is shared, respectively. Both *inca* and *mid* use 1-byte RAs (*inca RA* and *mid RA* in Fig. 12). Although 8 bits are still over-sufficient, there are unfortunately no smaller register units in the RA to store them. Packing multiple *mid*s (or *inca*s) in one register is not feasible due to memory accessing restrictions. In summary, FISSLOCK compresses the switch memory consumption of each lock to 18 bits, which is two orders of magnitudes smaller than prior work [76].

Using 1-byte registers for *mid* and *inca*, a single MAU is not enough for the million-scale lock amount. Therefore, FISSLOCK splits them into multiple MAUs, each storing a fixed range of locks. For these MAUs, the lock ID is translated into a MAU *index* and an *offset* within the MAU (*MAU-selection* stage in Fig. 12).

Packet processing pipeline. The decider is realized as a 6-stage pipeline that processes four types of packets in FISSLOCK (see Table 2), where all packets share the same header format but use different header fields. Each stage checks the metadata in the packet or loaded from former stages for selecting proper actions to execute. Actions leverage the ALUs attached to RAs to read, update, and write back the metadata simultaneously. FISSLOCK organizes the MAU order to ensure that all metadata is loaded from RAs before being used by subsequent stages. Only packet types marked in Fig. 12 are processed in corresponding stages. The logic of each stage is described as follows.[1]

MAU-selection stage translates the lock ID into the MAU *index* and the *offset* in the MAU in Fig. 12 for Check stage and MID stage. In Check stage and MID stage, MAUs other than the indexed MAU are skipped.

Check stage checks and updates the incarnation. `ACQUIRE` packets for a shared lock increment the incarnation by 1. `GRANT` packets with agent and `FREE` packets reset the incarnation if the current lock mode is exclusive[2] or the incarnation in the packet is matched, i.e., there are no delayed packets. Otherwise, the packet is marked as invalid and will be skipped by the rest stages except for Destination stage.

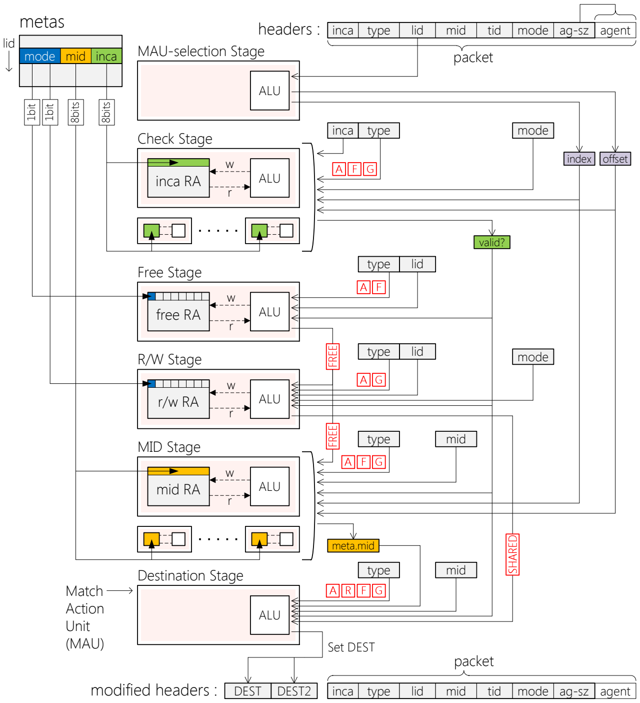


Fig. 12: *The metadata of locks stored in register array of MAUs and packet processing pipeline (data flow) in programmable switch.*

Free stage updates the free register in RA and loads its original value to the FREE flag in Fig. 12. `ACQUIRE` and `FREE` packets set the free register to 0 (held) and 1 (free) despite its original value, respectively, as 1-bit RAs do not support conditional updates with regard to the original register value.

R/W stage updates the r/w register in RA. `GRANT` packets update the r/w register to the next holder's mode. `ACQUIRE` packets only update the r/w register to the requester's mode when the lock is free (Line 3 in Fig. 8), i.e., the FREE flag is 1. This stage sets the SHARED flag in Fig. 12 that indicates whether both the *pkt.mode* and the r/w register are `SHARED`, which determines packet destinations later.

MID stage loads and updates *meta.mid* in mid RA.`ACQUIRE` and `GRANT` packets store the *pkt.mid* into mid RA as the new agent's location when the lock is free (Line 3 in Fig. 8) and when carrying the agent (Lines 17–18 in Fig. 8), respectively. `FREE` packets reset *meta.mid* to 0 (Line 14 in Fig. 8).

Destination stage routes packets to correct egress ports. The destination is controlled by two fields defined by the switch (*DEST* and *DEST2* in Fig. 12), which specify the ports that the packet should be replicated and forwarded to. Both fields can be null for dropping the corresponding packet replica. Packets specify their destination in DEST and the multicasted packet's destination in DEST2. Programmed egress pipelines update the type of multicasted packets to `GRANT`.

`ACQUIRE` packets are forwarded to the *pkt.mid* when the lock is free (Line 6 in Fig. 8), *meta.mid* when the request is suspended (Line 11 in Fig. 8), and both *mids* when a shared lock is granted to a shared requester (Lines 10–11 in Fig. 8).

[1] Stages for identifying lock packets and supporting retransmissions are not included for convenience.

[2] The mode field of packet header is used to transfer the current lock mode.

RELEASE packets are always forwarded to *meta.mid* (Line 13 in Fig. 8). GRANT and FREE packets are forwarded to *pkt.mid* (Line 19 in Fig. 8) and dropped (Line 15 in Fig. 8), respectively, unless marked as invalid in Check stage, in which case they are forwarded back to *meta.mid*.

5.3 Failure Handling

Failure model. FissLock can tolerate individual and simultaneous switch and server failures, but it does not guarantee availability (without data replication). Switch and server failures are detected by a reliable external coordinator using heartbeats. On failed servers, the granted locks are considered expired, and pending lock acquire operations are considered aborted. FissLock assumes that applications will handle lock expiration and aborted operations, such as manually aborting ongoing transactions. The availability of FissLock can be achieved through existing methods that replicate the switch data plane to backup switches [39], which are orthogonal to our work.

Failure recovery. Switch and server failures in FissLock may result in the loss of network packets (see Table 2), switch states (*metas*), lock agent states (*agents*), and lock client states (granted and pending lock requests). When a failure is detected by the coordinator, FissLock will restart the failed switch if necessary and perform three steps sequentially to recover the aforementioned states.

- **Server data aggregation (S1)**. All surviving servers pause lock operations and submit lock agent states and lock client states to the coordinator.
- **Server data recovery (S2)**. All surviving servers recover the lost and inconsistent states by referring to the aggregated states from the coordinator.
- **Switch data recovery (S3)**. The switch recovers its states by referring to lock agent states from all surviving servers.

5.4 Scale to Multiple Racks

FissLock can be scaled out by partitioning locks to each ToR switch. Each switch only handles requests for the locks it manages and routes other requests to responsible racks. Each machine has a global *mid*, which is translated by the switch to an egress port for the next-hop switch or the machine. In the current implementation of FissLock, these translation rules are statically predetermined, which disables on-demand scaling. However, on-demand scaling could potentially be achieved by updating these rules through the switch control plane. Although requests for remote-rack locks may experience higher network latency, they still have a stable grant time without any queueing delay. The imbalance of loads among switches is not a significant concern, as switches have orders-of-magnitude higher packet processing speed than servers. Even under heavy loads, servers would reach saturation before a hotspot switch does.

6 Implementation

We implemented FissLock from scratch using roughly 1,200 lines of P4 code and 5,000 lines of C++ code.[3] DPDK is used for packet sending and receiving.

Non-linear lock IDs. The lock ID can be sparse and inefficient for indexing RAs. In this case, FissLock maps lock IDs to linear RA indexes with an RPC daemon. Lock clients cache the mapping and embeds the RA index in packets.

Lock scales. FissLock supports efficient management of over 1 million on-switch locks. In our experiments on TPC-C, we use them to protect billion-scale data by protecting a range of data objects with each lock (see §7.4). To support out-of-range locks that exceed the switch capacity, FissLock adopts ParLock as a fallback, i.e., these locks are handled by on-server LMs, and the switch forwards their requests to the server (see §7.6).

Read/Write preference. FissLock is implemented as read-preferring since it is common. For a write-preferring design, the switch requires an additional 1-bit state *ww* (write-waiter) to indicate the existence of exclusive waiters. When encountering an exclusive waiter, the decider sets *ww* to true, so subsequent shared requests are not granted even if all holders are shared. The server-side write-preferring implementation is identical to server-based lock managers [24, 53].

Policy support. In FissLock, the on-server lock agents are tasked with enabling various lock policies (e.g., fairness), as they determine the next holders when the current holders release the lock. For example, FissLock ensures first-come-first-served fairness by using a FIFO wait queue, which prioritizes waiters that are enqueued earlier.

Starvation prevention. In the read-preferring design, FissLock uses an additional 1-bit state per lock on the switch to prevent readers from starving writers. This state is periodically set by the agent if there exists a writer in the wait queue and is cleared when all current holders release the lock. In the write-preferring design, readers are not starved. When the lock is held by a writer, all incoming lock requests are appended to the agent's FIFO wait queue. This guarantees that writers are not granted ahead of preceding readers.

Deadlocks. FissLock offers a lock aborting mechanism to assist applications resolve deadlocks. Specifically, it sets a local timeout for each lock request and aborts pending lock requests that have not been granted after the timeout.

7 Evaluation

7.1 Experimental Setup

Testbed. The experiments were conducted on a cluster consisting of four machines, each has two 12-core Intel CPUs, 128 GB of RAM, and two ConnectX-5 100 Gbps NICs. All

[3] The source code of FissLock is available at https://github.com/SJTU-IPADS/fisslock.

Table 3: *Workload description.*

Microbenchmark			TATP			TPC-C		
Wkld	**Type**	**Ratio**	**Txn**	**Type**	**Ratio**	**Txn**	**Type**	**Ratio**
UH	R	50%	GS	R	35%	NEW	RW	45%
	W	50%	GD	R	10%	PAY	RW	43%
RM	R	90%	GA	R	35%	DLY	RW	4%
	W	10%	US	W	2%	OS	RO	4%
RO	R	100%	UL	W	14%	SL	RO	4%
	W	0%	IF	W	2%			
			DF	W	2%			

NICs are connected to a Top-of-Rack (ToR) wedge100BF-32x programmable switch, which is equipped with an Intel Tofino ASIC [16]. Each machine hosts two logical nodes, each has one CPU connected with one NIC used by all threads running on it. For each node, we assign 10 cores to application threads for issuing lock requests and receiving grant replies, and 1 core to FissLock's agent pool for maintaining lock metadata. Incoming packets are triaged utilizing DPDK Flow Director and Receive Side Scaling. To further adjust the degree of concurrency, we use coroutines in each application thread to simulate multiple clients.

Comparing targets. We compare FissLock with the state-of-the-art centralized lock manager NetLock [76] and two traditional server-based lock managers, namely SrvLock and ParLock. We re-implemented NetLock following its open-sourced artifact [9], which is not compatible with our programmable switch. The maximum number of locks that NetLock manages on the switch is determined by the scripts in its artifact. Due to the lack of open-source artifacts, we hand-crafted ParLock following the specification of popular commercial systems [6, 13], and used the lock server implementation of NetLock as SrvLock. ParLock uses the same allocation of CPU cores as FissLock, while NetLock and SrvLock use one node as the dedicated lock server and the other seven nodes as clients. For fairness, all systems are assigned the same total amount of CPU cores, i.e., 8 cores as lock managers and 80 cores as lock clients.

Workloads. We use one microbenchmark to evaluate lock granting performance, and two transaction benchmarks to study the impact on accelerating transaction execution (see Table 3). We trace all lock requests during a pre-execution phase for NetLock to profile the workload. To maintain fairness, we evaluate all lock managers using the same lock request traces and report the actual execution performance, like prior work [75, 76].

Microbenchmark. We built a microbenchmark to emulate the typical use of locks in modern distributed in-memory systems, where shared (resp. exclusive) locks are acquired before and released after data reads (resp. updates) to serialize these operations. To study the lock granting performance of lock managers under different read-write ratios and workload skewness, the microbenchmark includes three representative workloads: update-heavy (UH), read-mostly (RM), and read-only (RO). Each workload has both Uniform (Uni.) and Zipfian (Zipf.) lock request distributions.

Transaction benchmarks. TATP [12] and TPC-C [14] are evaluated to study the impact of lock managers on the end-to-end performance of distributed in-memory systems. TATP represents low-locality[4] and read-intensive (80% of transactions are read-only) workload, while TPC-C represents high-locality (∼90% of transactions only access local tables [14, 37]) and write-intensive (8% of transactions are read-only) workload. We use the default population size (100,000 subscribers) for TATP and adopt the same per-client warehouse number as prior work [29, 66] for TPC-C. We protect a range of records with each lock to control the total amount of locks in the benchmark. We run 160 clients for TATP and 1,200 clients for TPC-C, all clients issue lock requests synchronously. We adopt the two-phase locking (2PL) protocol when executing transactions. After acquiring all locks required by each transaction, we delay around 2 µs for TATP and 10 µs for TPC-C to simulate the execution time in Table 1. Transactions executed over 10 ms are aborted to avoid deadlocks.

7.2 Memory Consumption

We first study the switch memory usage of FissLock, which limits the maximum number of locks a programmable switch can host. Following the internal structure of programmable switch ASICs (see Fig. 2), two factors collectively impact the limitation: the number of MAUs and the memory capacity of each MAU. Let N be the number of locks in the system, C be the per-MAU memory capacity in bits, then the amount of MAUs required M can be described as:

$$M = \left\lceil \frac{N}{C} \right\rceil \times 2 + \left\lceil \frac{N \times 8}{C} \right\rceil \times 2 + 4$$

where 4 MAUs are occupied by the control and computing logic of the lock decider, and others denote MAUs for lock mode (*mode*), machine ID (*mid*), and incarnation (*inca*). The switch ASIC in our testbed has 12 MAUs at the ingress pipeline, each providing about 560 KB stateful storage (i.e., register array). Assuming that all MAUs are used by FissLock and following an optimal allocation scheme (i.e., 2 MAUs for *mode*, 3 MAUs for *mid*, and 3 MAUs for *inca*), the maximum number of locks that can be hosted is 1.68 million. In contrast, NetLock can only manage a few thousand locks on the same switch ASIC throughout our experiments.

7.3 Lock Granting Performance

We study the lock granting performance of FissLock and baselines through the grant time distribution (Fig. 13) and lock request throughput (Fig. 14 (left)) in the microbenchmark. All experiments use 160 clients and 1 million locks.

[4] Like prior work [29, 37], we do not improve the locality by deliberately partitioning TATP tables.

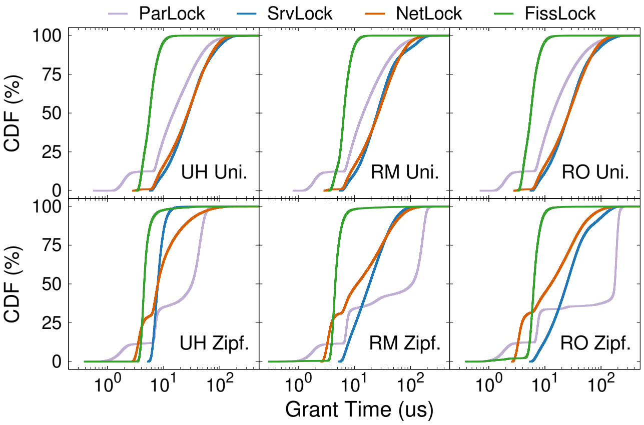


Fig. 13: *The CDF of lock grant time in the microbenchmark when using different lock managers.*

Overall, FissLock achieves low and stable grant time under all workloads. Compared with baselines, it cuts down the median grant time by up to 79.5% (SrvLock), 79.1% (NetLock), and 96.4% (ParLock), and the 90^{th} percentile grant time by up to 89.7% (SrvLock), 88.9% (NetLock), and 96.5% (ParLock). Consequently, FissLock outperforms baselines on lock request throughput by up to 4.08× (ParLock), 4.79× (SrvLock), and 4.99× (NetLock). The performance gain of FissLock mainly comes from three aspects: (1) the elimination of queueing delay by switch-based grant deciding, (2) the pervasiveness of acceleration by lock fission, and (3) the automatic load balancing by dynamic agent migration.

Uniform workloads. FissLock mainly benefits from (1) and (2) under Uniform workloads. The queueing delay dominates the grant time of all three baseline systems, which grows to up to 84.5 µs (SrvLock), 76.2 µs (NetLock), and 54.8 µs (ParLock) at 90^{th} percentile. Oppositely, FissLock controls the 90^{th} percentile grant time of all workloads under 9.42µs by eliminating queueing delay. NetLock falls back to SrvLock under Uniform workloads because only ∼1% of requests are handled by the switch, while FissLock accelerates all requests. ParLock handles around 12.5% of requests locally, which alleviates the remote LM's load and helps it slightly outperform SrvLock and NetLock.

Zipfian workloads. FissLock benefits from all three aspects under Zipfian workloads. Even with workload profiling, NetLock accelerates merely ∼27% of requests under Zipfian workloads. Although these requests have slightly (0–2µs) lower grant time than FissLock because FissLock devotes additional time to skim through the agent pool, the other 73% of requests are still handled by the server and have similar performance to SrvLock. Oppositely, FissLock makes grant decisions on the switch for all lock requests. ParLock suffers from severe load imbalance under Zipfian workloads, which significantly increases its grant time. Meanwhile, FissLock balances the load among servers and minimizes the impact of workload skewness, achieving up to 96.4% lower grant time than ParLock. The R/W lock contention becomes the main restriction of FissLock's throughput under Zipfian workloads and significantly reduces the lead of FissLock.

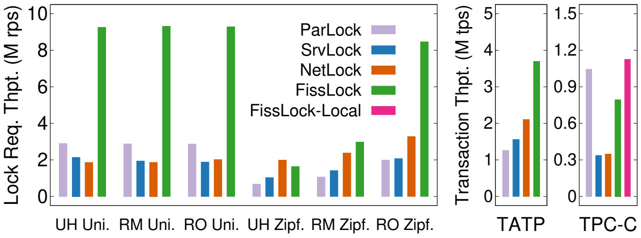


Fig. 14: *The lock request throughput in the microbenchmark, and the transaction throughput on TATP and TPC-C with different lock managers.*

7.4 Distributed Transaction Performance

We study the impact of lock managers on the end-to-end latency (Fig. 15) and throughput (Fig. 14 (right)) of transaction execution with TATP and TPC-C benchmarks. We show the latency distribution of each type of transaction individually.

TATP. TATP is a read-dominated workload containing 7 short transactions. Overall, FissLock outperforms baseline systems by 2.93× (ParLock), 2.37× (SrvLock), and 1.76× (NetLock) on transaction throughput. Since most transactions acquire only one or two locks, there were no deadlocks throughout the test. When executing transactions that only acquire one lock, FissLock exhibits similar performance to the microbenchmark case, i.e., the latency remains low for 99% of GS (17.5 µs), GA (15.4 µs), and UL (27.7 µs) transactions. For multi-lock transactions (GD, US, IF, and DF), the latency of FissLock is proportional to the average amount of locks acquired when executing the transaction. ParLock suffers from severe load imbalance when executing GS, US, UL, IF, and DF transactions, which drags down its throughput and results in lower queueing delay in GD and GA. Moreover, the effect of ParLock's local fast-path is less apparent in multi-lock transactions as the possibility of local grants is powered. NetLock has similar performance to SrvLock because the switch LM has limited acceleration proportion (17% in total), even if all rows are selected in a non-uniform manner, because the on-switch lock proportion is too restricted (0.09%). Queueing delay dominates the transaction latency of both systems.

TPC-C. TPC-C is a write-dominated workload containing 5 types of complicated transactions, where 90% of transactions solely acquire local locks. Given this workload locality information, ParLock can serve most requests with local lock servers, achieving superior performance. To show that FissLock can also benefit from this workload-aware optimization, we additionally evaluate FissLock-Local, which follows ParLock's method of co-locating locks with the clients acquiring them. Among lock managers that do not exploit

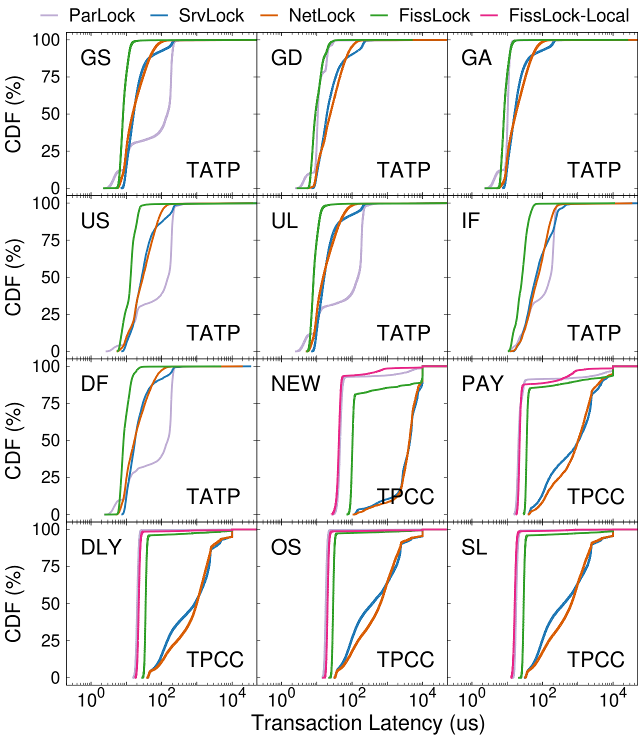


Fig. 15: *The CDF of transaction latency on TATP and TPC-C workloads when using different LMs.*

workload locality, FISSLOCK outperforms SrvLock and NetLock by 2.36× and 2.28× on transaction throughput, respectively. Both FISSLOCK-Local and ParLock handle over 90% of requests locally, where they have similar performance. However, FISSLOCK-Local still has 1.08× higher throughput than ParLock because of better remote lock requesting performance. We analyze the latency of write-intensive and read-only transactions separately as follows.

Write-intensive transactions (NEW and PAY). NEW and PAY acquire 14 and 4 locks on average. Even when acquiring almost all locks remotely, FISSLOCK still achieves fairly low and stable latency for 81% of NEW (< 125.5 µs) and 85% of PAY (< 44.5 µs) transactions. The latency of other transactions is dominated by the wait time due to lock contention. NetLock falls back to SrvLock because only 6% of requests to 0.3% of locks are handled by the switch, due to sparse data accesses in large datasets [29, 66]. Both LMs have over an order-of-magnitude higher transaction latency than FISSLOCK because of queueing delay. ParLock and FISSLOCK-Local have 40%–60% lower transaction latency than FISSLOCK due to local lock request handling. FISSLOCK-Local exhibits a shorter tail than ParLock because of higher remote lock acquire performance.

Read-only transactions (DLY, OS, and SL). All read-only transactions are local, which explains the identical performance of ParLock and FISSLOCK-Local. The latency turning point of FISSLOCK appears later (96.5%, 97.7%, and 96.2% for DLY, OS, and SL) because of lower wait time when acquiring shared locks. The performance of NetLock and SrvLock is similar to the write-intensive case as the queueing delay instead of wait time dominates the latency.

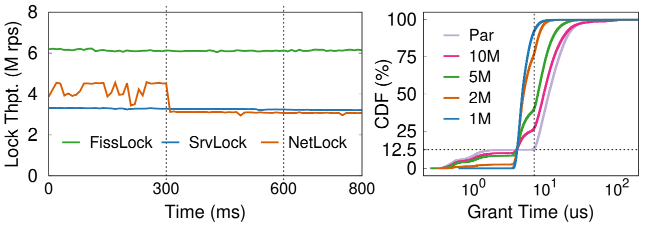


Fig. 16: *The timeline of lock request throughput on a dynamic workload (left), and the CDF of lock grant time with various lock scales when using a uniform RM workload (right).*

7.5 Dynamic Workload

To study the robustness of lock managers under dynamic workloads, we alter the hotspot of 1 million locks every 300 ms (i.e., 2,500 hot locks). In this workload, half of the requests target the hotspot, while the other half are evenly distributed to the remaining locks. As shown in Fig. 16 (left), FISSLOCK achieves a consistently high throughput of over 6 million requests per second (M rps) regardless of hotspot changes, thanks to its pervasive acceleration. In contrast, NetLock's throughput fluctuates between 3.44 M and 4.58 M rps, as the switch only handles half of the requests. When the hotspot changes at 300 ms, the throughput instantly drops to around 3 M rps, close to SrvLock (its fallback), and remains low until the hotspot returns.[5]

7.6 Lock Scales

We further evaluate the lock granting performance of FISSLOCK as the number of locks increases, using a uniform RM workload. We also report the result of ParLock, the fallback approach of FISSLOCK, with 10M locks as a reference. As shown in Fig. 16 (right), as expected, the performance of FISSLOCK gradually approaches that of ParLock as the number of locks increases. We found that FISSLOCK still achieves 39.9% lower 20^{th} percentile and 9.9% lower 90^{th} percentile grant time compared to ParLock for 10M locks by shipping the load of around 10% requests to the switch and thereby relieving server CPUs. At 0–12.5 percentiles, workloads with fewer locks perform worse because almost all requests for on-switch locks are handled by the switch, while around 12.5% of requests for out-of-range locks are handled locally.

7.7 Lock Granularity

To justify the necessity of using fine-grained locks for large-scale datasets, we conduct an experiment that varies the number of locks used to protect accesses to 10 million objects. These objects are evenly distributed among the locks. As shown in Fig. 17 (left), when using coarse-grained locks,

[5] Due to space limitations, we omit this part in Fig. 16 (left).

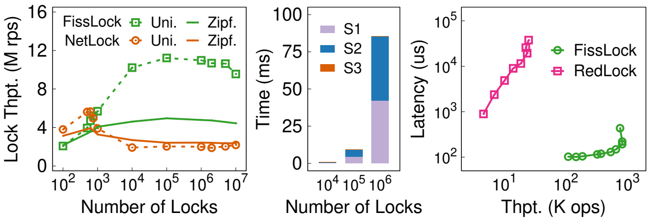


Fig. 17: *The throughput of* FissLock *and NetLock using different number of locks for a 10-million dataset (left), the recovery time breakdown for* FissLock *(mid), and the performance of mobile banking application using* FissLock *and RedLock (right).*

the lock granting throughput of FissLock is significantly dragged down (up to 5.40×) by lock contention in a uniform RM workload. NetLock achieves the peak throughput when using around 1,000 locks due to limited switch memory. However, the peak throughput is only 5.65 M rps due to severe lock contention. In contrast, FissLock unleashes full-scale acceleration with over one million fine-grained locks, thereby resulting in 1.99× (Uniform) and 1.26× (Zipfian) higher peak throughput than NetLock.

7.8 Failure Recovery

To study the performance of failure recovery, we manually inject a simultaneous failure of one switch and one server into the experiment used in §7.7. As shown in Fig. 17 (mid), the recovery time mainly comes from aggregating states of surviving servers (S1) and repairing them (S2), and is proportional to the number of locks due to scanning the metadata (e.g., granted requests) of all locks. The recovery time for switch states (S3) is trivial, as it only involves held locks.

7.9 Application: Mobile Banking

We build a mobile banking application that supports common banking operations like balance checking (BC) and funds transferring (FT) [7, 8]. The application server uses FissLock following the 2PL protocol for transactions and uses Redis-backed in-memory store [11]. It combines Redis asynchronous APIs and coroutines to hide network latency and maximize throughput when serving massive clients. We use Redis's official implementation of distributed locks (RedLock [5]) as the baseline and use a mixed workload containing 90% BC and 10% FT operations, which reflects the user behavior that checks balance much more frequent than transferring funds [11]. We initialize the bank with 1 million accounts. By adjusting the number of clients that issue operations, we compare accumulative throughput of all clients and median latency of the application using FissLock and RedLock. As shown in Fig. 17 (right), the operation throughput when using RedLock peaks at 24.8 K ops due to lock contention. Conversely, when using FissLock, the operation throughput scales to 825.4 K ops because FissLock grants and transfers locks faster. Additionally, FissLock cuts down the median latency of banking operations by at least one order of magnitude.

8 Related Work

Distributed lock management. There have been many efforts to investigate distributed lock management which are classified into two categories, centralized LM [6, 13, 21, 30, 54, 55, 65, 76] and decentralized LM [28, 56, 73, 75]. Centralized LMs are widely used because they enable rich properties such as latency predictability [31, 38, 46], starvation freedom [36], and performance isolation [76]. Decentralized LMs leverage one-sided RDMA primitives to bypass the CPU bottleneck of lock managers [28, 56, 73, 75], which offers better performance but loses support to the properties above. Prior work [76] uses programmable switches to host part of locks, achieving desired performance without sacrificing centralized properties. However, it assumes that the workload is highly skewed and predictable. Differently, FissLock can accelerate million-scale locks for diverse workloads without prior knowledge.

In-network optimization. The emergence of programmable switches [3, 16, 33] inspires numerous in-network designs for distributed systems, including distributed cache [35, 42, 47, 49, 50], consensus and concurrency control [26, 27, 34, 43, 44, 59, 76], machine learning [17, 40, 48, 63, 64, 77], task scheduling [69, 74], and distributed data coherence [41, 45, 70]. These systems primarily leverage the stronger packet processing power and shorter network round trip of switches to achieve higher performance for a portion of workloads. NetLock [76], the system most relevant to FissLock, implements a full lock manager on the switch to handle requests on hot locks. However, due to limited switch memory, the on-switch lock manager can only optimize thousands of locks. In contrast, FissLock achieves consistent performance improvement for millions of locks via lock fission.

9 Conclusion

This paper presents FissLock, a switch-centric lock service that enables lock fission scheme to provide microsecond-scale lock grant time for millions of locks. The concept of lock fission—decoupling the locking process to align with the characteristics of heterogeneous hardware—could be applied to other contexts. For instance, the locking process can be decoupled differently in various heterogeneous environments, such as disaggregated memory. We leave the investigation of these to future work.

10 Acknowledgement

We sincerely thank the OSDI reviewers and our shepherd for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62272291, 61925206, 6213201), the HighTech Support Program from STCSM (No. 22511106200), as well as research grants from Huawei Technologies and Shanghai Artificial Intelligence Laboratory. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] Amazon ElastiCache for Redis. https://aws.amazon.com/elasticache/redis/.

[2] Bio2RDF: Linked Data for the Life Science. http://bio2rdf.org/.

[3] Cisco nexus 34180yc and 3464c programmable switches data sheet. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html.

[4] DBpedia's SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[5] Distributed Locks with Redis. https://redis.io/docs/manual/patterns/distributed-locks/.

[6] Druid | interactive analytics at scale. https://druid.apache.org/.

[7] Mifos-Mobile Android Application for MifosX. https://github.com/openMF/mifos-mobile.

[8] Mobile Banking App - Flutter. https://github.com/sangvaleap/app-flutter-mobile-banking.

[9] NetLock: Fast, Centralized Lock Management Using Programmable Switches. https://github.com/netx-repo/NetLock/.

[10] Oltpbench. https://github.com/oltpbenchmark/oltpbench/.

[11] Redis Enterprise for Mobile Banking. https://redis.com/solutions/use-cases/redis-enterprise-for-mobile-banking/.

[12] Tatp. https://tatpbenchmark.sourceforge.net/.

[13] Teradata: Business analytics, hybrid cloud & consulting. http://www.teradata.com/.

[14] Tpc-c. http://www.tpc.org/tpcc/.

[15] YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.

[16] A. Agrawal and C. Kim. Intel tofino2 – a 12.9tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32, Los Alamitos, CA, USA, aug 2020. IEEE Computer Society.

[17] Daniel Amir, Tegan Wilson, Vishal Shrivastav, Hakim Weatherspoon, and Robert Kleinberg. Poster: Scalability and congestion control in oblivious reconfigurable networks. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 1138–1140. ACM, 2023.

[18] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, 2020.

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[20] Andrew Brook. Evolution and practice: Low-latency distributed applications in finance: The finance industry has unique demands for low-latency distributed systems. *Queue*, 13(4):40–53, apr 2015.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.

[22] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2477–2489, New York, NY, USA, 2021. Association for Computing Machinery.

[23] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[24] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14:667–668, 1971.

[25] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 727–743, USA, 2018. USENIX Association.

[26] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[27] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[28] A. Devulapalli and P. Wyckoff. Distributed queue-based locking using advanced network features. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 408–415, 2005.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[30] A.B. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 22–31, 1990.

[31] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. A top-down approach to achieving performance predictability in database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 745–758, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'10, page 11, USA, 2010. USENIX Association.

[33] Broadcom Inc. Trident3-x7 / bcm56870 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 35–49, USA, 2018. USENIX Association.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136. ACM, 2017.

[36] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 295–308, USA, 2008. USENIX Association.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[38] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[39] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 223–244. ACM, 2021.

[40] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M. Swift. ATP: in-network aggregation for multi-tenant learning. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 741–761. USENIX Association, 2021.

[41] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[42] Jason Lei and Vishal Shrivastav. Seer: Enabling future-aware online caching in networked systems. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 635–649. USENIX Association, 2024.

[43] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[45] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 387–406. USENIX Association, 2020.

[46] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, March 2016. USENIX Association.

[48] Zhaoyi Li, Jiawei Huang, Yijun Li, Aikun Xu, Shengwen Zhou, Jingling Liu, and Jianxin Wang. A2TP: aggregator-aware in-network aggregation for multi-tenant learning. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 639–653. ACM, 2023.

[49] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 795–809. ACM, 2017.

[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[51] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[52] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. Hiengine: How to architect a cloud-native memory-optimized database engine. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2177–2190, New York, NY, USA, 2022. Association for Computing Machinery.

[53] John Mellor-Crummey and Michael Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. volume 26, pages 106–113, 07 1991.

[54] C. Mohan and Inderpal Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, page 193–207, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[55] Mohindra and Devarakonda. Distributed token management in calypso file system. In *Proceedings of the 1994 6th IEEE Symposium on Parallel and Distributed Processing*, SPDP '94, page 290–297, USA, 1994. IEEE Computer Society.

[56] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 583–590, 2007.

[57] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.

[58] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[59] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[60] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 401–417, USA, 2016. USENIX Association.

[61] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014.

[62] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Vll: A lock manager redesign for main memory database systems. *The VLDB Journal*, 24(5):681–705, oct 2015.

[63] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[65] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, page 19–es, USA, 2002. USENIX Association.

[66] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 433–448, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proc. VLDB Endow.*, 11(5):648–662, jan 2018.

[68] R. Kent Treiber. *Systems Programming: Coping with Parallelism.* Research Report RJ 5118 (53162). International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[69] Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh, and Samer Al-Kiswany. Draconis: Network-accelerated scheduling for microsecond-scale workloads. pages 333–348, 04 2024.

[70] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[71] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[72] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, Carlsbad, CA, July 2022. USENIX Association.

[73] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[74] Parham Yassini, Khaled M. Diab, Saeed Mahloujifar, and Mohamed Hefeeda. Horus: Granular in-network task scheduler for cloud datacenters. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 1–22. USENIX Association, 2024.

[75] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1571–1586, New York, NY, USA, 2018. Association for Computing Machinery.

[76] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 126–138. ACM, 2020.

[77] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. QCMP: load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing, FIRA@SIGCOMM 2023, New York, NY, USA, 10 September 2023*, pages 35–40. ACM, 2023.

[78] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

Chop Chop: Byzantine Atomic Broadcast to the Network Limit

Martina Camaioni Rachid Guerraoui Matteo Monti
Pierre-Louis Roman Manuel Vidigueira Gauthier Voron
Ecole Polytechnique Fédérale de Lausanne (EPFL)

Abstract

At the heart of state machine replication, the celebrated technique enabling decentralized and secure universal computation, lies Atomic Broadcast, a fundamental communication primitive that orders, authenticates, and deduplicates messages. This paper presents Chop Chop, a Byzantine Atomic Broadcast system that uses a novel authenticated memory pool to amortize the cost of ordering, authenticating and deduplicating messages, achieving "line rate" (i.e., closely matching the complexity of a protocol that does not ensure any ordering, authentication or Byzantine resilience) even when processing messages as small as 8 bytes. Chop Chop attains this performance by means of a new form of batching we call *distillation*. A distilled batch is a set of messages that are fast to authenticate, deduplicate, and order. Batches are distilled using a novel interactive protocol involving *brokers*, an untrusted layer of facilitating processes between clients and servers. In a geo-distributed deployment of 64 medium-sized servers, Chop Chop processes 43,600,000 messages per second with an average latency of 3.6 seconds. Under the same conditions, state-of-the-art alternatives offer two orders of magnitude less throughput for the same latency. We showcase three simple Chop Chop applications: a Payment system, an Auction house and a "Pixel war" game, respectively achieving 32, 2.3 and 35 million operations per second.

1 Introduction

Is an Internet computer feasible? A computer that is highly-available, decentralized, secure, universal and shared by all? Theory says yes: state machine replication (SMR) [28, 66] enables decentralized universal computation in the face of arbitrary failures [50, 68]. In practice, however, SMR's inefficiency still makes for limited throughput. At the heart of SMR lies Atomic Broadcast [24], a powerful consensus-equivalent primitive that comes with fundamental bounds [29] and constraints [32], hindering its real-world performance despite decades of extensive research [5, 9, 14, 18, 20, 47, 53, 62, 79, 80, 82] and attention from industry, where SMR powers a myriad of blockchains and ledgers [4, 35, 46, 48, 52, 58, 72–74, 77, 78].

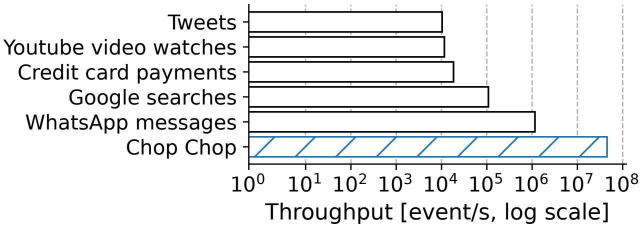


Figure 1: **Throughput of Internet-scale services.**

When deployed globally, seminal Atomic Broadcast implementations, such as BFT-SMaRt [9] and HotStuff [80], can deliver a few thousand messages per second, three orders of magnitude short of the millions of requests per second collectively handled by the Internet's largest, centralized services (Fig. 1). Pushing Atomic Broadcast's throughput into the tens of millions of messages per second seems a necessary stepping stone towards achieving an Internet-scale computer.

Towards line rate. While slow and expensive, ordering messages in Atomic Broadcast is amenable to *batching* [18]: order once, deliver in bulk. This observation motivated the development of *memory pool* (mempool) protocols [26, 34, 69], as initiated by Narwhal [26], designed to amortize ordering. This strategy proved effective, e.g., Bullshark [69] delivers in the order of 380,000 messages per second when accelerated by Narwhal. Despite this improvement, however, state-of-the-art batching still falls short of achieving *line rate*, i.e., matching the communication complexity of a protocol that does not ensure any ordering, authentication, or Byzantine resilience. In such a simplified setting, a server could simply deliver a sequence of application messages as it receives them from the network: b bits received, b bits delivered. Modern connections have enough bandwidth to receive tens of millions of application messages per second:[1] 2.5 orders of magnitude of gap still exist between Atomic Broadcast and unordered, unauthenticated dissemination. It is natural to ask

[1] Payloads as small as 12 bytes can have real-world applications (see §2.1). A 5 Gbit/s link can receive 52 millions such payloads per second.

if such a large gap is inherent to atomicity's unavoidable cost of ordering, authenticating and deduplicating messages. This paper answers in the negative, accelerating Atomic Broadcast by a further two orders of magnitude with a system that performs close to optimal efficiency, i.e., within 8% of line-rate, even when handling 40 million requests per second.

Chop Chop. We present Chop Chop, a Byzantine Atomic Broadcast system using a novel authenticated mempool. Mempools amortize the cost of ordering by having an underlying instance of Atomic Broadcast order batches. Classic methods of batching, however, fail to also amortize authentication and deduplication: each payload in a batch still carries an individual public key, signature and sequence number.

Chop Chop addresses this shortcoming with a new form of batches: *distilled batches*. Unlike a classic batch, a distilled batch contains condensed information that allows to authenticate and deduplicate its messages in bulk, much faster than in existing schemes. Distilled batches leverage the strong ordering of Atomic Broadcast to minimize redundant information.

Trustless brokers. Chop Chop produces distilled batches using a novel interactive protocol involving *brokers*, a layer of facilitating processes between clients and servers. Distilled batches are faster for servers to receive and process, but expensive for brokers to produce: distillation is interactive and relies on expensive cryptographic operations for brokers.

Importantly, however, incorrectly distilled batches are visibly malformed. As such, brokers can be *untrusted*: good brokers take load off the servers; bad ones cannot compromise the system's safety. Servers are exposed to every message in the system, bottleneck easily, and only a threshold of them can be compromised before the system loses safety. Brokers, instead, can be spun up by anyone, outside of Chop Chop's security perimeter, to meet the load produced by clients.

Evaluation. We evaluate Chop Chop in a cross-cloud, geo-distributed environment including 320 medium-sized AWS EC2 machines and 64 OVH machines. We simulate up to 257 million clients and consider 12 experimental environments. Setting up each environment requires the installation of 13 TB of synthetic workload. A naive installation using `scp` from a single machine would take 68 hours. We designed `silk`, a one-to-many peer-to-peer file transfer tool optimized for high latency connections, to install the files in 30 minutes instead.

We compare Chop Chop's throughput and end-to-end latency against its baselines in multiple real-world scenarios including server failures, adverse network conditions, and applications running. In all scenarios, Chop Chop's throughput outperforms its closest competitor by up to two orders of magnitude, with no penalty in terms of latency. When put under stress, Chop Chop orders, authenticates and deduplicates upwards of 43,600,000 messages per second with a mean latency of 3.6 seconds. Except under the most adverse network conditions and proportions of faulty clients, Chop Chop still achieves millions of operations per second.

Applications. Unlike most Atomic Broadcast implementations [9, 26, 69, 80], Chop Chop does not offload authentication and deduplication to the application. This allows Chop Chop-based applications to focus entirely on their core logic without ever engaging in expensive, and easy to get wrong, cryptography. To showcase this, we implement three simple applications to evaluate on top of Chop Chop: a Payment system, an Auction house and an instance of the game "Pixel war". These three simple applications (300 lines of logic) work effectively with messages as small as 8 bytes, further underlying the communication overhead represented by public keys, signatures and sequence numbers in non-distilled systems. Both Payments and Pixel war inherit Chop Chop's throughput, respectively processing over 32 and 35 million operations per second. Even the Auction house, which is single-threaded, achieves 2.3 million operations per second. (These applications are meant as examples, and further optimization is beyond the scope of this paper.)

Contributions. We identify authentication and deduplication as the main bottlenecks of batched Atomic Broadcast; we introduce distilled batches to extend the amortizing properties of batching to authentication and deduplication; we present distillation, an interactive protocol to produce distilled batches, and identify the opportunity to offload it to an untrusted set of brokers; we implement Chop Chop, a Byzantine Atomic Broadcast system that takes advantage of distillation through an authenticated mempool; we thoroughly evaluate Chop Chop, improving state-of-the-art Atomic Broadcast throughput by two orders of magnitude, maintaining near line-rate performance up to 40 million requests per second; we showcase Chop Chop through a Payment system, an Auction house and an instance of the "Pixel war" game, respectively achieving 32, 2.3 and 35 million operations per second.

Roadmap. §2 introduces Atomic Broadcast, discusses classic batching mechanisms and highlights the cost of authenticating and deduplicating messages in the resulting batches. §3 presents distilled batches and introduces a simplified failure-free version of Chop Chop's protocol. §4 describes Chop Chop's fault-tolerant protocol in detail. §5 discusses Chop Chop's implementation. §6 discusses Chop Chop's empirical evaluation, highlighting the challenges of such large scale experiments. We summarize related work in §7 and future work in §8. Appendix A describes Chop Chop's artifact. The full correctness proof of Chop Chop is available online [15].

2 Atomic Broadcast

In an Atomic Broadcast system [19], *clients* broadcast messages that are delivered by *servers*.

Properties [13]. Correct servers deliver the same messages in the same order (*agreement*). Messages from correct clients are eventually delivered (*validity*). Spurious messages cannot be attributed to correct clients (*integrity*). No message is delivered more than once (*no duplication*).

2.1 Cost of Atomic Broadcast

Informally, Atomic Broadcast's most distinctive property, agreement, is also the most challenging to satisfy. Correct servers must coordinate to *order* messages without compromising liveness. A great deal of research effort has been put in developing ordering techniques, optimizing for latency [47, 56] or communication complexity [55, 63].

Integrity and no duplication, instead, allow for simple solutions. Clients can ensure integrity by *authenticating* their messages using digital signatures: servers simply ignore incorrectly authenticated messages. For no duplication, clients can tag each message with a strictly increasing *sequence number*: after ordering, servers discard old messages as replays.

Both techniques—we call them *classic authentication* and *classic sequencing*—are non-interactive, easy to implement, and agnostic of the protocol employed to order messages. Arguably due to the simplicity and effectiveness of classic authentication and sequencing, most Atomic Broadcast implementations overlook integrity and no duplication entirely: they offload authentication and sequencing to the application, focusing on the more challenging task of ordering.

Batching for ordering. Lacking an efficient technique to minimize its complexity, ordering could be Atomic Broadcast's main bottleneck.[2] The well-known strategy of *batching*, however, is both general and effective at amortizing the agreement cost of an Atomic Broadcast implementation [18, 68].

Broadly speaking, batching is orchestrated by a *broker* as follows [26].[3] Over a small window of time, the broker collects multiple client-issued messages in a batch, which it disseminates to the servers; the broker then submits to an underlying instance of Atomic Broadcast a cryptographic hash of the batch it collected; upon delivering the hash of a batch from Atomic Broadcast, a server retrieves the batch, and delivers to the application all the messages it contains. Because the size of a hash is constant, the cost of ordering a batch does not depend on its size: as batches become larger, the cost of ordering each message goes to zero. In practice, batching can effectively eliminate the cost of ordering in any real-world implementation of Atomic Broadcast.

Cost of integrity and no duplication. Batching does not efficiently uphold integrity and no duplication. Regardless of how many messages are batched together, the cost of classic authentication and sequencing stays constant: one public key, one signature and one sequence number for each message.

In practice, these costs dominate the computation and communication budget of a batched Atomic Broadcast system (see §3.2). On the one hand, signatures are among the most CPU-intensive items in the standard cryptographic toolbox, dwarfing in particular symmetric primitives such as hashes and ciphers. On the other, public keys, signatures and sequence numbers can easily account for the majority of a batch's size.

To illustrate these costs, consider the example of a payment system. A payment operation requires three fields: sender, recipient, and amount. Sender and recipient fit in 4 B each if the system serves less than 4 billion users. Amount needs 4 B for payments between 1 cent and 40 millions. Hence, a payment can be encoded in just 12 B. Using public keys to identify sender and recipient (2×32 B using Ed25519 [8, 40]) and attaching a signature (64 B) and a sequence number (8 B) to each message inflates payloads to 140 B. For payments, *91% of the bandwidth is spent on integrity and no duplication*.

2.2 Existing Mitigations

Chop Chop integrates the two following techniques to reduce the bandwidth and CPU cost of authentication.

Short identifiers. Repeated public keys consume a significant slice of a server's communication budget. A workaround is to have servers store public keys in an indexed *directory* [2]. Upon first joining the system, a client announces its public key via Atomic Broadcast to *sign up*. Upon delivering a sign-up message, a server appends the new public key to its directory. The same public key appears at the same position in the directory of all correct servers thanks to Atomic Broadcast's agreement. Having signed up, a client uses its position in the directory as identifier instead of its public key.

In the previous example of a payment system, using such identifiers reduces a payment size by 40%, from 140 B to 84 B. However, a signature per payment must still be transmitted.

Pooled signature verification. Authenticating a batch by verifying its signatures is a computationally intensive task for a server [18, 71]. However, Red Belly [23] and Mir [71] showed that not all servers need to authenticate all batches. Indeed, assuming at most f faulty servers, a broker optimistically asks only $f+1$ servers to authenticate a batch to be certain to reach at least one correct server. If $f+1$ servers do not reply by a timeout, the broker extends its request to f additional servers, thus reaching at least $f+1$ correct servers.

A correct server that authenticates a batch sends back to the broker a *witness shard*, i.e., a signed statement that the batch is correctly signed. The broker aggregates $f+1$ identical shards into a *witness*, which it sends to the other $2f$ servers. Because every witness contains at least one correct shard, the servers can trust the witness instead of verifying the batch.

Assuming $3f+1$ servers, this technique shaves up to two-thirds off the system's authentication complexity.

3 Distilled Batches

Chop Chop's main contribution is *distillation*, a set of techniques aimed at extending the amortizing properties of batches to authentication and sequencing.

[2] Byzantine Atomic Broadcast among n participants cannot be achieved with a bit complexity smaller than $\Theta(n^2)$ [29].

[3] In the literature, servers usually play the role of brokers. As we discuss in §4, however, Chop Chop minimizes its load on the servers by offloading brokerage to a separate, trustless set of processes.

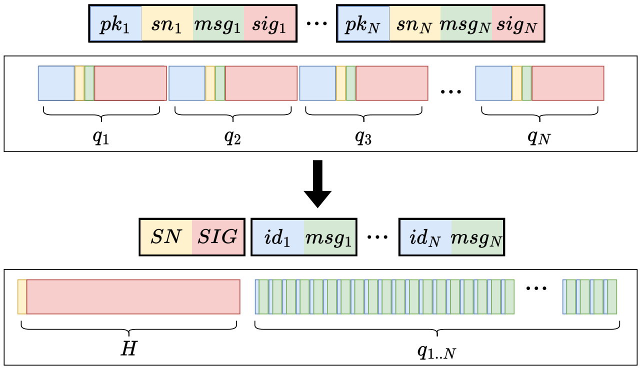


Figure 2: **Full distillation in action.** With classic authentication and sequencing, each payload q_i contains a public key pk_i, a sequence number sn_i, a message msg_i and a signature sig_i. In the fully distilled case, each q_i reduces to just id_i and msg_i: one header H, composed of one aggregate sequence number SN and one aggregate signature SIG, is sufficient for the entire batch. Bars are to scale if small messages are broadcast using Ed25519 for signatures and BLS12-381 for uncompressed multi-signatures: sn_i and SN are 8 B, msg_i is 8 B, pk_i is 32 B, sig_i is 64 B, SIG is 192 B.

Background: multi-signatures. Chop Chop makes use of multi-signature schemes [39] to authenticate batches. Secret keys produce signatures that can be verified against the corresponding public keys. Public keys and signatures, however, can be *aggregated*. Let $(p_1, r_1), \ldots, (p_n, r_n)$ be distinct key pairs, and $s_1, \ldots, s_n$ be signatures produced by $r_1, \ldots, r_n$ on the *same* message m: $p_1, \ldots, p_n$ (resp., $s_1, \ldots, s_n$) can be aggregated into a constant-sized aggregate public key p (resp., aggregate signature s).

Remarkably, s can be verified in constant time against p and m [10, 57]. Chop Chop uses BLS multi-signatures [10] which can be aggregated cheaply and non-interactively: even a non-signing process can compute p (resp., s) once provided with $p_1, \ldots, p_n$ (resp., $s_1, \ldots, s_n$) by computing a single multiplication over an elliptic curve.

3.1 Distillation at a Glance

In brief, distillation aims to produce *distilled batches*. A distilled batch has some of its signatures (resp., sequence numbers) replaced by an *aggregate signature* (resp., *aggregate sequence number*). When maximally successful, distillation produces a *fully distilled batch*, where all signatures (resp., sequence numbers) have been replaced by a *single* aggregate signature (resp., sequence number). As we discuss below, distilled batches are vastly cheaper for servers to receive and process. Fig. 2 depicts the effect of distillation on a batch.

Full distillation (failure-free). For pedagogical purposes, we introduce distillation under the assumption that all processes are correct. We detail Chop Chop's fault-tolerant distillation techniques in §4.2, optimized and adapted to the Byzantine setting. As in the classic batching case, a set $\chi_1, \ldots, \chi_b$ of clients submit their messages $m_1, \ldots, m_b$ to a broker β. Each χ_i selects for its message m_i a sequence number k_i (greater than any sequence number it previously used), then sends (k_i, m_i) to β. Upon receiving all (k_i, m_i)-s, β computes the aggregate sequence number

$$k = \max_i k_i$$

then builds the *batch proposal*

$$B = [(x_1, k, m_1), \ldots, (x_b, k, m_b)]$$

where x_i is χ_i's numerical identifier in the system (see §2.2). β then sends B back to every χ_i. Upon receiving B, χ_i produces a multi-signature s_i for the hash $H(B)$ of B, which it sends back to β. Having collected all multi-signatures, β computes the aggregate signature

$$s = \prod_i s_i$$

In doing so, β obtains the fully distilled batch

$$\tilde{B} = [s, k, ((x_1, m_1), \ldots, (x_b, m_b))]$$

Upon receiving $\tilde{B}$, any server now can: compute B by inserting k between each (x_i, m_i); compute $H(B)$; use each x_i to retrieve χ_i's public key p_i from its directory; compute the aggregate public key

$$p = \prod_i p_i$$

and finally verify s against p and $H(B)$.

Distillation outcome. Having engaged with β to distill the batch, every χ_i multi-signs the *same* message $H(B)$ and updates its sequence number to the *same* k. This allows β to authenticate and sequence all of $\tilde{B}$ using s and k only.

Distillation safety. The proposed distillation protocol has no safety drawback. First, because (x_i, k, m_i) appears in B, χ_i still gets to authenticate m_i. Intuitively, χ_i's multi-signature on $H(B)$ publicly authenticates *whatever message in B is attributed to* χ_i, m_i in this case. Second, because $k \geq k_i$, k is still a valid sequence number for m_i. Sequence number distillation might cause χ_i to skip some sequence numbers whenever any χ_j issues some $k_j > k_i$. Contiguity of sequence numbers, however, is not a requirement for deduplication. As with classic sequencing, χ_i produces—and servers deliver—messages with strictly increasing sequence numbers; servers disregard all other messages as replays.

3.2 Distillation Microbenchmark

Having discussed how distilled batches are produced, we now estimate the significance of their effect by means of a back-of-the-envelope calculation and a simple microbenchmark on AWS. Consider a setting where 100 million clients broadcast 8-byte messages, e.g., to issue payments (see § 2.1). We compare *classic authentication and sequencing*, where clients are identified by their public keys, messages are individually signed and sequenced, against *fully distilled batches* where

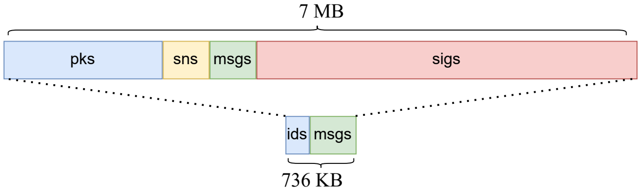


Figure 3: **Full distillation of a batch of 65,536 payloads (sizes to scale).** The aggregate signature and aggregate sequence number do not appear as a result of their small size.

clients are identified by a numerical identifier and each batch contains only one aggregated signature and sequence number. We use Ed25519 [40] for signatures (32 B public keys, 64 B signatures) and BLS12-381 [12] for multi-signatures (192 B uncompressed signatures). We use uncompressed BLS multi-signatures to save computation time at the cost of storage space (96 B compressed vs. 192 B uncompressed).

Communication complexity. Payloads are 112 B per message in the classic case (32 B of public key, 8 B of sequence number, 8 B of message, 64 B of signature) vs. 11.5 B in the fully distilled case (28 bits = 3.5 B of identifier to represent 257M clients, 8 B of message). Assuming batches of 65,536 messages (Fig. 3), classic batches are exactly 7 MB long, while fully distilled batches are 736 KB long including aggregate signature and sequence number.

Computation complexity. Running at maximum load, an Amazon EC2 c6i.8xlarge instance authenticates 16.2 ± 0.4 classic batches per second using Ed25519's batch verification for 65,536 signatures. The same machine authenticates 457.1 ± 0.3 fully distilled batches per second: each authentication requires the aggregation of 65,536 BLS12-381 public keys and the verification of one BLS12-381 multi-signature.

Summary. By the order-of-magnitude calculations above, fully distilled batches hold the promise to reduce the costs of authentication and sequencing by a factor 9.7 for network bandwidth, and 28.2 for CPU. Chop Chop aims to deliver on that promise for a real-world fault-tolerant system.

4 Chop Chop

This section overviews Chop Chop's architecture, Chop Chop's protocol, and provides arguments for its correctness.

Overview. Chop Chop involves three types of processes (Fig. 4): broadcasting clients, delivering servers and a layer of broadcast-facilitating brokers between them. Servers run an Atomic Broadcast instance among themselves, to which brokers submit messages. Chop Chop is agnostic to the implementation of Atomic Broadcast used by the servers. On top of the provided broker-to-server Atomic Broadcast, Chop Chop implements a much faster client-to-server Atomic Broadcast: clients submit messages to the servers, aided by brokers.

Chop Chop's protocol unfolds in two phases: *distillation* (§4.2) and *submission* (§4.3). In the distillation phase, clients interact with a broker to gather their messages in a distilled batch (see §3). In the submission phase, the broker disseminates the distilled batch to the servers and submits the batch's hash to the server-run instance of Atomic Broadcast. Upon delivering its hash from Atomic Broadcast, servers retrieve the batch and deliver its messages. Chop Chop's contributions mainly focus on the distillation phase. Chop Chop's submission strategy closely resembles prior batch-based Atomic Broadcast implementations [26, 34, 69].

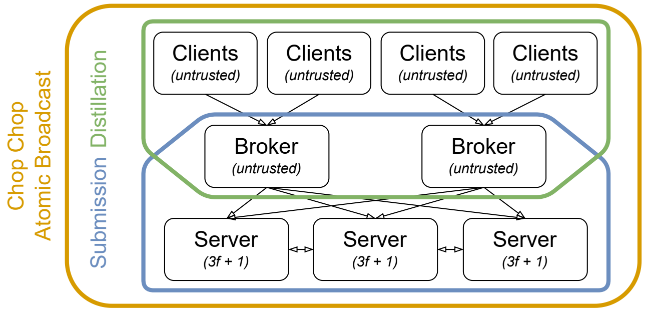


Figure 4: **Chop Chop architecture.**

4.1 Architecture and Model

Chop Chop augments the architecture of a classic Atomic Broadcast, as described in §2, with novel brokers.

Clients and servers. *Clients* broadcast messages to a (distinct) set of *servers*. We assume that less than one third of servers can be faulty and behave in an arbitrary manner, i.e., be Byzantine [50], while all clients can be faulty. For simplicity, servers form a fixed set that is known by all correct processes at system startup. Chop Chop can be extended for reconfiguration thanks to its modular use of Atomic Broadcast [9, 49] (Fig. 4). Clients issue messages after broadcasting their public keys to the system (see §2.2).

Brokers. We discussed in §3 how both classic and distilled batches are assembled by a broker. The role of brokers is traditionally taken by servers. Given the additional strain put on brokers by Chop Chop's interactive distillation protocol, however, having servers be brokers would result in a waste of scarce, trusted resources. Importantly, however, distillation is *trustless*. On the one hand, agreement rests entirely on Chop Chop's underlying Atomic Broadcast instance, for which brokers are only clients. On the other hand, as we argue in §§4.2 and 4.4.1, a faulty broker cannot compromise integrity or no duplication: distilled batches are publicly authenticated, and correct clients cannot be tricked into using stale sequence numbers. Hence, *brokers need no trust*: a broker either does its job correctly or produces distilled batches that are visibly malformed, and easily discarded by all correct servers.

This observation is of paramount importance to the performance of Chop Chop: *because distillation is heavy but trustless, brokers should be distinct from servers.* Along with clients and servers, we thus assume a third, *independent set of brokers*, sitting between clients and servers, to accelerate Atomic Broadcast by assembling client messages in distilled batches. We assume that at least one broker is correct; the system loses liveness but not safety if all brokers are faulty.

Network. Chop Chop guarantees that the batches collected and submitted to servers by correct brokers are well-formed even in asynchrony, but achieves full distillation when the network is synchronous (see §4.2). Chop Chop inherits the network requirements of its underlying Atomic Broadcast.

4.2 Distillation Phase

We introduced in §3 a simplified, failure-free distillation protocol. This section describes how Chop Chop renders distillation tolerant to arbitrary failures and improves its performance via a sequence of improvements, each addressing a shortcoming of the simplified protocol. The complete fault-tolerant protocol of Chop Chop is depicted in Fig. 5.

In the failure-free distillation protocol: clients $\chi_1, \ldots, \chi_b$ send their messages $m_1, \ldots, m_b$, with sequence numbers $k_1, \ldots, k_b$ (#2) to a broker β (Fig. 5, #1); β identifies the maximum submitted sequence number k and builds a batch proposal $B = [(x_1, k, m_1), \ldots, (x_b, k, m_b)]$ (#3); β disseminates B to $\chi_1, \ldots, \chi_b$ (#4); each χ_i produces a multi-signature s_i on $H(B)$ (#5), which it sends to back β (#6); β aggregates $s_1, \ldots, s_n$ into an aggregate s, thus producing a fully distilled batch $\tilde{B} = [s, k, ((x_1, m_1), \ldots, (x_b, m_b))]$ (#7).

Background: Merkle trees. Chop Chop uses Merkle trees [59] to hash batches. An l-element vector $z_1, \ldots, z_l$ is hashed into a *root* r, used as commitment. For each i, z_i's value can be proved by means of a proof of inclusion p_i, verifiable against r and z_i. Proofs of inclusions are $O(\log l)$ in size and are verified in $O(\log l)$ time.

What if a broker forges messages? A faulty β could try to falsely attribute to some χ_i a message $m'_i \neq m_i$. β could do so by replacing m_i with m'_i in B, then having χ_i sign $H(B)$, thus implicitly authenticating m'_i. This is easily fixed by having χ_i check that m_i correctly appears in B before signing $H(B)$.

Can a broker avoid sending the entire batch? A clear inefficiency of the simplified protocol is that β has to convey all of B back to each χ_i. This is fixed using Merkle trees. Upon assembling B, β computes the Merkle root r of B, along with the Merkle proof p_i for each (x_i, k, m_i) in B. Instead of sending B to all clients, β just sends r, k and p_i to each χ_i. Upon receiving r, k and p_i, χ_i checks p_i against r and (x_i, k, m_i), producing s_i on r only if the check succeeds. If χ_i signs r, then (x_i, k, m_i) is necessarily an element of B. Importantly, however, β could inject $(x_i, k, m'_i \neq m_i)$ somewhere else in B, while still providing χ_i only with the proof for (x_i, k, m_i). This is solved by having servers ignore every distilled batch where two or more messages are attributed to the same client. This way, if χ_i signs r, then either m_i is the only message in B attributed to χ_i, or $\tilde{B}$ is rejected by all servers as malformed: either way, integrity is upheld.

What if a client does not multi-sign? Under the assumption that $\chi_1, \ldots, \chi_b$ are correct, β can safely wait until it collects all $s_1, \ldots, s_b$. This policy is clearly flawed in the Byzantine setting: a single crashed client can prevent β from ever aggregating s. Furthermore, lacking an assumption of synchrony, β cannot exclude from $\tilde{B}$ those clients that do not sign r by some timeout: consistently slow clients would always be excluded, and validity would be lost. This issue is fixed by the fallback mechanism introduced in the following.

Fault-tolerant distillation. Upon first sending (k_i, m_i) to β (#2), χ_i also sends an individual, non-aggregable signature t_i for (x_i, k_i, m_i), which β stores. β then waits for s_i-s on r until either all s_i-s are collected, or a timeout expires. For every s_i that ends up missing, due to χ_i being crashed or delayed, β attaches (k_i, t_i) to $\tilde{B}$. Upon receiving $\tilde{B}$, a server first checks each individual signature t_i against the corresponding (x_i, k_i, m_i). The server then checks s against the public keys of the clients for which an individual signature t_i was not given, i.e., the public keys of all clients that signed r in time.

In summary: fast, correct clients who successfully produce their s_i-s in time authenticate their message by multi-signing r; slow or crashed clients still get their messages through, individually authenticated by the t_i-s that they originally produced. Full distillation is achieved whenever the network is synchronous and all clients are correct, which we argue is the case in practice for the majority of a system's lifetime. When the network is asynchronous, however, a fraction of clients might fail to produce their s_i in time, resulting in a *partially distilled batch*. At the limit where all clients fail to sign r in time, $\tilde{B}$ reduces to a classic batch, degrading server-side performance to pre-distillation levels. We underline that safety and liveness are preserved regardless of synchrony.

What if a broker replays messages? A problem introduced by the last fix is that χ_i authenticates both k_i and k as sequence numbers for m_i, allowing a faulty β to play m_i twice, hence breaking Atomic Broadcast's no duplication. This is fixed by having each client engage in the broadcast of only one message at a time. This way, while β can indeed replay m_i, it can only do so consecutively: all sequence numbers χ_i authenticates for m_i belong to a range that does not contain sequence numbers for any other message $m_{i' \neq i}$ issued by χ_i.

This observation is key to the following fix: along with the last sequence number $\bar{k}_\chi$ each client χ used, a correct server σ stores the last message $\bar{m}_\chi$ that χ broadcast; upon ordering a message m with sequence number k from χ, σ delivers m if and only if $k > \bar{k}_\chi$ and $m \neq \bar{m}_\chi$. In doing so, σ discards all consecutive replays of $\bar{m}_\chi$, thus preventing replays in general.

What if a client broadcasts too frequently? The last fix relies on clients broadcasting one message at a time. Depending on latency, a client broadcasting too frequently might

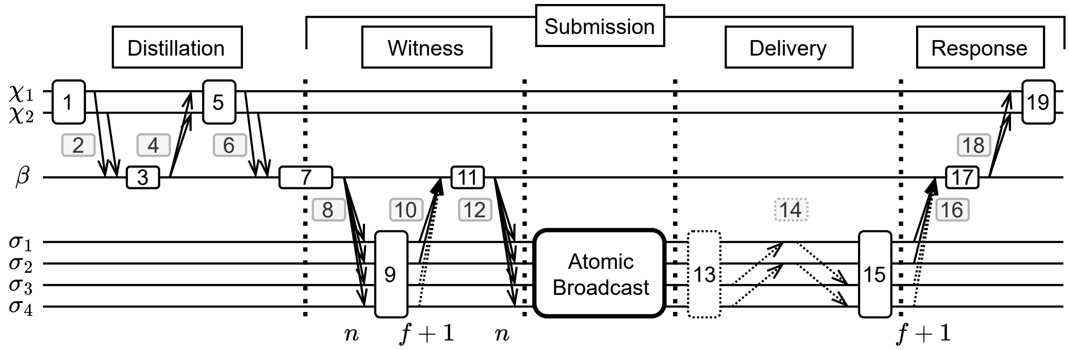


Figure 5: **Overview of the Chop Chop protocol between two clients (χ_1, χ_2), a broker (β) and four servers (σ_1–σ_4).** The protocol is comprised of 19 steps (#1–#19) and of an underlying instance of Atomic Broadcast such as BFT-SMaRt or HotStuff.

accrue an ever-growing queue of pending messages. This issue is fixed by flushing application messages in bursts, akin to Nagle's buffering algorithm for TCP.

What if a client uses the largest possible sequence number? Assuming that a finite number of bits (e.g., 64) are allocated to representing sequence numbers, a faulty client χ_m could set its k_m to the largest possible sequence number k_{max} (e.g., $2^{64}-1$). In doing so, χ_m would force all other χ_i-s to update their sequence number to k_{max}. Since correct clients only use strictly increasing sequence numbers, no χ_i could ever broadcast again: sequence numbers would run out. Proving the *legitimacy* of sequence numbers fixes this issue.

Legitimate sequence numbers. By the rule that we established, no more than one message from the same client can appear in the same batch. Moreover, correct clients always tag their messages with the smallest sequence number they have not yet used, i.e., the largest they have used plus one. By induction, we then have that unless some client misbehaves, no client ever needs to use a sequence number larger than the number of batches ever delivered by the servers: the largest sequence number any client submits to the very first batch is 0, therefore no client submits a sequence number larger than 1 to the second batch, and so on. This observation allows us to define as *legitimate* any sequence number smaller than the number of batches servers have delivered at any given time.

Legitimacy proofs. This definition of legitimacy allows for the generation of *legitimacy proofs*: upon delivering the n-th batch, a server publicly states so with a signature. By collecting $f+1$ server signatures stating that the n-th batch was delivered into a certificate l_n, any process can publicly prove that any sequence number smaller than n is legitimate.

Upon initially submitting k_i (#2), χ_i also sends to β a certificate l_n, for any $n > k_i$; β ignores client submissions that lack such certificate, except when $k_i = 0$ since no certificate is needed. Upon sending k back to all χ_i-s (#4), β attaches the highest $l_{\hat{n}}$ it collected: $l_{\hat{n}}$ proves that k is legitimate since $\hat{n} > k$. χ_i signs r (#5) only if k is proved legitimate by $l_{\hat{n}}$.

This technique ensures correct clients always use legitimate sequence numbers. Since legitimate sequence numbers grow only with the number of delivered batches, no correct client is forced to skip too far ahead, compromising its own liveness.

What if a broker crashes? If β fails to engage in the protocol, each χ_i can submit its message to any other broker.

4.3 Submission Phase

The submission phase ensures that all servers efficiently deliver a distilled batch, and that all broadcasting clients receive a proof that their messages were delivered.

Witness. Having gathered a distilled batch $\tilde{B}$ (#7), β moves on to have $f+1$ servers signs a *witness shard* for $\tilde{B}$. In signing a witness shard for $\tilde{B}$, a server σ simultaneously makes two statements. First, $\tilde{B}$ is *well-formed*: σ successfully verified $\tilde{B}$'s signatures and found all messages in $\tilde{B}$ to have a different sender. Second, $\tilde{B}$ is *retrievable*: σ stores $\tilde{B}$ and makes it available for retrieval, should any other server need it. We call a *witness* for $\tilde{B}$ the aggregation of $f+1$ witness shards for $\tilde{B}$. Because any set of $f+1$ processes includes a correct process, when presented with a witness for $\tilde{B}$ any server can trust $\tilde{B}$ to be well-formed and retrievable.

As discussed in §2.2, witnesses optimize server-side computation. Only $f+1$ servers need to engage in the expensive checks required to safely witness $\tilde{B}$. All other servers can trust $\tilde{B}$'s witness, saving trusted CPU resources.

In order to collect a witness for $\tilde{B}$, β sends $\tilde{B}$ to all servers (#8). Optimistically, β asks only $f+1$ servers to sign a witness shard for $\tilde{B}$, progressively extending its request to $2f+1$ servers upon expiration of suitable timeouts. Upon receiving $\tilde{B}$ (#9), a correct server σ stores $\tilde{B}$. If asked to witness $\tilde{B}$, σ checks that $\tilde{B}$ is well-formed and sends back to β its witness shard for $\tilde{B}$ (#10). β collects and aggregates $f+1$ shards into a witness for $\tilde{B}$ (#11), then submits $\tilde{B}$'s hash and witness to the server-run Atomic Broadcast (#12).

Delivery. Upon delivering $\tilde{B}$'s hash and witness from Atomic Broadcast (#13), a correct server σ retrieves $\tilde{B}$, either from its local storage (if it directly received $\tilde{B}$ from β at #8) or from another server (#14). Because $\tilde{B}$ is retrievable, σ

is guaranteed to eventually find a server to pull $\tilde{B}$ from. Having retrieved $\tilde{B}$ (#15), σ delivers all non-duplicate messages in $\tilde{B}$ (see §4.2 for how σ detects duplicates).

Response. Finally, σ signs a *delivery certificate*, listing the messages in $\tilde{B}$ that σ delivered. σ sends its signature back to β (#16). By agreement of Atomic Broadcast, all correct servers deliver the same subset of messages in $\tilde{B}$. As such, β is guaranteed to eventually collect $f+1$ signatures on the same delivery certificate (#17). Upon doing so, β distributes a copy of $\tilde{B}$'s delivery certificate to $\chi_1, \ldots, \chi_b$ (#18). Armed with $\tilde{B}$'s delivery certificate, a correct χ_i can publicly prove the delivery of m_i (#19) and safely broadcast its next message.

4.4 Correctness

This section summarizes Chop Chop's correctness analysis. We prove Chop Chop's correctness to the fullest extent of formal detail in an extended document available online [15].

4.4.1 Safety

The safety of Chop Chop is given by its agreement, integrity and no duplication properties (see §2).

Agreement. Chop Chop inherits agreement from its underlying, server-run instance of Atomic Broadcast. A correct server delivers messages only upon delivering the hash of a batch from the server-run Atomic Broadcast. Upon doing so, a correct server retrieves the full batch, checks its hash, and delivers all its messages in order of appearance. All correct servers deliver the same messages in the same order assuming cryptographic hashes are collision-resistant.

Integrity. A correct server only delivers messages included in a batch witnessed by $f+1$ servers, i.e., by at least one correct server. A correct server witnesses a batch only if: no more than one message in the batch is attributed to the same client; every client in the batch authenticates its message with a signature or the root of the batch's Merkle tree with a multi-signature. A correct client multi-signs the root of a batch's Merkle tree only upon receiving a proof of the inclusion of its message in the batch. As such, if a correct client multi-signs the root of a batch's Merkle tree, either the batch contains only the client's intended message or it is not witnessed. In summary, a correct server delivers a message m from a correct client χ only if χ broadcast m.

No duplication. A correct client only broadcasts one message at a time. As such, while the client might attach multiple sequence numbers to the same message (different brokers may propose different aggregate sequence numbers for the client to authenticate) the sequence numbers the client attaches to each message belong to distinct ranges. A correct server delivers client messages only in increasing order of sequence number, and ignores repeated messages. This means that a correct server delivers at most one message from each sequence number range. In summary, no server delivers a correct client's message more than once.

4.4.2 Liveness

The liveness of Chop Chop is given by its validity property.

Validity. If a correct client submits its message to a correct broker, the message is guaranteed to eventually be delivered by all correct servers: even if the client fails to engage in distillation in a timely manner, its message is still included in a batch which gets disseminated, witnessed and delivered by all correct servers. Faulty brokers can clearly refuse to service (specific) clients. Upon expiration of a suitable timeout, however, a correct client submits its message to a different broker. As we assume that at least one broker is correct, all correct clients are eventually guaranteed to find a correct broker and get their messages delivered by all correct servers.

4.4.3 Other Attacks

As we outlined in §§4.4.1 and 4.4.2, Chop Chop satisfies all properties of Atomic Broadcast. In this section, we consider other attacks an adversary might deal to impair Atomic Broadcast's performance and fairness [43] in Chop Chop.

Denial of service. A faulty broker may refuse to service clients, thus forcing them to fall back on other brokers, increasing latency. A faulty broker may also submit deliberately non-distilled batches to servers to force them to waste trusted resources to receive and verify individual signatures. While handling DoS is beyond the scope of this paper, Chop Chop is amenable to accountability mechanisms [36]. Brokers could be asked to stake resource to join the system. Correct, high-performance brokers could be rewarded, akin to gas fees in Ethereum [78]. Brokers that accrue a reputation of misbehavior or slowness could be banned and lose their initial stake.

Front-running. A faulty broker might impact fairness by front-running messages of interest [25, 83]. While front-running resistance is beyond the scope of this paper, Chop Chop is compatible as-is with existing mechanisms to mitigate or prevent front-running, most notably schemes that have clients submit encrypted messages whose content is revealed only after delivery [60, 81]. Importantly, these encrypt-order-reveal schemes could be selectively employed only for those messages that are vulnerable to front-runs, e.g., messages used for stock trading [65]. Maintaining Chop Chop's throughput while providing quorum-enforced fairness for every message [82] opens a valuable future avenue of research.

5 Implementation Details

A straightforward implementation of the protocol we presented in §4 would not achieve the throughput and latency we observe in §6. In this section, we discuss some of the techniques and optimizations required on the way to practically achieving Chop Chop's full potential. (Many optimizations are however left out due to space constraints).

Code. Chop Chop is implemented in Rust, totaling 8,900 lines of code. The main libraries Chop Chop depends on are: tokio 1.12 for an asynchronous, event-based runtime; rayon 1.5 for worker-based parallel computation; serde 1.0 for serialization and deserialization; blake3 1.0 for cryptographic hashes; ed25519-dalek 1.2 for EdDSA signatures on Curve25519 [40]; blst 0.3.5 for multi-signatures on the BLS12-381 curve [12]. Chop Chop also depends on in-house libraries: talk (9,800 lines of code) for basic distributed computing and high-level networking and cryptography; zebra (7,100 lines of code) for Merkle-tree based data structures.

5.1 Broker

The goal of a Chop Chop broker is to produce batches as distilled as possible (to minimize server load), as large as possible (to amortize ordering), and as quickly as possible (to minimize latency). Our target is for a broker to assemble one fully distilled batch of 65,536 messages ($\sim$ 736 KB, see Fig. 3) per second, with a 1 second distillation timeout.

Reliable UDP. Short-lived TCP connections between broker and clients are easier to work with, but unfeasible for the broker to handle. Assuming an end-to-end broadcast time of up to 10 seconds, the broker would need to maintain upwards of 600,000 simultaneous TCP connections, which preliminary tests immediately proved unfeasible on the hardware we have access to. This makes UDP the only option for client-broker communication. However, UDP lacks the reliability properties of TCP, and tests showed non-negligible packet loss even within the same AWS EC2 availability zone. As we discussed in §4.2, message loss immediately translates to partial distillation. We address this issue by means of an in-house, ACK-based, message retransmission protocol based on UDP that also smoothens the rate of outgoing packets.

EdDSA batch verification. To avoid spoofing, all client messages are authenticated with signatures. At the target rate, however, individually verifying each signature is unfeasible for a broker. Luckily, ed25519-dalek allows for more efficient batched verification. A broker buffers the client messages it receives and authenticates them in batches.

Tree-search invalid multi-signatures. Clients contributing to the same batch produce matching multi-signatures for the batch's root. At the target rate the broker cannot independently verify each multi-signature. We tackle this problem by gathering multiple matching multi-signatures on the leaves of a binary tree: internal nodes aggregate their children. For each tree, the broker verifies the root multi-signature, recurring only on the children of an invalid parent. This allows to identify invalid multi-signatures in logarithmic time while enabling batched verification in the good case.

Caching legitimacy proofs. Clients justify their sequence numbers with legitimacy proofs. Again, the broker cannot verify each proof in time. We address this problem by having the broker verify a legitimacy proof only if higher than the highest it previously observed. As a result, a faulty client might get away with submitting an invalid legitimacy proof but, importantly, not an illegitimate sequence number.

5.2 Server

The goal of a Chop Chop server is to process distilled batches as quickly as possible without overflowing its memory.

Batch garbage collection. Servers update each other on which batches they delivered. A server garbage-collects a batch, both messages and metadata, as soon as it is delivered by all other servers. We underline that, even if a single server fails to deliver a batch, the others cannot garbage-collect it as the slow server might be correct. This is an inherent limitation of Atomic Broadcast: agreement without synchrony can be ensured only in the infinite-memory model.

Identifier-sorted batching. No two messages from the same client must appear in the same batch. To simplify processing, brokers sort the messages in a batch by client identifier. Servers reject batches whose identifiers are not strictly increasing, thus verifying that all identifiers are distinct in constant size and in linear time. Sorting messages by identifier also enables parallel deduplication: messages are split by identifier range, chunks are deduplicated independently.

6 Evaluation

We evaluate Chop Chop focusing on the following research questions (RQs): What workload can Chop Chop sustain (§ 6.3)? What are the benefits of Chop Chop's distillation (§6.4)? How does Chop Chop scale to different numbers of servers (§6.5)? How efficiently does Chop Chop use resources overall (§6.6)? How does Chop Chop perform under adverse conditions, such as server failures (§6.7)? What performance can applications achieve using Chop Chop (§6.8)?

6.1 Baselines

We compare Chop Chop against four baselines:

- HotStuff [80]: an Atomic Broadcast protocol designed for high-throughput (written in C++);
- BFT-SMaRt [9]: an Atomic Broadcast protocol, similar to PBFT [18], designed for low-latency (written in Java);
- Narwhal-Bullshark: the DAG-based Atomic Broadcast protocol Bullshark [69] with the state-of-the-art high-throughput mempool Narwhal [26] (written in Rust);
- Narwhal-Bullshark-sig: akin to Narwhal-Bullshark but with Narwhal modified to authenticate messages, thus matching Chop Chop's guarantees.

We deploy Chop Chop with two distinct underlying Atomic Broadcast protocols (Fig. 5): HotStuff and BFT-SMaRt.

HotStuff and BFT-SMaRt. Evaluating HotStuff and BFT-SMaRt allows us to assess the base performance of an Atomic Broadcast protocol and determine how much acceleration Chop Chop provides. We evaluate Chop Chop on top of the same implementations of HotStuff [22] and BFT-SMaRt [21] we benchmark against. These implementations

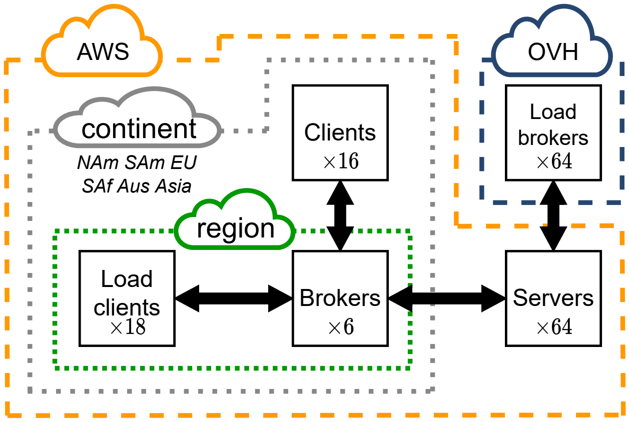


Figure 6: **Cross-cloud deployment summary.**

are production-ready and do not use state-of-the-art mempool protocols, only some basic form of batching. When evaluated stand-alone, each message in these systems includes 80 B of header composed of a client identifier (8 B), a sequence number (8 B), and a signature (64 B) verified by the servers. Both systems use batches of 400 messages, i.e., of 34.4 KB.

Narwhal-Bullshark. As a state-of-the-art mempool, Narwhal is a close point of comparison for Chop Chop. Servers in Narwhal scale out following a primary-workers model: each server is paired with one or several workers into a server group. Similarly to Chop Chop, Narwhal greatly accelerates its underlying Atomic Broadcast (here, Bullshark). Unlike Chop Chop, however, Narwhal leaves the responsibility of authenticating and deduplicating messages to the application.

Narwhal-Bullshark-sig. For a better comparison, we also benchmark Narwhal-Bullshark-sig: Narwhal-Bullshark where messages are authenticated by Narwhal in a state-of-the-art way, i.e., using batched, multi-core Ed25519 signature verification. Each message includes an 80 B header as for HotStuff and BFT-SMaRt. As for Narwhal-Bullshark, the remaining parameters are the default ones, e.g., 500 KB batches.

6.2 Setup

Unless otherwise specified—in §§ 6.5 and 6.6—the Chop Chop benchmarks involve 64 c6i.8xlarge AWS servers, of 32 Intel vCPUs each, geo-distributed across 14 regions. Brokers assemble, and servers process, batches of 65,536 messages. Each message is 8 B in length, resulting in 736 KB batches (Fig. 3). Baselines always use the same set of server machines as their Chop Chop counterpart. All experiments run with maximum resilience, e.g., the system survives 21 faulty servers out of 64. Fig. 6 overviews the used deployment.

Matching trusted and total resources. Unlike its baselines, Chop Chop leverages *untrusted* resources, brokers, to boost its performance. Lacking a well-defined conversion between trusted and untrusted resources, two extremes can be taken to compare Chop Chop with its baselines: we can either match trusted resources, e.g., same number of Chop Chop servers as Narwhal workers, or match total resources, e.g., same number of servers and brokers in Chop Chop as workers in Narwhal.

Intuitively, the first approach considers untrusted resources to be free while the second considers untrusted resources to be as costly as trusted resources. We use the first approach in §§6.3 to 6.5, 6.7 and 6.8 to stress Chop Chop, provisioning the system with enough brokers to bottleneck servers. We use the second approach in §6.6 to assess how efficiently Chop Chop uses its hardware resources, trusted or not.

Load clients and load brokers. We show in §6.3 that Chop Chop servers handle up to 43.6 million operations per second with an average latency of 3.6 seconds. To produce this level of workload, a real-world deployment would require over 700 brokers, each handling around 200,000 clients broadcasting back-to-back thus totaling hundreds of millions of machines. As we cannot experiment at such a scale, we introduce two new actors: *load clients* and *load brokers*. (In the rest of this section, “brokers” and “clients” denote real brokers and real clients; the term “load” is always used explicitly.)

Load clients connect to brokers and simulate thousands of concurrent client requests. Most system evaluation typically use this approach to stress the system and measure latency. However, we explicitly separate clients from load clients in this evaluation. Clients run on very small machines—less powerful than most smartphones—to provide more accurate end-to-end latency measurements. We similarly split clients from load clients in all baseline runs.

Load brokers are unique to Chop Chop. Even using load clients, we could not deploy enough brokers to bottleneck Chop Chop’s servers. Load brokers work around this limitation, submitting batches of pre-generated messages directly to the servers. Free from interactions with clients and expensive cryptography, a load broker puts on the servers a load equivalent to that of tens of brokers working at full capacity.

Using load clients and load brokers, we manage to show that brokers can quickly generate large batches of messages, and servers can process large numbers of batches.

Cross-cloud deployment. All servers are deployed on AWS, balanced across 14 regions: Cape Town, São Paulo, Bahrain, Canada, Frankfurt, Northern Virginia, Northern California, Stockholm, Ohio, Milan, Oregon, Ireland, London, and Paris. For system sizes of 8 in §6.5, we distribute servers across the first 8 regions from the list, which constitute the most adversarial setup with the highest pairwise latency.

Load brokers are placed in a separate cloud provider, OVH, for two purposes. First, it provides a better representation of Internet load than a single-cloud deployment. AWS operates under its own AS so any AS peering bottlenecks would be bypassed by an AWS-only deployment. Second, OVH is one of the few cloud providers with enough peering with AWS to stress Chop Chop without charging for egress bandwidth, saving us from using AWS' costly bandwidth. The final cost amounted to 25,000 USD in AWS credits. Using OVH saved us more than 70,000 USD since each of Chop Chop's data point on a figure would have cost 1,700 USD in AWS egress bandwidth—21 TB at 0.08 USD per GB ≈ 1,700 USD.

For all experiments, we deploy one broker in each continent (Cape Town, São Paulo, Tokyo, Sydney, Frankfurt, and Northern Virginia) and one client in each of the 14 regions above, plus Tokyo and Sydney. Clients connect to their nearest broker. We configure the network for geo-distribution and high load, e.g., TCP buffer sizes [37] and UDP parameters.

All baselines run on the same parameters. For Narwhal-Bullshark, we collocate each server with one of the workers in its server group. We reproduced Narwhal-Bullshark's original experiments [69] and matched the results.

Hardware. All servers, brokers and load clients run on c6i.8xlarge machines with an Intel Xeon Platinum 8375C (32 virtual CPUs, 16 physical cores, 2.9 GHz baseline, 3.5 GHz turbo), 64 GB of memory and 12.5 Gb/s of bandwidth. We selected these machines since they provide good performance and are in the same "commodity" price range as those chosen initially for Chop Chop's main baseline: Narwhal-Bullshark. Clients run on t3.small machines: 2 vCPUs, 1 physical core, 2 GB of memory, and up to 5 Gb/s bandwidth—of which they use less than 1 KB/s. All machines run Linux Ubuntu 20.04 LTS on the AWS patched version of the Linux kernel 5.15.0, except for the load brokers on OVH which run on Linux kernel 5.4.0—the same kernel was not available.

Challenges. The most significant evaluation challenges arose from the scale of the targeted deployment. The setup and orchestration alone required simultaneous handling of up to 320 machines across two different cloud providers and 25 regions, as well as transferring 13TB of files—mostly public keys and pre-generated batches—for each of the 12 setups. To handle this, we developed a new command-line tool to efficiently deploy distributed systems: `silk`. Among other things, we use `silk` for peer-to-peer-style file transfer over aggregated TCP connections, as well as for grouped process control. With `silk`, transferring all files from a single machine takes around 30 minutes, compared to 68 hours with `scp`. The code for `silk` can be found at [*anonymized*].

Additional challenges came from the real-world nature of the targeted deployment. First, the connection between OVH and AWS's Asia and Pacific regions was particularly unstable at certain times of day especially when close to saturation. For example, Tokyo's connection was frequently degraded between 3pm and 5pm UTC. Second, the performance of some machines sometimes deviated from their specifications. As an example, in a setup size of 64, we observed around 2 machines operating with a 10% lower CPU turbo clock rate than specified. Considering these variations, we increased the number of servers a broker initially asks for witness shards (see §4.3) by a margin, e.g., $f+5$ instead of $f+1$. This improves system stability—i.e., lower latency variability—while slightly reducing maximum throughput. Unless otherwise specified, we set the margin to 4 in all experiments, i.e., $f+5$.

Plots. Every data point is the mean of 5 runs of 2 minutes each (after excluding warmup and cooldown, the relevant cross-section is at least 1 minute). All plots further depict one standard deviation from the mean using either colored shaded areas or black error bars (which may be too small to notice). Experimental data can be found at [*anonymized*].

6.3 RQ1 – Load Handling

Fig. 7 shows the latency and throughput of Chop Chop and all its baselines for various input rates of 8 B messages. The variability is represented using shaded areas.

Baselines. Both BFT-SMaRt and HotStuff showcase stable performances under low loads, respectively achieving around 1,400 and 1,600 operations per second. BFT-SMaRt's latency is consistently better than HotStuff's up to its inflection point (0.45–0.53 s vs. 1.2–1.6 s). We measure up to 3.8M op/s for Narwhal-Bullshark and up to 382k op/s for Narwhal-Bullshark-sig. The difference in respective throughputs highlights the cost of authentication for servers: verifying signatures reduces the throughput of Narwhal-Bullshark by one order of magnitude. We observe a latency of around 3.6 s for both Narwhal-Bullshark and Narwhal-Bullshark-sig.

Chop Chop. Chop Chop achieves close to 44M op/s while running on top of both HotStuff and BFT-SMaRt. Chop Chop's latency range is 3.0–3.6 s with BFT-SMaRt and 5.8–6.5 s with HotStuff. Notably, the latency of Chop Chop-HotStuff decreases under high load. This is due to the internal batching mechanism of the HotStuff implementation: buffers fill faster under higher load, thus avoiding timeouts. This has an immediate impact on Chop Chop, which feeds HotStuff at a low rate: HotStuff alone accounts for over 60% of Chop Chop-HotStuff's overall latency. BFT-SMaRt makes a better fit for Chop Chop, as its throughput is sufficient for Chop Chop's needs, and its latency is lower than HotStuff's.

Mempools' trade-off. In comparison to BFT-SMaRt and HotStuff, Chop Chop trades latency in favor of throughput. This trade-off is mostly explained by batching and distillation. When assembling a batch, a broker has to wait twice: once to collect enough messages to fill a batch, and once to collect all multi-signatures from clients engaging in distillation. We set both waits' timeout to 1 second. Notably, Narwhal-Bullshark seems to incur a similar latency cost, as Chop Chop's latency approximately matches that of Narwhal-Bullshark, even though Chop Chop needs an extra round trip between clients and broker (Fig. 5, #4–#6).

6.4 RQ2 – Distillation Benefits

We showcase the benefits of distillation by: evaluating throughput with and without distillation, evaluating distilla-

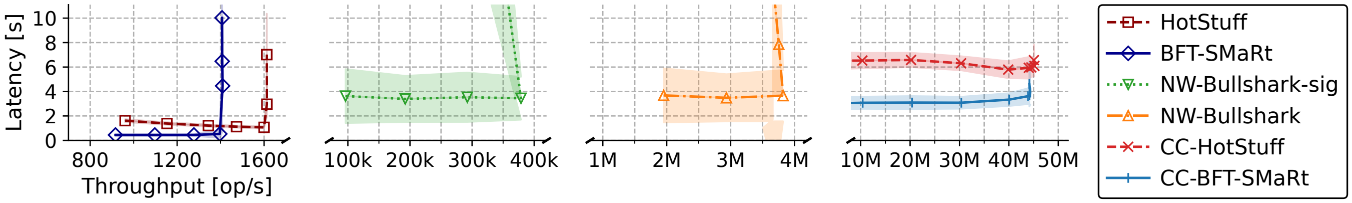
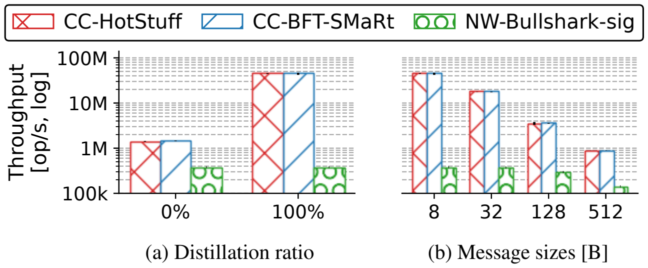


Figure 7: **Throughput-latency of Chop Chop and of notable Atomic Broadcast systems under various input rates.**



(a) Distillation ratio

(b) Message sizes [B]

Figure 8: **Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) Chop Chop has no distillation and with (b) varying message size.**

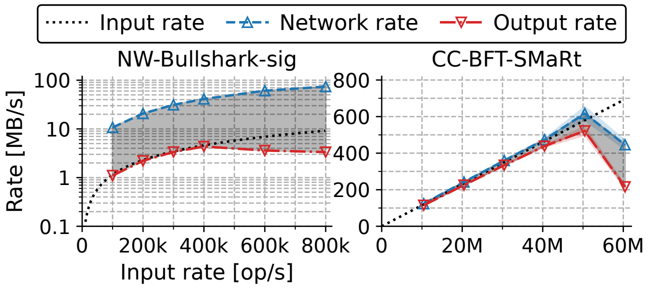


Figure 9: **Throughput efficiency of authenticated Narwhal with Bullshark (left, log scale) and Chop Chop with BFT-SMaRt (right, linear scale) with various input rates.**

tion for messages of different sizes, and observing the impact of distillation on network bandwidth to achieve line rate.

Distillation vs. mitigations. Along with distillation, Chop Chop makes use of two techniques available in the literature to mitigate the cost of Atomic Broadcast's authentication: short identifiers and pooled signature verification (see §2.2).

Fig. 8a breaks down Chop Chop's throughput, measuring how significantly distillation alone contributes to Chop Chop's performance. When no message is distilled, Chop Chop's servers bottleneck at 1.5M op/s, 3.9× higher than Narwhal-Bullshark-sig. This result is in line with both systems bottlenecking on server CPU, as the technique employed by Chop Chop to mitigate authentication complexity has only one third of the servers verify each client signature. (We conjecture that the additional 1.3 factor may be owed to engineering differences.) When batches are fully distilled, Chop Chop's throughput grows to 44M op/s, accounting for the additional 29-fold boost to Chop Chop's performance.

Distillation for larger messages. Fig. 8b illustrates Chop Chop's maximum throughput for message sizes of 8 B to 512 B which may be relevant to applications that cannot work around smaller message sizes, e.g., many smart contracts. Chop Chop's throughput is similar with BFT-SMaRt and HotStuff, decreasing at an approximately 1-to-1 ratio as the message size increases: 44.3M op/s for 8 B, 17.6M op/s for 32 B, 3.5M op/s for 128 B and 890k op/s for 512 B.

This is in line with expectations. As we discuss in §3.2, a server should receive $\sim b$ bytes in order to deliver a b-bytes message in a large, fully distilled batch, as full distillation amortizes to zero the communication cost of authenticating and sequencing each message. For 8 B messages, servers encounter a CPU bottleneck slightly before the link between load brokers and servers is saturated. This explains why the throughput decreases only 2.52× when messages grow to 32 B: all remaining server-bound bandwidth is used to convey messages (as messages are larger) while the load on server CPUs is reduced (as less messages are delivered overall). The system remains communication-bottlenecked as the size of the messages increases, and throughput starts decreasing linearly with message size, e.g., Chop Chop's throughput for 512 B messages is 4.00× smaller than for 128 B.

By contrast, Narwhal-Bullshark-sig bottlenecks on server CPUs longer, due to signature verification, maintaining a stable throughput until 512 B messages finally fill server links. Overall, Narwhal-Bullshark-sig's throughput only decreases from 382k op/s for 8 B messages to 142k op/s for 512 B messages, which matches their non-authenticated evaluation with 512 B messages. The gap between Chop Chop and Narwhal-Bullshark-sig at 512 B messages can be mostly attributed to Chop Chop's more efficient use of server bandwidth: unlike Narwhal, Chop Chop offloads the dissemination of batches to external brokers. Narwhal's use of worker-to-worker communication in its common path also makes it more prone to be affected by AWS's various upload limitations, e.g., AWS upload bandwidth is half the stated download bandwidth, and there are network credit limits for "burst" uploading.

Line rate. Fig. 9 illustrates Chop Chop's near line-rate network use by depicting its input, network and output rates:

- Input rate measures the total bytes of useful information—i.e., client identifiers and messages—that clients, load clients and load brokers all broadcast per time unit;

- Network rate measures the ingress bandwidth of servers at their network interface, i.e., useful information captured by the input rate as well as the Atomic Broadcast's overhead for ordering, authentication and deduplication;
- Output rate, or "goodput", measures the total bytes of useful information that each server delivers per time unit.

A system with perfect line rate would match all three rates: input rate would match output rate as messages can be delivered in a timely fashion with no backlogging, and output rate would match network rate as a server would only receive useful information, with no overhead due to Atomic Broadcast. The gray-shaded areas in Fig. 9 highlight this overhead, i.e., the difference between network and input rates. Network and output rates are averaged over all servers.

In this experiment, each of the 257M simulated clients broadcast 8 B messages. This results in 11.5 B of useful information per broadcast as 28 bits = 3.5 B are sufficient to represent every identifier. This conversion is captured by the dotted line which converts the input rate from op/s, represented on the x-axis, to B/s, represented on the y-axis.

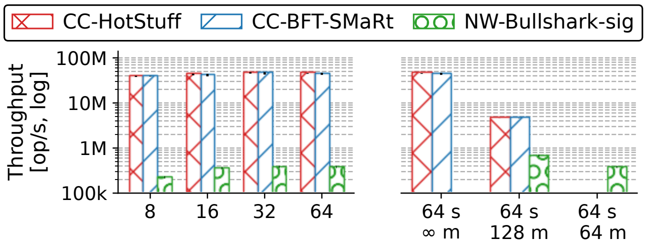
For authenticated Narwhal-Bullshark, the output rate closely matches the input rate until signature verification becomes the bottleneck at 378k op/s, shown by the plateauing output rate. The gap between Narwhal-Bullshark-sig's network and input rates is evident, differing by one order of magnitude (notably in line with our back-of-the-envelope calculation in §3.2). In contrast, thanks to distillation, Chop Chop practically achieves line-rate up to its maximum throughput. Before its inflection point at 40M op/s, the overhead of Chop Chop is less than 8%. The drop in output and network rates at 60M op/s is due to servers surpassing their computational capacity: broadcasts stall, server witness verification gets backlogged and brokers, suspecting server faults, ask for more batch witnesses, further stressing servers' CPUs.

6.5 RQ3 – Number of Servers

Fig. 10a illustrates the maximum throughput for systems of 8 ($f = 2$), 16 ($f = 5$), 32 ($f = 10$) and 64 ($f = 21$) servers. For Chop Chop, we adjust the witnessing margin as the system grows by 0, 1, 2, and 4 for 8, 16, 32 and 64 servers respectively (see § 6.2). Both Chop Chop and authenticated Narwhal-Bullshark scale well to 64 servers. Note that, unless trust assumptions are modified, Narwhal-Bullshark-sig only scales vertically: if a Narwhal server or any of its workers are faulty, the entire server group is compromised. Chop Chop, instead, scales horizontally with the number of brokers.

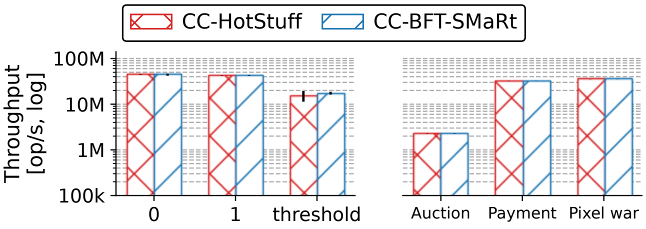
6.6 RQ4 – Overall Efficiency

The center cluster of bars in Fig. 10b compares Chop Chop's throughput with that of authenticated Narwhal-Bullshark when overall hardware resources are matched. In this setting, both systems have 128 machines at their disposal. Chop Chop is provided with 64 servers, 64 brokers and 0 load brokers. Since a load broker uses pre-generated synthetic data to simulate tens of brokers (see §6.2), involving load brokers in this experiment would give an unfair advantage to Chop Chop. Narwhal-Bullshark-sig is provided with 128 workers, to match Chop Chop's total machines, balanced across 64 server groups, to match Chop Chop's servers. The left and right clusters of bars depict Chop Chop using load brokers and Narwhal-Bullshark-sig with 64 server groups containing 1 worker each, respectively, as in the other experiments.



(a) System sizes (b) Matching resources

Figure 10: **Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) varying system size, and when (b) varying the number of overall machines ("m") with 64 servers ("s").** Load brokers in Chop Chop simulate tens of brokers, hence are noted "∞ m".



(a) Server failures (b) Applications

Figure 11: **Throughput of Chop Chop (log scale) with (a) various server failures and for (b) different applications.**

We observe 4.6M op/s for Chop Chop, with servers reporting around 5% CPU usage. We observe 679k op/s for Narwhal-Bullshark-sig. Chop Chop's higher throughput is in line with expectations. In Narwhal-Bullshark-sig, workers are trusted, and as such a worker can only contribute to its own server group. Instead, since Chop Chop brokers are untrusted, a broker's work is useful to all servers.

6.7 RQ5 – Chop Chop Under Failures

Fig. 11a depicts Chop Chop's throughput when some servers crash 30 seconds into the run. Performance drops marginally (from 44M op/s to 43M op/s) with one crash and by 66% (down to 15M op/s) when one-third of the servers crash, resulting in less CPU globally available to witness batches.

Fig. 8a captures Chop Chop's performance hit when clients fail to engage in distillation. This could be caused by clients being slow or crashed, or brokers being malicious. Under the most extreme conditions, where no client engages in distillation, the throughput drops from 44M op/s to 1.5M op/s.

6.8 RQ6 – Application Use Cases

Fig. 11b depicts the maximal stable throughput for various application use cases. In the Auction app, a client can bid an amount on a token it does not own, or take the highest offer it received for an item it owns. The highest amount bid on each token is locked and cannot be used to bid elsewhere. Money bid is transferred when the owner of the token takes the offer, or refunded when the bid is raised by another client. The Auction app is single-threaded and many clients bid on the same token to approximate a real auction. In the Payments app, clients choose a recipient and an amount to transfer. In Pixel war, clients choose a pixel and an RGB color to paint on a 2,048 by 2,048 board. Operations are generated at random.

We observe 2.3M op/s for the Auction, 32M op/s for Payments and 35M op/s for Pixel war. The bottleneck is the application in all cases, thus Chop Chop has sufficient capacity for high, single-application throughput. Chop Chop can also support many separate high-throughput applications simultaneously, making it a fitting Atomic Broadcast candidate to power a universal SMR system, i.e., an Internet computer.

7 Related Work

We overview below the state-of-the-art most relevant to Chop Chop, namely Atomic Broadcast systems with high-throughput and efficient signature aggregation schemes.

High-throughput Atomic Broadcast. Narwhal [26] is a mempool protocol that separates the reliable distribution of payloads from the communication-expensive ordering in order to accelerate DAG-based Atomic Broadcast [33, 42, 69]. Narwhal utilizes trusted workers to increase throughput while Chop Chop relies on *trustless* brokers, for the same effect, and scales out more efficiently. To circumvent the bottleneck associated with the broadcast leader, approaches using multiple leaders have been developed—both for crash [31, 61] and arbitrary [3, 6, 70, 71] faults—to scale the broadcast throughput linearly with the number of leaders. Dissemination trees [44, 63] have also been employed to reduce communication cost and maximize network bandwidth utility, while sharded [46, 76] and federated [54] approaches reduce communication cost by promoting local communication in geo-distributed setups. In comparison, Chop Chop shows that an optimal distillation mechanism for batches achieves better performances without adding complexity to the Atomic Broadcast protocol itself.

Other approaches have shown that the underlying hardware of servers can also be exploited for higher throughput, such as FPGA [38, 41] and Intel SGX enclaves [7]. In comparison, Chop Chop uniquely boosts throughput by exploiting *trustless hardware* via brokers. Atomic Broadcast can also be accelerated in data centers by using the topology of the network [51, 64] or even by running within the network itself using P4-programmable switches [27, 45]. In such low latency environments, the processing overhead incurred by the operating system kernel can be bypassed to further increase the throughput of Atomic Broadcast [1, 45, 75].

Signature aggregation. Aggregate signatures were first proposed to save space by compacting a large number of signatures into just one [11, 67]. Up until recently, aggregation could also save verification time but only in certain cases: either when the signatures are generated by the same signer [17, §5.1], or when the signatures are on the same message, i.e., multi-signatures [39]. In the latter case, aggregation mechanisms have been proposed to achieve constant-time verification of aggregated multi-signatures for both BLS [10] and Schnorr [57] signature schemes. In particular, multi-signatures are used in cryptocurrencies to have many servers sign the same batch of payloads [30, 44]. Servers in Chop Chop use rapidly-verifiable BLS multi-signatures [10] for that very purpose. In addition to aggregating server signatures on batches, Chop Chop's distillation mechanism also aggregates all client signatures in a batch in a way that provides constant-time verification. The theoretical scheme Draft [16] proposed signature aggregation with similar verification performances but is tailored to Reliable Broadcast. It is however unclear how Draft could be implemented as a real-world system without compromising liveness. Indeed, Draft assumes infinite memory to prevent message replay attacks which would rapidly exhaust servers' memory if deployed to match Chop Chop's maximum throughput (see §6.2). Chop Chop also aggregates client sequence numbers to significantly reduce bandwidth consumption when small messages are broadcast (Fig. 2). Chop Chop aggregates sequence numbers thanks to the ordering of and thanks to novel legitimacy proofs (see §4.2).

8 Concluding Remarks

Chop Chop's performance comes with two limitations. First, Chop Chop's high throughput makes memory management a challenge: servers fill their memory quickly if unable to garbage-collect under heavy load. Second, all servers in Chop Chop are known at startup and it is unclear if its performance would be maintained when deployed on thousands of servers. Interesting avenues of future research include sharding to achieve even higher throughput by running multiple, independent, coordinated instances of Chop Chop, and offloading more tasks to the brokers, such as public key aggregation.

Acknowledgments

We thank the OSDI '23 reviewers for their continuous involvement in the revision process. We further thank Vasileios Trigonakis for his early feedback. This work has been supported in part by AWS Cloud Credit for Research, the Hasler Foundation (#21084), and Innosuisse (46752.1 IP-ICT).

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. uBFT: Microsecond-scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS*, 102, 2010.

[3] Salem Alqahtani and Murat Demirbas. BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2021.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[5] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4), 2015.

[6] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. FnF-BFT: A BFT Protocol with Provable Performance Under Attack. In *30th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2023.

[7] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[8] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography (PKC)*, volume 3958. 2006.

[9] Alysson Bessani, Joao Sousa, and Eduardo E.P. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[10] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology – ASIACRYPT*, 2018.

[11] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Advances in Cryptology — EUROCRYPT*, 2003.

[12] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. BLS Signatures. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05, 2022. Work in Progress.

[13] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science, 2011.

[14] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology (JCrypt)*, 18(3), July 2005.

[15] Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. Chop Chop: Byzantine Atomic Broadcast to the Network Limit, 2024.

[16] Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira. Oracular Byzantine Reliable Broadcast. In *36th International Symposium on Distributed Computing (DISC)*, 2022.

[17] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch Verification of Short Signatures. *Journal of Cryptology (JCrypt)*, 25(4), 2012.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[19] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)*, 43(2), 1996.

[20] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[21] Authors' implementation of BFT-SMaRt in Java. https://github.com/bft-smart/library.

[22] Authors' implementation of HotStuff in C++. https://github.com/hot-stuff/libhotstuff.

[23] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[24] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Information and Computation (IC)*, 118(1), 1995.

[25] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[26] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[27] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking (ToN)*, 28(4), 2020.

[28] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4), 2004.

[29] Danny Dolev and Rüdiger Reischuk. Bounds on Information Exchange for Byzantine Agreement. *Journal of the ACM (JACM)*, 32(1), 1985.

[30] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for Consensus. In *29th USENIX Security Symposium (SEC)*, 2020.

[31] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient Replication via Timestamp Stability. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.

[32] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2), 1985.

[33] Adam Gagol, Damian Leundefinedniak, Damian Straszak, and Michal Swietek. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT)*, 2019.

[34] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Scaling Blockchain Consensus via a Robust Shared Mempool. In *39th IEEE International Conference on Data Engineering (ICDE)*, 2023.

[35] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[36] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[37] IBM. TCP Tuning guide. https://www.ibm.com/docs/en/linux-on-systems?topic=recommendations-network-performance-tuning.

[38] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[39] Kazuharu Itakura and Katsuhiro Nakamura. A public-key cryptosystem suitable for digital multisignatures. *NEC Research & Development*, 1983.

[40] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 https://rfc-editor.org/rfc/rfc8032, 2017.

[41] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[42] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[43] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-Fairness for Byzantine Consensus. In *Advances in Cryptology – CRYPTO*, 2020.

[44] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (SEC)*, 2016.

[45] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[46] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[47] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[48] Aptos Labs. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure, 2022. https://github.com/aptos-labs/aptos-core/blob/main/developer-docs-site/static/papers/whitepaper.pdf.

[49] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a State Machine. *SIGACT News*, 41(1), 2010.

[50] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[51] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[52] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, and Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[54] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[55] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic Quorum Systems. *Information and Computation*, 170(2), 2001.

[56] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. In *2005 International Conference on Dependable Systems and Networks (DSN)*, 2005.

[57] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. *Designs, Codes and Cryptography (DCC)*, 87(9), 2019.

[58] David Mazieres. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, 2016. https://stellar.org/papers/stellar-consensus-protocol.

[59] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*, 1987.

[60] Peyman Momeni, Sergey Gorbunov, and Bohan Zhang. FairBlock: Preventing Blockchain Front-Running with Minimal Overheads. In *Security and Privacy in Communication Networks (SecureComm)*, 2023.

[61] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[62] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free Asynchronous Byzantine Consensus with T <N/3 and O(N2) Messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[63] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[64] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[65] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying Blockchain Extractable Value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[66] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[67] Claus P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology (JCrypt)*, 4(3), 1991.

[68] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT Protocols under Fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[69] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[70] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State Machine Replication Scalability Made Simple. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[71] Chrysoula Stathakopoulou, David Tudor, Matej Pavlovic, and Marko Vukolić. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research (JSys)*, 2(1), 2022.

[72] The DFINITY Team. The Internet Computer for Geeks, 2022. https://eprint.iacr.org/2022/087.

[73] The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

[74] The MystenLabs Team. The Sui Smart Contracts Platform, 2022. https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf.

[75] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[76] Jiaping Wang and Hao Wang. Monoxide: Scale out Blockchain with Asynchronous Consensus Zones. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[77] Roger Wattenhofer. *Blockchain Science: Distributed Ledger Technology*. Inverted Forest Publishing, 2019.

[78] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[79] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[80] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[81] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galiñanes, and Bryan Ford. Flash Freezing Flash Boys: Countering Blockchain Front-Running. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2022.

[82] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[83] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-Frequency Trading on Decentralized On-Chain Exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

A Artifact Appendix

Abstract

This artifact is a full implementation of Chop Chop, an end-to-end client-server Byzantine Atomic Broadcast system leveraging a third-party—brokers—to optimize network usage and vastly reduce the computational load of authenticating client requests on the server-side while preserving safety.

Scope

This artifact can be used to validate the following claims (*provided the setup is the same*):

- End-to-end performance (Figs. 7 and 10).
- Distillation benefits (Fig. 8) and throughput efficiency (Fig. 9).
- Performance under faults (Fig. 11).

Note that reproducing the same plots (including error bands) can be prohibitively costly given the scale of the evaluation (number and hourly price of machines, see §6.2).

Contents

The artifact contains all the source code used to implement Chop Chop. We provide a docker file to set up experiments. We also provide automated scripts to extract and interpret the data as well as generate plots. Please refer to the README.md file in the repository for more details.

Hosting

The repository can be accessed through GitHub[4] (see the instructions on the README.md file).

Requirements

There are no special hardware or software requirements beyond a recent enough version of Rust (§5) and the desired network layout to evaluate. If you wish to reproduce our results exactly, please see §6.2 for the setup used.

[4] https://github.com/Distributed-EPFL/chop-chop-osdi24

Enabling Tensor Language Model to Assist in Generating High-Performance Tensor Programs for Deep Learning

Yi Zhai[1] Sijia Yang[2] Keyu Pan[3] Renwei Zhang[2]
Shuo Liu[1] Chao Liu[2] Zichun Ye[2]
Jianmin Ji[1] Jie Zhao[4] Yu Zhang[1*] Yanyong Zhang[1*]

[1]*University of Science and Technology of China*
[2]*Huawei Technologies Co., Ltd.* [3]*ByteDance Ltd.* [4]*Hunan University*

Abstract

Obtaining high-performance tensor programs with high efficiency continues to be a substantial challenge. Approaches that favor efficiency typically limit their exploration space through heuristic constraints, which often lack generalizability. Conversely, approaches targeting high performance tend to create an expansive exploration space but employ ineffective exploration strategies.

We propose a tensor program generation framework for deep learning applications. Its core idea involves maintaining an expansive space to ensure high performance while performing powerful exploration with the help of language models to generate tensor programs efficiently. We thus transform the tensor program exploration task into a language model generation task. To facilitate this, we explicitly design the language model-friendly tensor language that records decision information to represent tensor programs. During the compilation of target workloads, the tensor language model (TLM) combines knowledge from offline learning and previously made decisions to **probabilistically sample** the best decision in the current decision space. This approach allows more informed space exploration than **random sampling** commonly used in previously proposed approaches.

Experimental results indicate that TLM excels in delivering both efficiency and performance. Compared to fully tuned Ansor/MetaSchedule, TLM matches their performance with a compilation speedup of 61×. Furthermore, when evaluated against Roller, with the same compilation time, TLM improves the performance by 2.25×. Code available at https://github.com/zhaiyi000/tlm.

1 Introduction

As deep learning rapidly grows in both scale and complexity, the gap between the computational needs of deep learning workloads and the capabilities of existing computing platforms is widening. This gap underscores the imperative for low-latency execution of deep learning workloads. Solutions for low-latency execution primarily include kernel libraries provided by vendors (e.g., cuDNN [1], oneDNN [2]) and search-based tensor compilers (e.g., TVM [3], Halide [4], Tensor comprehensions [5], Flextensor [6], NeoCPU [7]). Existing deep learning frameworks (e.g., TensorFlow [8], PyTorch [9], and MXNet [10]) map the operators (e.g., convolution, matrix multiplication) in deep learning workloads to vendor-provided kernel libraries to optimize performance. Given the high development costs of vendor-provided kernel libraries, developers are increasingly turning to tensor compilers to auto-explore for tensor programs, i.e., optimized, low-level implementations of operators.

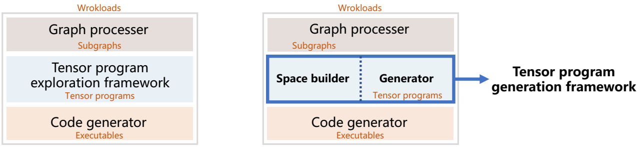


Figure 1: The left shows the architecture of common tensor compilers, while the right illustrates the structure incorporating our tensor program generation framework.

The left of Figure 1 shows that a tensor compiler can be divided into three parts: a graph processor (e.g., Relay [11], HLO), a tensor program exploration framework (e.g., AutoTVM [12], Ansor [13], MetaSchedule [14], AKG [15]), and a code generator (e.g., LLVM [16], NVCC [17]). The graph processor is responsible for converting various deep learning workloads of different formats (e.g., ONNX [18], TensorFlow PB) into a unified graph representation, performing graph optimizations, and splitting the workloads into subgraphs. Then, the tensor program exploration framework takes subgraphs as inputs, which, after scheduling, are lowered into tensor programs. Finally, a code generator is invoked to produce executables explicitly tailored for the target hardware.

The tensor program exploration framework, in lowering subgraphs to tensor programs, is tasked with making a series of critical decisions. These include determining the tiling

[*] Corresponding authors.

sizes of loop axes, setting unroll steps, choosing computation locations for operators, strategizing on parallelization and vectorization, and, particularly on GPUs, deciding thread bindings. The options for each decision create a decision space, and all decision combinations form the exploration space. The primary objective of this exploration framework is to find a decision combination that minimizes the execution latency of the resulting tensor program.

Previous work either introduces heuristic constraints to define or prune the exploration space, resulting in limited performance or constructs an expansive exploration space but employs less effective exploration strategies – for instance, the random sampling used by Ansor [13] and MetaSchedule [14]. For a detailed literature review, please refer to section §2.1.

To address this challenge, we propose an approach to maintain an expansive exploration space while conducting more powerful exploration – i.e., enabling language models to assist in generating high-performance tensor programs. Specifically, we present a *generation-based* tensor program exploration framework, i.e., a tensor program generation framework, for deep learning applications. The proposed framework consists of two components, a space builder and a generator, as shown in the right of Figure 1. The space builder is dedicated to building an expansive tensor program exploration space, ensuring high performance; meanwhile, the generator focuses on efficiently generating high-performance tensor programs, regardless of the exploration space's magnitude. They function independently without mutual constraints, each focusing solely on ensuring high performance and high efficiency.

Generating tensor program source code with language models naturally presents challenges. The length of tensor program source codes often exceeds ten thousand tokens, and these programs must adhere to strict syntactic rules. Crafting such lengthy and syntactically valid tensor program source codes is nearly unfeasible. Therefore, instead of aiming for direct end-to-end generation source code, we utilize tensor compilers for constructing tensor programs and leverage language models to assist in decision-making. To facilitate this, we explicitly design the language model-friendly tensor language to represent tensor programs. A tensor language sentence (abbreviated as tensor sentence) uniquely corresponds to a tensor program by recording the input subgraph, hardware specifications, and decision information of the tensor program. Compared to tensor program source code, a tensor sentence conveys the same semantics (i.e., both represent a tensor program) but in a far more concise manner, capped at no more than 1024 tokens. Building on the tensor language, we draw on the training methods of ChatGPT [19] to develop a language model, the tensor language model (TLM). We utilize millions of tensor sentences and a select few demonstration sentences (corresponding to high-performance tensor programs) to pre-train and fine-tune TLM in a supervised manner. After that, during the compilation of target workloads, TLM combines knowledge from offline learning and previous decisions to make probabilistic predictions for the current decision space, resulting in more effective exploration than random sampling.

It is noteworthy that, in contrast to methods like Ansor/MetaSchedule that depend exclusively on online data, TLM requires about 300K pieces of offline labeled data (i.e., tensor programs with measured execution latency) to select demonstration sentences, requiring tens of hours to collect. However, the labeled data for TLM is still significantly less than that for TenSet [20] or TLP [21], which amounts to 8.6 million and requires several weeks to collect.

In this paper, we also refer to our proposed tensor program generation framework as the TLM framework. The primary innovations of the TLM framework focus on the tensor language and tensor language model. The supported decision spaces are adapted from previous frameworks, which is largely an engineering effort. The space builder currently supports decision spaces adapted from several previous search frameworks, including Ansor, MetaSchedule, AKG, and AKG-MLIR.

We conducted extensive experiments to validate the high efficiency and performance of the TLM framework, examining scenarios with various exploration budget points. Under a limited budget, TLM's performance matches that of Ansor/MetaSchedule yet compiles 61× faster. While its compilation time aligns with Roller, its performance is 2.25× better. In ample exploration times, TLM's compilation duration is consistent with Ansor and MetaSchedule, delivering a performance boost of 1.08× and 1.04×, respectively.

In summary, this paper makes the following contributions:

- We design the language model-friendly tensor language to represent tensor programs, bridging the gap between tensor programs and language models.
- We develop a tensor language model that combines knowledge from offline learning and previously made decisions to probabilistically sample the best decision in the current decision space, enabling more effective space exploration.
- Experimental results show that TLM excels in delivering both high efficiency and performance.

2 Background

2.1 Tensor Program Exploration Framework

In the development of tensor program exploration frameworks, previous studies mainly employ search algorithms to locate optimal tensor programs automatically. As a result, the search-based tensor program exploration framework often gets dubbed as the tensor program search framework, with its exploration space used as the search space.

Earlier, the Halide auto-scheduler [4] aggressively prunes the search space by evaluating incomplete programs; AutoTVM and FlexTensor [6] employ predefined, manually written templates to define their search space. These methodologies introduce constraints that limit the search space, thereby missing out on many potent decision combinations and leading to suboptimal performance.

Subsequently, Ansor [13] and MetaSchedule [14] utilize derivation rules to build an expansive search space, pushing performance to state-of-the-art levels. However, this came with a notably vexing issue — excessive compilation time. In our experience, using Ansor to bring a BERT-base workload to convergence takes 21.6 hours on the NVIDIA V100 GPU; on the Intel i7-10510U CPU, it requires 13.1 hours. The reason behind this is that Ansor/MetaSchedule utilizes a strategy of random sampling followed by evaluation using a learnable cost model. Until completing the random sampling of all decision spaces, Ansor/MetaSchedule lowers subgraphs to tensor programs and then extracts statistical features, including computation, memory access, and arithmetic strength for performance evaluation with the learnable cost model. However, this strategy has three main issues:

- Random sampling represents a form of inefficient exploration, with equal probabilities of sampling optimal or suboptimal decisions.
- The evaluation demonstrates hysteresis. Decisions made during the initial sampling might lead to poor performance but can only be ascertained when evaluating.
- The inadequacy of training data and the small parameter size of the cost model weaken the model's learning ability, limiting its effectiveness in guiding the search algorithm. Ansor/MetaSchedule relies exclusively on minimal online data to train cost models. Moreover, their models are primarily based on low-parameter machine learning or deep learning models (e.g., XGBoost [22], MLP, LSTM).

More recent studies, Roller [23], TenSet [20], and TLP [21], were proposed to address the issue of slow compilation. Roller speeds up the compilation by aligning tensor shapes with the properties of the hardware. However, in doing so, Roller experiences a declining performance (§6.4.2), still due to the generalizability of the heuristic constraint. TenSet and TLP collect offline datasets before compiling the target workloads to build a stronger cost model. While these approaches expedite compilation, they exhibit two key shortcomings. Firstly, they adopt the Ansor/MetaSchedule strategy of conducting random sampling followed by evaluating with a cost model. Moreover, they rely on 8.6 million pieces of labeled data.

An example. We illustrate an example to analyze the differences between heuristic compilers, exemplified by Roller, search-based frameworks like Ansor/MetaSchedule, and our generation-based TLM framework from a probabilistic perspective. Consider a decision space, D_i, with four valid decisions. The optimal decision is d_3, as depicted in Figure 2.

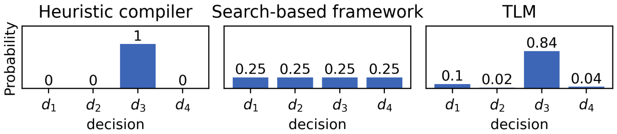


Figure 2: Example of probability distributions in a decision space for several different methods.

Heuristic compilers use heuristic constraints to eliminate d_1, d_2, and d_4. After this pruning, d_3 has a 100% chance of being sampled, reducing the search space and speeding up the compilation. However, such heuristics aren't always accurate and can sometimes degrade performance. In contrast, search-based frameworks keep all decision options and sample one decision randomly. Here, d_3 has a 25% sampling probability. Until all decision spaces are sampled, a learnable cost model evaluates the results. After sufficient searching, search-based frameworks can always approximate the optimal solution, making them performance-oriented but inefficient methods. Unlike search-based frameworks, TLM combines the knowledge learned offline with the decisions already made to make probabilistic predictions about the best decision in the current decision space.

2.2 Language Model

Deep learning language models mainly fall into two categories: the Masked Language Model (MLM) and the Causal Language Model (CLM), with CLM also known as the Autoregressive Language Model. Notable examples of CLM include the GPT series, such as GPT-2 [24] and GPT-3 [25]. CLMs, recognized for their natural text generation method, often excel in tasks that need coherent text creation, like writing or chatbot conversations. In this paper, a language model refers specifically to a CLM.

Before training a language model, a vocabulary is created through tokenization. Tokenization typically refers to fragmenting an input sentence into its constituent tokens for subsequent language analysis or as input to a model, with all such tokens collectively forming a vocabulary. The steps ①② and ⑦⑧ in Figure 3 represent the processes of tokenizing a sentence into tokens and converting tokens back into a sentence, respectively. For simplicity in description, here we tokenize by words (in practice, the process is more complex). The only thing language models perform is to combine the learned knowledge with the given input to predict the probability distribution of the next token being a specific one from the vocabulary. The initial input is referred to as a prompt. During the training of a language model, natural language sentences are utilized as input. Through the backpropagation algorithm [26], the language model's parameters are iteratively updated to maximize the probability that the next token generated aligns with the corresponding token within the natural language.

Employing the language model for inference, take a real-life instance as an example: we ask a language model, "What

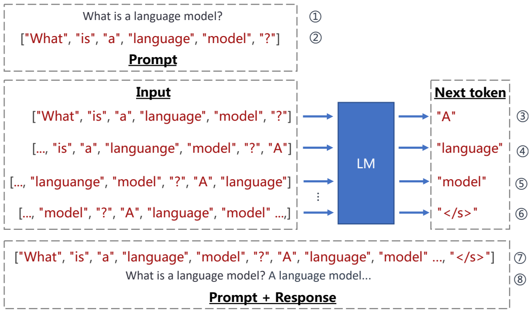

What is a language model?
["What", "is", "a", "language", "model", "?"]
Prompt
①
②
Input
["What", "is", "a", "language", "model", "?"]
[..., "is", "a", "languange", "model", "?", "A"]
[..., "languange", "model", "?", "A", "language"]
[..., "model", "?", "A", "language", "model" ...,]
LM
Next token
"A"
"language"
"model"
"</s>"
③
④
⑤
⑥
["What", "is", "a", "language", "model", "?", "A", "language", "model" ..., "</s>"]
What is a language model? A language model...
Prompt + Response
⑦
⑧

Figure 3: Employing a language model to generate a natural language sentence.

is a language model?" and it responds, "A language model is a type of artificial intelligence that is trained to understand, generate, and respond to human language...". The generation process is illustrated in Figure 3. In step ③, the language model uses the prompt as input to predict a probability for each token in the vocabulary. This probability signifies the likelihood of each token logically following the input. The next token is then sampled based on these probabilities. Step ④ involves using the prompt and the token generated in step ③ as input to sample the next token probabilistically, continuing this process until the "</s>" token is generated. The ending "</s>" signifies the end of a sentence.

2.3 ChatGPT/InstructGPT

ChatGPT [19] and InstructGPT [27] employ a similar training methodology that encompasses: pre-training, SFT, and RLHF (RM + RL). TLM partially adopts these training approaches. The training process for InstructGPT includes:
Pre-training GPT-3. InstructGPT is based on GPT-3, which boasts 175 billion parameters, leverages approximately 45TB of Internet text for training, and necessitates a significant amount of hardware resources throughout its training process, with the aim of enabling GPT-3 to learn the fundamental structures and semantics of language.
Supervised fine-tuning (SFT). Training a supervised policy by collecting demonstration data (around 14K entries) to fine-tune GPT-3 through supervised learning, with the demonstration data provided by humans representing the desired output behavior corresponding to a prompt.
Reward modeling (RM). Training a reward model by collecting comparison data (around 51K entries), wherein the comparison data, also provided by humans, offers a ranking of model outputs from best to worst.
Reinforcement learning (RL). Optimizing a policy against the reward model using the PPO [28] reinforcement learning algorithm, with rewards determined by the preceding reward model, this step will be iterated multiple times.

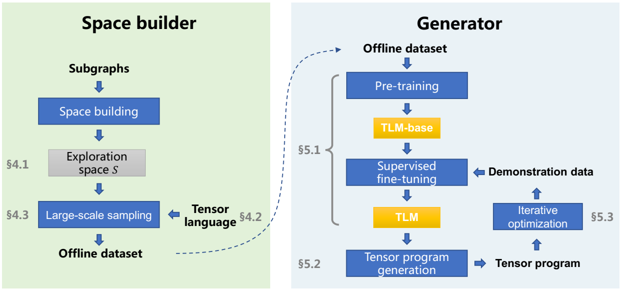


Figure 4: System overview of the TLM framework.

3 System Overview

The TLM framework is a tensor program generation framework designed to generate high-performance tensor programs efficiently. Maintaining a large exploration space can ensure that high-performance tensor programs are not eliminated by heuristic constraints. However, this requires stronger exploration capabilities. Utilizing deep learning models to glean knowledge from offline data for aiding online tensor program exploration presents a promising strategy. Given the current learning capabilities, language models are the most powerful method available. Hence, we propose to leverage the language model to assist in generating high-performance tensor programs, transforming the tensor program exploration task into a language model generation task.

Figure 4 shows the system overview and marks the corresponding subsections that detail each system component. Section 4 centers on collecting a large-scale offline dataset necessary for the pre-training of TLM. Section 4.1 details the exploration space for data sampling and its build process, providing a theoretical foundation for tensor language design through formalization of the exploration space. Section 4.2 discusses tensor language design, emphasizing its role in preserving tensor programs sampled from the exploration space in a format more amenable to language models. Following the discussion of the sampling space and preservation methods, Section 4.3 introduces large-scale sampling, where extensive random sampling achieves an unbiased estimation of the exploration space.

Section 5 focuses on the development of a tensor language model. Initially, Section 5.1 addresses the model architecture, training methods, and the training process of TLM, encompassing both pre-training and supervised fine-tuning. Upon completing pre-training, TLM should be able to generate any valid tensor program within the exploration space. Section 5.2 then details how the TLM framework generates tensor programs with decision-making support from TLM. For generating high-performance tensor programs, we employ demonstration data (corresponding to high-performance tensor programs) to fine-tune TLM through supervised learning, aligning the tensor sentences it generates with our anticipated

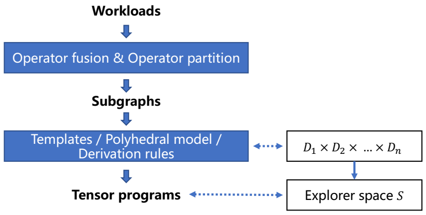

Workloads
Operator fusion & Operator partition
Subgraphs
Templates / Polyhedral model / Derivation rules
$D_1 \times D_2 \times \ldots \times D_n$
Tensor programs
Explorer space S

Figure 5: Two intermediate representations of common tensor compilers.

demonstration sentences. Section 5.3 discusses obtaining this demonstration data via iterative algorithms.

4 Space Builder

The function of a space builder is to create a large exploration space to ensure high performance. This exploration space consists of all possible tensor programs. Regarding the idea that a large exploration space can guarantee high performance, two aspects need to be discussed. First, a larger space implies that, in comparison to a smaller one, it is constructed with fewer constraints, thus enabling the generation of more tensor programs. To some extent, it can be said that the large space includes the smaller space. Second, ensuring high performance means maintaining the potential for high-performance tensor programs to exist within the exploration space. The larger the space, the less likely it is to be pruned by constraints, but this also demands greater exploration capabilities.

4.1 Exploration Space

In this section, we introduce two primary topics. The first is about building the exploration space. The second involves formalizing the tensor program generation process, providing a theoretical basis for designing the tensor language.

Tensor compilers customarily extract two intermediate representations (IRs) for optimization purposes, i.e., the graph layer and the tensor layer, which are specifically tailored for hardware-independent and hardware-dependent enhancements, respectively. As depicted in Figure 5, the graph layer ingests a workload, optimizing with several passes and employing algorithms for operator fusion and partitioning. Then, it produces subgraphs, each consisting of one or more operators. A subgraph describes the expected computational results, whereas the tensor program exploration framework transforms this subgraph into a tensor program, providing a detailed computational implementation. The collective of all possible tensor programs forms the exploration space S of the tensor layer. Previous exploration frameworks utilize expert knowledge, including templates (AutoTVM), the polyhedral model (AKG), and derivation rules (Ansor, MetaSchedule), to map subgraphs into tensor programs. For areas where expert knowledge falls short, these frameworks use tunable parameters (AutoTVM, AKG), annotations (Ansor), or random variables (MetaSchedule) to create a decision space D_i and then employ search algorithms, such as simulated annealing [29] and genetic algorithm [30], to locate the optimal solution.

The decision spaces chiefly involve determining tiling sizes for loop axes, setting unroll steps, selecting computation locations for operators, strategizing parallelization and vectorization, and on GPUs, specifically determining thread bindings.

From the analysis above, tensor programs result from optimizing input subgraphs through a series of decisions. We formalize the tensor layer's exploration space as follows:

$$S = \left\{ s^{(n)} \;\middle|\; \begin{array}{l} s^{(i)} = \text{apply}\left(s^{(i-1)}, d_i\right), \\ \forall d_i \in D_i,\ 1 \le i \le n \end{array} \right\}$$

where $s^{(0)}$ denotes the initial program of the input subgraph, and d_i represents a random sample from the set D_i. Thus, the size of the exploration space aligns with the number of decision combinations. We have:

$$|S| = |D_1| \times |D_2| \times \ldots \times |D_n|.$$

This work discusses only the exploration space of the tensor layer. On a CPU, the exploration space size corresponding to a subgraph is roughly 10^6; on a GPU, it approximates 10^9.

We reuse the decision space from previous work (e.g., Ansor, MetaSchedule), as we think that this space is already sufficiently large to yield high-performance tensor programs. Expanding the exploration space of tensor layers to larger is a considerable challenge. In future endeavors, we advocate exploring larger spaces mainly by integrating the decision space of the graph layer. Certainly, the TLM framework supports exploration spaces of varying sizes since they all integrate with TLM in the same manner.

4.2 Tensor Language

In this section, we focus on how to store tensor programs sampled from the exploration space.

Recall that our objective is to teach language models of these tensor programs, enabling them to aid in generating high-performance tensor programs during target workload compilation. Tensor program source codes often exceed ten thousand tokens, posing a challenge for language models to generate such lengthy, coherent, and valid source codes. Therefore, we do not pursue the end-to-end generation of tensor program source codes. Instead, we utilize language models to assist in decision-making. To facilitate this, we explicitly design the language model-friendly tensor language that records decision information to represent tensor programs.

As understood from Section 4.1, a tensor program $s^{(n)}$ pertains exclusively to the initial program $s^{(0)}$ of the input subgraph and decisions $d_1, d_2, \ldots, d_n$. Each decision d_i is randomly sampled from decision space D_i, the design of which

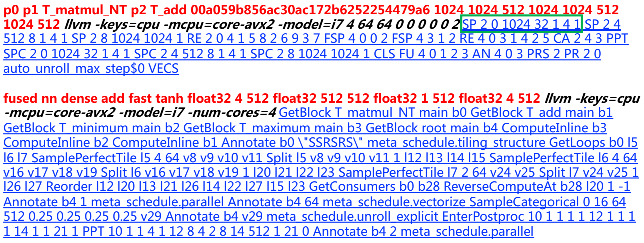
Algorithm 1: Sampling tensor sentences from decision spaces.

```
Func GenerateSampleData(subgraph, hardware):
    tokens = []
    ExtractTokensFromSubgraph(subgraph, tokens)
    ExtractTokensFromHardware(hardware, tokens)
    decision_spaces = DetermineDecisionSpaces(subgraph,
      hardware)
    foreach space in decision_spaces do
        switch space.type do
            case "tile_size" do
                HandleTileSizeSpace(space, tokens)
            case "unroll" do
                HandleParallelSpace(space, tokens)
            // Additional space types
            case ... do
                ...
    return tokens
Func HandleTileSizeSpace(space, tokens):
    tokens.append("split")
    tokens.extend(Serialize(space.operator))
    tokens.extend(Serialize(space.axis))
    tiles = RandomSample(space)
    tokens.extend(Serialize(tiles))
    // Other properties
```

hinges on the hardware platform. Consequently, to ensure that a tensor language sentence uniquely corresponds to a tensor program, a tensor sentence must encapsulate the input subgraph, hardware specifications, and decisions. We utilize Algorithm 1 to sample data from the exploration spaces. Each sentence extracts information from the input subgraph, encompassing the type and shape information of each operator within the subgraph. Furthermore, it retrieves hardware details, including the number of processing cores and the supported vector instructions. Successively extracting information from each decision space, Algorithm 1 demonstrates how to extract tokens from the tile size decision space, conserving details such as the corresponding operator, axis, decision space type, sampled decisions from the space, and other pivotal information. A similar method is applied to other decision spaces as well.

Tensor language is a form of natural language, not a programming language, and thus does not strictly follow the Backus-Naur Form [31]. Its primary intent is to represent a tensor program using a single sentence, emphasizing its role in recording rather than programming. This recording process offers significant **flexibility**, with only one constraint being the **consistency** between tensor sentences collected offline and generated online. Such consistency is essential for deep learning models to ensure that training and testing data are independently and identically distributed.

Given the flexibility of tensor language, there are no strict guidelines on the exact details that need to be recorded about input subgraphs, hardware specifications, and decision information. Under the premise of maintaining consistency, these details can be dynamically adjusted in conjunction with specific engineering projects. Figure 6 showcases tensor sentence samples tailored for Ansor (top) and MetaSchedule (bottom).



p0 p1 T_matmul_NT p2 T_add 00a059b856ac30ac172b6252254479a6 1024 1024 512 1024 1024 512 1024 512 ***llvm -keys=cpu -mcpu=core-avx2 -model=i7 4 64 64 0 0 0 0 0 2*** SP 2 0 1024 32 1 4 1 SP 2 4 512 8 1 4 1 SP 2 8 1024 1024 1 RE 2 0 4 1 5 8 2 6 9 3 7 FSP 4 0 0 2 FSP 4 3 1 2 RE 4 0 3 1 4 2 5 CA 2 4 3 PPT SPC 2 0 1024 32 1 4 1 SPC 2 4 512 8 1 4 1 SPC 2 8 1024 1024 1 CLS FU 4 0 1 2 3 AN 4 0 3 PRS 2 PR 2 0 auto_unroll_max_step$0 VECS

fused nn dense add fast tanh float32 4 512 float32 512 512 float32 1 512 float32 4 512 ***llvm -keys=cpu -mcpu=core-avx2 -model=i7 -num-cores=4*** GetBlock T_matmul_NT main b0 GetBlock T_add main b1 GetBlock T_minimum main b2 GetBlock T_maximum main b3 GetBlock root main b4 ComputeInline b3 ComputeInline b2 ComputeInline b1 Annotate b0 \"SSRSRS\" meta_schedule.tiling_structure GetLoops b0 l5 l6 l7 SamplePerfectTile l5 4 64 v8 v9 v10 v11 Split l5 v8 v9 v10 v11 1 l12 l13 l14 l15 SamplePerfectTile l6 4 64 v16 v17 v18 v19 Split l6 v16 v17 v18 v19 1 l20 l21 l22 l23 SamplePerfectTile l7 2 64 v24 v25 Split l7 v24 v25 1 l26 l27 Reorder l12 l20 l13 l21 l26 l14 l22 l27 l15 l23 GetConsumers b0 b28 ReverseComputeAt b28 l20 1 -1 Annotate b4 1 meta_schedule.parallel Annotate b4 64 meta_schedule.vectorize SampleCategorical 0 16 64 512 0.25 0.25 0.25 0.25 v29 Annotate b4 v29 meta_schedule.unroll_explicit EnterPostproc 10 1 1 1 1 12 1 1 1 1 14 1 1 21 1 PPT 10 1 1 4 1 12 8 4 2 8 14 512 1 21 0 Annotate b4 2 meta_schedule.parallel

Figure 6: Tensor sentence samples tailored for Ansor (top) and MetaSchedule (bottom), encompassing **input subgraph**, ***hardware specifications***, and decision information. For example, at the top, "SP 2 0 1024 32 1 4 1" represents "split operator_index axis_index axis_extent tile_size_0 tile_size_1 tile_size_2 save_manner".

4.3 Large-scale Sampling Tensor Sentences

This section introduces employing the sampling algorithm to create a large-scale offline dataset.

Large-scale sampling serves two main purposes. Firstly, it is to create an unbiased estimation of the exploration space S through widespread random sampling, which allows TLM to learn the basic structures and semantics of tensor language, enabling TLM to generate any tensor sentence within space S. Secondly, it is to build as large a vocabulary (§2.2) as possible, where all decision options like "i.0=16" and "i.0=32" are tokenized into discrete tokens. If a decision, such as "i.0=17", is not in the vocabulary, it will never be generated.

The workload dataset configured for TLM takes cues from TenSet. The workloads are derived from PyTorch's Vision Model Zoo and Huggingface's Transformer Model Zoo, encompassing tasks emblematic of both computer vision (CV) and natural language process (NLP). We adjust the input shape to generate a variety of subgraphs. Note that we focus on small batch sizes in this dataset because tensor compilers are mainly used for optimizing trained models for inference. Altogether, the dataset consists of 138 workloads, with 12 held out for testing; from the remaining approximately 3K subgraphs are extracted.

We collect 2 million tensor sentences for 3K subgraphs to pre-train TLM. It is worth noting that the 2 million data entries are distinct from the 8.6 million used in TenSet/TLP. The data here is unlabelled, i.e., it does not require measuring execution latency; measurement is typically the most time-consuming part. On a 96-core server, it takes about 2 hours to collect 2 million CPU data entries, and around 10 hours for the GPU. This longer duration for GPU data is due to additional checks, such as ensuring thread binding meets hardware constraints.

5 Tensor Language Model

5.1 Model Details

Model architecture. Taking into account both learning capacity and resource overhead, TLM adopts the architecture of GPT-2 Small, which encompasses approximately 100 million parameters. TLM is composed of 12 Transformer layers, each featuring 12 attention heads and 768 hidden units.

Training methodology. The TLM training is partially inspired by the training procedure utilized by ChatGPT, as introduced in §2.3. To put it succinctly, we modify the formula from $1 \times \text{pre-training} + 1 \times \text{SFT} + n \times (\text{RM} + \text{RL})$ to $1 \times \text{pre-training} + n \times (\text{measurement} + \text{SFT})$.

We substitute reward model (RM) with measurement. RM allocates a reward value to the model output and evaluates its quality. Here, tensor language has a natural advantage. A tensor sentence can be converted into a tensor program (§5.2), allowing its execution latency to be directly measured on hardware, with a lower latency suggesting a better tensor sentence. While ChatGPT only performs supervised fine-tuning (SFT) once, we conduct it multiple times. The input for SFT is demonstration data, provided by humans, symbolizing the desired output behavior corresponding to a prompt. In the NLP field, it's hard to say which demonstration data is the "best" for a prompt, but this is achievable in tensor language. That is, the sentence with the lowest execution latency is the "best" for its prompt. Performing measurement and SFT multiple times is consistently seeking the best tensor sentences (§5.3). We abstained from using reinforcement learning (RL) mainly because it requires adjusting various hyperparameters and presents a training challenge to convergence, and we find the performance was sufficiently good after performing SFT multiple times.

Training details. The model resulting from pre-training is termed TLM-base, and TLM is derived by performing SFT on TLM-base. We pre-train TLM-base using the offline dataset gathered in §4.3 in the same manner as pre-training other language models. It is noteworthy that TLM-base converges within just 2 epochs. This quicker convergence is due to the more pronounced regularity in tensor language compared to the broad diversity in natural languages. Among the 2 million data entries, the input subgraph types, hardware types, and decision types are all limited. Pre-training TLM-base for 2 epochs takes about 10 hours using 4 NVIDIA V100s.

Following its pre-training, TLM-base possesses the ability to generate valid tensor sentences. Furthermore, for the input subgraphs, we expect that TLM-base can aid in generating high-performance tensor programs. To this end, we employ demonstration data to fine-tune TLM-base through supervised learning. Demonstration data refers to the tensor sentences corresponding to a small subset of tensor programs with the lowest execution latency for a given input subgraph. **The purpose of SFT of a language model with demonstration data is to achieve that the model's responses to prompts align with the human intentions reflected in the demonstration data.** Similarly, we apply demonstration data to TLM, aiming to empower it to generate high-performance outputs, in response to prompts. Performing SFT on TLM-base requires a small batch (about 3K, selecting the best one for each subgraph) of demonstration data. We employ iterative optimization (§5.3) to continuously optimize these demonstration entries. With 3K demonstration data, using 4 NVIDIA V100s to perform SFT on TLM-base once takes approximately 10 minutes.

Algorithm 2: Generating tensor programs aided by TLM in decision-making.

```
Func GenerateTensorProgram(subgraph, hardware):
    tokens = []
    ExtractTokensFromSubgraph(subgraph, tokens)
    ExtractTokensFromHardware(hardware, tokens)
    program = GetInitProgram(subgraph, hardware)
    decision_spaces = DetermineDecisionSpaces(subgraph,
      hardware)
    foreach space in decision_spaces do
        switch space.type do
            case "tile_size" do
                ApplyTileSize(space, tokens, program)
            case "unroll" do
                ApplyParallel(space, tokens, program)
            // Additional space types
            case ... do
                ...
    return program
Func ApplyTileSize(space, tokens, program):
    tokens.append("split")
    tokens.extend(Serialize(space.operator))
    tokens.extend(Serialize(space.axis))
    response_tokens = TLM(tokens)
    tiles = ConvertTokensToTiles(response_tokens)
    if not CheckValidTiles(space, tiles) then
        raise Exception("Invalid Tensor Program")
    tokens.extend(Serialize(tiles))
    program.apply(space.operator, space.axis, tiles)
    // Other properties
```

5.2 Tensor Program Generation

The TLM framework utilizes TLM for decision-making during the tensor program generation process, as detailed in Algorithm 2. The steps in Algorithm 2 align with those in Algorithm 1, adhering to the consistency required for generating tensor sentences. After TLM generates a decision, the framework checks its validity within the decision space. If the decision is invalid, the framework discards the tensor program and initiates regeneration. Since the decision is obtained through sampling, the next generation might sample a different decision, preventing continuous failure in regeneration. Fortunately, following pre-training and fine-tuning,

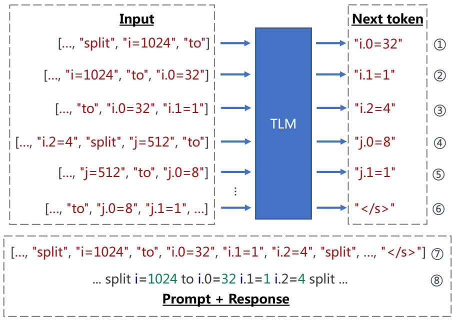

Input
[..., "split", "i=1024", "to"]
[..., "i=1024", "to", "i.0=32"]
[..., "to", "i.0=32", "i.1=1"]
[..., "i.2=4", "split", "j=512", "to"]
[..., "j=512", "to", "j.0=8"]
[..., "to", "j.0=8", "j.1=1", ...]
TLM
Next token
"i.0=32" ①
"i.1=1" ②
"i.2=4" ③
"j.0=8" ④
"j.1=1" ⑤
"</s>" ⑥
[..., "split", "i=1024", "to", "i.0=32", "i.1=1", "i.2=4", "split", ..., "</s>"] ⑦
... split i=1024 to i.0=32 i.1=1 i.2=4 split ... ⑧
Prompt + Response

Figure 7: Generating a tensor sentence for a matrix multiplication operator with dimensions $m = 1024$, $n = 512$, and $k = 1024$, with the tiling size component depicted therein.

we observe that invalid decisions are exceedingly rare. On average, generating a valid tensor program necessitates no more than 1.1 calls to Algorithm 2.

Figure 7 illustrates the process of *response_tokens* = `TLM`(*tokens*) in Algorithm 2 for a matrix multiplication operator with dimensions $m = 1024$, $n = 512$, and $k = 1024$, with the tiling size component depicted therein. This tensor sentence matches the one at the top of Figure 6 and is presented here (with the operator's index omitted) in a more human-readable format. In Step ①, known information—including input subgraph, hardware specifications, and "split i=1024 to" (corresponds to $m = 1024$) — serves as the input prompt to TLM. TLM then predicts the probability distribution of the next token based on this prompt, subsequently choosing a token, assumably "i.0=32", via probabilistic sampling from the distribution. In Step ②, the prompt and the predicted next token from Step ① are combined to formulate a new input for TLM to forecast the next token. Steps ③ replicates Step ②. After completing the three steps, TLM finalizes tile sizes for the *i*-axis and returns to Algorithm 2. When necessary to generate tile sizes for the *j*-axis, TLM will be invoked again. In Step ④, the input from Step ③, its predicted next token, and "split j=512 to" (corresponding to $n = 512$) are merged into a new input, which is then input into TLM to predict the next token. TLM can but does not employ the input graph and hardware specifications as a prompt to generate all decision information in one go. Instead, the framework repeatedly invokes TLM within Algorithm 2, generating only a subset of decisions each time. This granular approach efficiently filters out invalid data, enhancing TLM availability and stability.

5.3 Iterative Optimization

Fine-tuning TLM-base using demonstration sentences is crucial for its operation. This section focuses on the methods for acquiring a batch of demonstration data.

Throughout acquiring demonstration data, measurement is the most time-consuming step and lies on the critical path, acting as the bottleneck of the process. To address this, we develop a pipeline system that executes the iterative optimization algorithm for collecting demonstration data. This system maximizes the utilization of the measurement hardware by employing two separate processes: one for executing SFT and another for measurement, with the former's execution time controlled to be shorter than the latter, ensuring continuous measurement. When operating on GPUs, these processes operate on separate GPU cards.

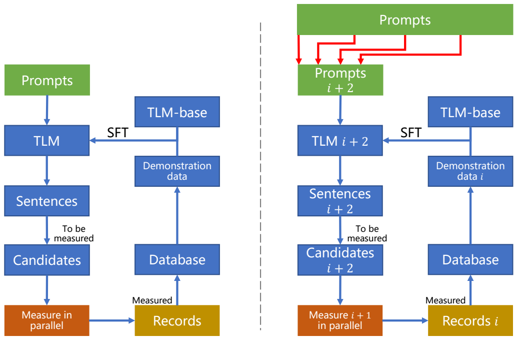


Figure 8: Flowchart of the iterative optimization process.

Figure 8 shows a flowchart of the iterative optimization on the left and a flowchart incorporating a pipeline on the right. We split (the text color matching that in Figure 8) all subgraphs (with each subgraph corresponding to one prompt) in the workload dataset into k_b (e.g., 4) batches and then cyclically select each batch. Upon the completion of the measurement of batch *i* (referred to as records *i*), the optimal batch of data is extracted from the database, noted as demonstration data *i*, and employed to fine-tune TLM-base, yielding TLM $i+2$. Subsequently, TLM $i+2$ is used to produce sentences $i+2$ (one prompt produces k_p (e.g., 16 or 32) sentences). TLM 0 and TLM 1 are replaced by TLM-base.

There are several details worth noting:

- When splitting subgraphs, we **sort them by subgraph type** (e.g., fused_nn_dense_add, fused_nn_conv2d_add_nn_relu, and fused_nn_adaptive_avg_pool2d), selecting one every k_b, based on the rationale that a superior decision often also has a certain optimizing effect on subgraphs of the same type.
- Only when TLM can generate superior tensor sentences can this iterative optimization algorithm operate normally. **So why can TLM generate more optimal data?** From a genetic perspective, when generating a sentence for a subgraph, TLM has already learned the demonstration data corresponding to the current subgraph, as well as demonstration data from other subgraphs, thereby inheriting the advantages of both itself and other subgraphs. When producing the next token for a prompt, probability sampling is used, which, owing to its stochastic nature, provides additional exploration and thus may introduce mutations. The combination of inheritance and mutation may potentially

generate higher quality data, aligning with the design philosophy of genetic algorithms.

- Demonstration refers to low execution latency. The total execution latency of all subgraphs can be expressed as $latency_{\text{total}} = \sum_{s_i \in \text{subgraphs}} \min(\text{all record latencies of } s_i)$. The latency of the demonstration data will not rise since it always selects the best from all measurements, so it decreases monotonically. Furthermore, the best latency of the demonstration data cannot be lower than its physical limit, so it is bounded. **Mathematically speaking, a monotonic and bounded limit results in convergence.** Section 6.2 of the evaluation discusses how much data needs to be measured for convergence.
- TLM is designed to assist in making decisions within a decision space, trained by a gradient descent strategy. TLM i may perform worse than TLM $i-1$ because gradient descent does not guarantee that TLM can always be trained to its optimal state. However, since TLM i is always trained from TLM-base, it remains unaffected by TLM $i-1$ and does not influence TLM $i+1$. Hence, **TLM i does not likely exhibit cumulative errors**, and the final TLM is only related to TLM-base and the final demonstration data.
- The demonstration data in Figure 8 is obtained through iterative optimization from scratch. Similarly, **there are also other methods to obtain a batch of demonstration data**. For instance, it can be obtained using other search algorithms; if there is a batch of demonstration data on Hardware A, it can be transferred to Hardware B through transfer learning; the data can be directly written using expert knowledge. The good news is that this demonstration data can still continue to use iterative optimization algorithms until convergence.
- **The iterative optimization algorithm can also be viewed as a tuning system, particularly suitable for situations where multiple workloads need to be tuned at once.** For instance, in dynamic shape scenarios, the target workload with different shapes need to be tuned simultaneously.

6 Evaluation

6.1 Experimental Settings

TLM supports several decision spaces, including those adapted from Ansor (V0.12), MetaSchedule (V0.12), AKG (V2.1), and AKG-MLIR (V0.1). In subsequent sections, these TLMs are referred to as TLM-Ansor, TLM-Meta, TLM-AKG, and TLM-AKG-MLIR. In the evaluation, we focus only on TLM-Ansor and TLM-Meta, which together are implemented in Python and C++ with about 10K lines of code.

The dataset we configured for TLM consists of 138 workloads. We hold out a test set that consists of 12 workloads, as shown in Table 1.

For the CPU experiments, we use a notebook equipped with a 4-core Intel(R) Core(TM) i7-10510U CPU supporting the AVX2 instruction set, 16GB memory, and running on Ubuntu 20.04. For the GPU experiments, we utilize a server outfitted with a 48-core Intel(R) Xeon(R) Gold 6226 CPU, 376GB memory, and four 32GB NVIDIA Tesla V100 GPUs. It runs on Ubuntu 20.04 with CUDA 11.6 and cuDNN 8.4.0.

Table 1: Workloads in the TLM test set.

Model	Input shape	Model	Input shape
ResNet-50 [32]	[1, 3, 224, 224]	DenseNet-121 [33]	[8, 3, 256, 256]
MobileNet-V2 [34]	[1, 3, 224, 224]	BERT-large	[4, 256]
ResNeXt-50 [35]	[1, 3, 224, 224]	Wide-ResNet-50 [36]	[8, 3, 256, 256]
BERT-base [37]	[1, 128]	ResNet3D-18 [38]	[4, 3, 144, 144, 16]
BERT-tiny	[1, 128]	DCGAN [39]	[8, 3, 64, 64]
GPT-2	[1, 128]	LLAMA [40]	[4, 256]

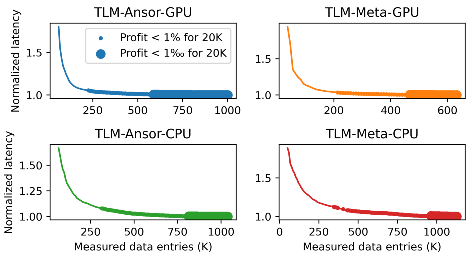


Figure 9: Demonstration data convergence curves of TLM-Ansor and TLM-Meta on the GPU and the CPU.

6.2 Convergence Behavior of Demonstration Data

This section discusses the convergence behavior of obtaining demonstration data through the iterative algorithm, which is the slowest phase of the entire pipeline. Figure 9 displays four curves corresponding to the convergence of TLM-Ansor and TLM-Meta on both the GPU and the CPU, respectively labeled as TLM-Ansor-GPU, TLM-Meta-GPU, TLM-Ansor-CPU, and TLM-Meta-CPU. These four scenarios utilize 2169, 3120, 2169, and 2657 subgraphs (extracted from 126 workloads). The horizontal axis denotes measured data entries (across all subgraphs), while the vertical axis indicates normalized total latency. We define "convergence" as the point at which 20K measurements (across all subgraphs) result in a performance improvement of less than one percent, denoted in the graph as "Profit < 1% for 20K." We also mark the instance where 20K measurements (across all subgraphs) yield a performance gain of less than one per thousand, corresponding to "Profit < 1‰ for 20K".

To reach the convergence state, an average of 104, 69, 145, and 130 measurements per subgraph are required, corresponding to overall totals of 225K, 213K, 314K, and 344K measurements, respectively, for all subgraphs. In other words, inducing convergence across all 126 workloads utilizing TLM calls for approximately 200K tensor program measurements on the GPU and about 300K on the CPU. TLM involves a data volume an order of magnitude smaller compared to TenSet and TLP, which utilize approximately 8.6 million measurements. Furthermore, for a performance gain of less than 1‰, individ-

Table 2: The overall speedup for the 23 TLM-Ansor subgraphs. The higher the overall speedup, the better. In the table, "Times" represents the measurement times for each subgraph.

		Ansor			TLM-Ansor
	Times	64	1K	10K	10K
TLM-Ansor	1	1.26	0.98	0.92	0.85
	10	1.40	1.08	1.03	0.95
	16	1.43	1.10	1.04	0.96
	32	1.45	1.12	1.06	0.98
	64	1.45	1.12	1.06	0.98
	1K	1.46	1.13	1.07	0.99
	10K	1.48	1.14	1.08	1.00
Ansor	10K	1.37	1.06	1.00	0.92

Table 3: The overall speedup for the 40 TLM-Meta subgraphs.

		MetaSchedule			TLM-Meta
	Times	64	1K	10K	1K
TLM-Meta	1	1.00	0.69	0.68	0.67
	10	1.41	0.96	0.95	0.94
	16	1.45	1.00	0.99	0.97
	32	1.46	1.01	1.00	0.97
	64	1.49	1.02	1.01	0.99
	1K	1.50	1.02	1.01	1.00
MetaSchedule	10K	1.48	1.01	1.00	0.99

ual subgraphs require 272, 150, 375, and 363 measurements each, which also translates to a cumulative total of 588K, 467K, 813K, and 963K measurements for all subgraphs. The TLM used in the experiments of the subsequent sections has been fine-tuned using about 300K labelled data.

6.3 Subgraph Benchmark

After SFT, we conduct subgraph experiments on the NVIDIA V100. The TLM test set comprises 12 workloads, yielding 232 and 364 subgraphs for TLM-Ansor and TLM-Meta, respectively. These subgraphs fall into 23 and 40 categories, and we select one representative subgraph from each category for the experiments. Two comparisons are established: TLM-Ansor vs. Ansor and TLM-Meta vs. MetaSchedule, with measurement times set at 10K, 10K, 1K, and 10K for each subgraph, respectively. The latency of a subgraph is defined as the lowest value among its measurements. The latency comparison curves are presented in Figures 14 and 15 in the Appendix B, while here, we focus on critical data highlights in Tables 2 and 3.

We define Framework$_k$ as follows:

$$\text{Framework}_k = \sum_{s_i \in \text{subgraphs}} \min\left(\begin{array}{c} \text{Framework's } k \\ \text{record latencies of } s_i \end{array} \right),$$

where Framework can be TLM-Ansor, TLM-Meta, Ansor, or MetaSchedule, and k represents the measurement times. Each speedup in Table 2 represents the overall speedup for the 23 subgraphs of Framework1$_{k_1}$ in the first two columns compard to Framework2$_{k_2}$ in the first two rows. Table 3 is similar to Table 2.

In Tables 2 and 3, TLM-Ansor$_{1K}$ achieves 99% (the text color matching that in the tables) of the performance of TLM-Ansor$_{10K}$, while TLM-Meta$_{64}$ attains 99% of the performance of TLM-Meta$_{1K}$. Furthermore, MetaSchedule$_{10K}$ shows a speedup of 1.01× compared to MetaSchedule$_{1K}$. We observe that a tenfold increase in the number of measurements results in a performance gain of no more than one percent. Achieving further acceleration in the **current** exploration space is challenging, and a better approach to gain additional speedup is to explore a larger space.

The primary goal of the TLM framework is to generate high-performance tensor programs efficiently. Notably, even with 10 measurements, TLM can achieve 103% and 95% of the performance of Ansor and MetaSchedule after 10K measurements, respectively. Additionally, TLM-Ansor$_{32}$ achieves a 1.06× speedup over Ansor$_{10K}$, while TLM-Meta$_{32}$ reaches a 1.00× speedup compared to MetaSchedule$_{10K}$. It is evident that TLM's performance, with 32 measurements, has already exceeded that of both Ansor and MetaSchedule.

TLM-Ansor$_{1K}$ achieves a 1.13× speedup over Ansor$_{1K}$, TLM-Ansor$_{10K}$ reaches a 1.08× speedup compared to Ansor$_{10K}$, and TLM-Meta$_{1K}$ attains a 1.02× speedup over MetaSchedule$_{1K}$. These ratios demonstrate that TLM consistently achieves acceleration relative to Ansor and MetaSchedule with equal measurement times. To attain a higher acceleration ratio, building a larger exploration space might be necessary.

6.4 End-to-End Workload Benchmark

6.4.1 Comparison with Ansor/MetaSchedule

For both GPU and CPU, we set up four experiments: 1) Tuning with Ansor (V0.12) and conducting 20K measurements; 2) Tuning with MetaSchedule (V0.12) and conducting 20K measurements; 3) Generating tensor programs using TLM-Ansor, carrying out 20K measurements, and reporting end-to-end performance for $1 \times g$, $10 \times g$, and $32 \times g$ measurements, where g indicates the number of subgraphs partitioned from the test workload[1]; 4) Generating tensor programs using TLM-Meta, with the same settings as in 3.

GPU results. Initially focusing on the last four columns of each workload in Figure 10, TLM-Ansor-20K shows a speedup of 0.99-1.38× compared to Ansor-20K across 12 test workloads, with the average speedup being 1.08. Similarly, TLM-Meta-20K achieves a speedup of 0.98-1.14× relative to MetaSchedule-20K, with the average speedup being 1.04. These results align with those from the subgraph benchmark, indicating that TLM offers acceleration over Ansor/MetaSchedule.

The primary goal of the TLM framework is to efficiently generate high-performance tensor programs. We now turn our attention to the first six columns in Figure 10. For Ansor-20K,

[1] For instance, TLM-Ansor can partition 9 subgraphs from BERT-base; thus, $1 \times g$ represents 9 measurements, $10 \times g$ represents 90, and $32 \times g$ represents 288.

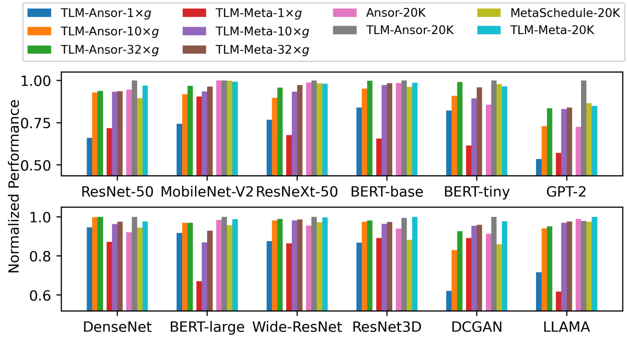


Figure 10: Workload inference performance comparison with Ansor/MetaSchedule on V100.

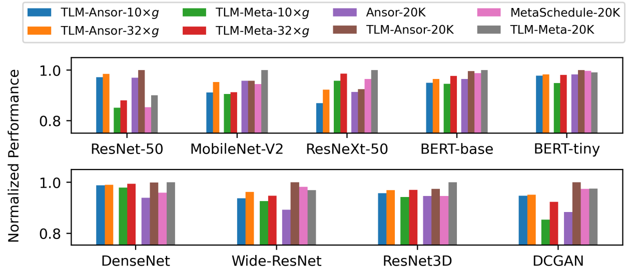


Figure 11: Workload inference performance comparison with Ansor/MetaSchedule on the Intel CPU.

TLM-Ansor-1×g achieves a 0.68-1.03 speedup, with an average speedup of 0.83; TLM-Ansor-10×g offers a 0.91-1.08 speedup, averaging at 0.99; and TLM-Ansor-32×g speeds up by 0.96-1.16, with an average speedup of 1.03. Regarding MetaSchedule-20K, TLM-Meta-1×g can accelerate by 0.63-1.04, with an average speedup of 0.80; TLM-Meta-10×g achieves a 0.91-1.11 speedup, averaging at 1.00; and TLM-Meta-32×g offers a 0.96-1.12 speedup, with an average speedup of 1.02.

To summarize, TLM can reach 80% of Ansor/MetaSchedule's performance by conducting only 1×g measurements, as opposed to the 20K measurements required by Ansor/MetaSchedule. With 10×g measurements, TLM's performance aligns with that of Ansor/MetaSchedule. Notably, the purpose of measurement is to identify the tensor program with the lowest execution latency. The fact that only one measurement is needed implies no necessity for measurement, indicating that TLM can reach 83% of Ansor/MetaSchedule's performance without any measurement. This indicates the feasibility of applying the TLM strategy within deep learning training frameworks (e.g., PyTorch, TensorFlow, MindSpore [41]).

CPU results. The CPU results (excluding BERT-large, GPT-2, and LLAMA, which will trigger the OOM error), depicted in Figure 11, align with the GPU outcomes. Here, we simply present the average speedup. Relative to Ansor-20K, TLM-Ansor-10×g achieves a 1.01× speedup, while TLM-Ansor-32×g accomplishes a 1.03× speedup. Compared to MetaSchedule-20K, TLM-Meta-10×g attains a 0.97× speedup, and TLM-Meta-32×g achieves a 1.00× speedup.

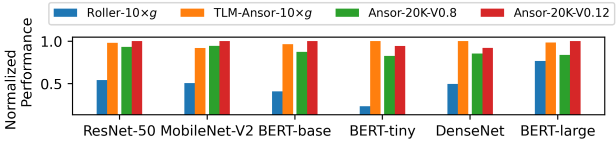


Figure 12: Workload inference performance comparison with Roller on V100.

Compilation time. This paper uses the measurement times to calculate the speedup of compilation time, the justification for which is discussed in the Appendix A. Simply put, the time allocated to measurement predominates the entire compilation time and remains unaffected by the system's load, establishing it as a stable metric for the issue.

In the TLM-Ansor test set, the 12 workloads comprise 5-72 subgraphs, averaging 20.9 subgraphs. As a result, TLM-Ansor-10×g, compared to Ansor-20K, can deliver the same performance level while accelerating compilation by 95×. Similarly, TLM-Meta-10×g can speed up compilation by 61× relative to MetaSchedule-20K.

Summary. In subgraph and end-to-end benchmarks, we primarily analyze from a statistical perspective rather than investigating why certain subgraphs or workloads surpass the baseline. This is due to the inherent randomness of the probabilistic/random sampling, which results in a lack of clear patterns in speedup. What becomes apparent, however, is that across nearly all subgraphs and workloads, TLM matches the baseline results with significantly fewer measurements. This indicates a general improvement in exploration capabilities.

6.4.2 Comparison with Roller on V100

Roller is implemented on top of TVM (V0.8) and Rammer [42]. We utilize the Docker image provided by Roller for experiments, which runs on Ubuntu 16.04 with CUDA 10.2 and cuDNN 7.6.5; it lacks maintenance to utilize the latest CUDA. To observe the impact of software versions on performance, we present the performance of Ansor, integrated within TVM (V0.8), as a bridge for comparing TLM and Roller; the performance of Ansor is measured in the same execution environment as Roller. Compiling workloads with Roller requires carefully configured, workload-specific script files; we offer script files for six workloads.

We set up four experiments: 1) Compiling with Roller and performing 10×g measurements; 2) Adopting TLM-Ansor-10×g from §6.4.1; 3) Tuning with Ansor (V0.8) and executing 20K measurements; 4) Adopting Ansor-20K from §6.4.1, designated as Ansor-20K-V0.12.

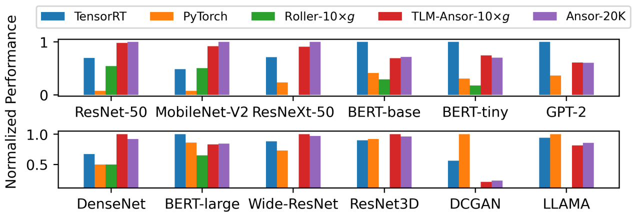


Figure 13: Workload inference performance comparison with TensorRT/PyTorch on V100.

Figure 12 illustrates that in the same execution environment, Ansor-20K-V0.8 outperforms Roller-10$\times g$. Additionally, TLM-Ansor-10$\times g$ and Ansor-20K-V0.12 have comparable performance levels, slightly better than Ansor-20K-V0.8. Therefore, it is evident that TLM-10$\times g$ surpasses Roller-10$\times g$ when the influence of software versions is excluded. In a direct comparison, TLM-Ansor-10$\times g$ achieves a speedup of 1.28-4.23× compared to Roller-10$\times g$, with the average speedup being 2.25. The reason is that, in pursuit of high efficiency, Roller significantly prunes the exploration space by aligning tensor shapes with the key features of the hardware, as discussed earlier.

6.4.3 Comparison with TensorRT/PyTorch on V100

In this section, we conduct a performance comparison against TensorRT [43] (V8.6) and PyTorch (V1.13.1) on V100. Both TensorRT and PyTorch are backed by static kernel libraries.

We set up five experiments: 1, 2) Performing inference using TensorRT and PyTorch; 3) Adopting Roller-10$\times g$ from §6.4.2; 4, 5) Adopting TLM-Ansor-10$\times g$ and Ansor-20K from §6.4.1.

Figure 13 illustrates that relative to TensorRT, TLM-Anosr-10$\times g$ achieves a 0.38-1.89× speedup, averaging at 1.04; compared to PyTorch, TLM-Anosr-10$\times g$ sees a 0.21-12.92× increase in performance, with an average speedup of 3.42. TensorRT outperforms TLM-Anosr-10$\times g$ in BERT-tiny, BERT-base, BERT-large, GPT-2, LLAMA, and DCGAN workloads, while PyTorch excels over TLM-Anosr-10$\times g$ in BERT-large, LLAMA, and DCGAN. The primary components of BERT and LLAMA are batch matmul operators, GPT-2 mainly involves matmul operators, and DCGAN primarily uses transposed 2D convolution operators. The speedup achieved by TensorRT/PyTorch is attributed to the deep optimization of batch matmul, matmul, and transposed 2D convolution operators in recent kernel libraries, as well as the utilization of hardware computing units. Overall, tensor compilers excel in supporting a wide range of operators, while static kernel libraries are more adept at deeply optimizing commonly used operators.

7 Related Work

Halide [44] introduces the concept of separating compute and schedule, employing a domain specific language (DSL) to define computations and scheduling primitives to abstract hardware characteristics, enhanced by an auto scheduler [4,45,46] for optimal primitive combination. TVM [3], inheriting the philosophy of Halide, utilizes scheduling primitives for operator implementation. It currently boasts three generations of tensor program search frameworks: The first generation maps subgraphs to tensor programs using templates and optimizes them through AutoTVM [12]. The second generation, Ansor [13], addresses the limitations of template-based exploration spaces by constructing tensor programs with derivation rules and searching for efficient programs using the genetic algorithm. The third generation, which includes TensorIR [47] and MetaSchedule [14], tackles the challenges of supporting TensorCore, introducing a block abstraction that isolates tensorized computations for mapping to tensor computing units. TenSet [20] and TLP [21] propose using offline datasets to address the issue of extended search times brought by Ansor/MetaSchedule. Roller [23] introduces a tile abstraction that encapsulates tensor shapes, aligning them with the key features of the underlying accelerator to limit shape choices. FlexTensor [6] is a schedule exploration and optimization framework proposing automatically general templates to map the tensor algorithms onto low-level implementations for different hardware platforms.

Tiramisu [48], AKG [15], and Tensor Comprehensions [5] apply polyhedral-based techniques, formulating the optimization of programs as an Integer Linear Programming (ILP) problem. Triton [49] introduces a tile-based template representation where programmers can specify block sizes and manage their scheduling for effective program optimization. CUTLASS [50] is a collection of template abstractions for implementing high-performance matrix-matrix multiplication and related computations within CUDA at all levels and scales. MLIR [51] builds reusable and extensible compiler infrastructure to address software fragmentation and improve compilation for heterogeneous hardware.

8 Discussion

Limitation. 1) At its core, TLM is a deep learning model that leverages offline data to guide exploration. The data distributions of the target and training scenarios need to be aligned. To achieve optimal performance in scenarios with significant discrepancy, it might be necessary to incorporate additional training data. 2) TLM's design philosophy is trading compile time for pre-compile time. The overhead of training TLM, which could span tens of hours, should not be overlooked. If the intent is solely to compile one or just a handful of models, TLM might not be the most economical choice. Instead, TLM is better suited for compiling a vast array of models.

Future Work. 1) With 100M parameters, TLM's time and hardware overhead for training and inference should be noticed. It may be worthwhile to explore substantial reductions in parameter size while ensuring TLM's performance remains robust. 2) In contrast, with just 100M parameters, there is an opportunity to substantially expand TLM's parameter size and the training data. These improvements could lead to remarkable generalization capabilities. Examples include compiling directly from TLM-generated results without measurement or achieving strong generalization across various hardware platforms. 3) Delve into a more expansive exploration space. Equipped with TLM's potent exploration capabilities, it's feasible to navigate vast territories without being constrained by exploration costs.

9 Conclusion

We introduce the TLM framework, a novel tensor program generation framework that consists of two decoupled components: a space builder and a generator. The TLM framework is pioneering in its integration of language models into the tensor program domain. Owing to TLM's dual strengths of adeptly learning from offline data and effectively capturing context, it demonstrates formidable generative capabilities. Experimental results show that TLM consistently delivers both high efficiency and high performance.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62332016, No. 62272434, and No. U20A20226).

References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[2] Intel® oneAPI Deep Neural Network Library. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.htm.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[4] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[5] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[6] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[7] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*, pages 58–68, 2018.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[13] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[14] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[15] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. Akg: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1233–1248, 2021.

[16] The LLVM Compiler Infrastructure. URL: https://llvm.org/.

[17] NVIDIA CUDA Compiler Driver NVCC. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[18] Open standard for machine learning interoperability. URL: https://onnx.ai.

[19] ChatGPT. URL: https://openai.com/blog/chatgpt.

[20] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[21] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 833–845, 2023.

[22] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[23] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

[24] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[25] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[27] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[29] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[30] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[31] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[34] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[35] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[36] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[38] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 3154–3160, 2017.

[39] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[41] MindSpore. URL: https://www.mindspore.cn/.

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[43] NVIDIA TensorRT. URL: https://developer.nvidia.com/tensorrt.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[45] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (ToG)*, 37(4):1–13, 2018.

[46] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[47] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[48] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[49] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[50] CUTLASS. URL: https://github.com/NVIDIA/cutlass.

[51] Multi-Level Intermediate Representation Overview. URL: https://mlir.llvm.org.

A Compilation Speedup Metrics

In this section, we discuss the rationality of using measurement times to calculate the speedup of compilation time.

$$
\begin{aligned}
T_{total} &= T_{exploration} + T_{post\ exploration\ compilation} \\
&= \sum_k \left(c \times T_{sample} + T_{measurement}\right) \\
&\quad + T_{post\ exploration\ compilation} \\
&= \sum_k \left(c \times T_{sample} + (T_{compilation} + T_{execution})\right) \\
&\quad + T_{post\ exploration\ compilation}
\end{aligned}
$$

The total time for compiling a workload (T_{total}), as indicated in the formula above, consists of two main components: the time spent exploration to identify high-performance tensor programs ($T_{exploration}$), and the time for the final compilation of the workload ($T_{post\ exploration\ compilation}$). The exploration process involves sampling c tensor programs ($c \times T_{sample}$) and then measuring the execution latency of one of these programs ($T_{measurement}$). In Ansor/MetaSchedule, the purpose of sampling multiple tensor programs is to use a cost model to select the most promising tensor program; if a program sampled by TLM is found to be invalid, it will be resampled. The sampling coefficients c_{Ansor} and $c_{MetaSchedule}$ are approximately 128, while c_{TLM} does not exceed 1.1. Measuring a tensor program includes both its compilation ($T_{compilation}$) and execution ($T_{exection}$).

$$
\begin{aligned}
T_{total} &= T_{exploration} + T_{post\ exploration\ compilation} \\
&\approx T_{exploration} \quad \text{if } k \geq k_0 \\
&= \sum_k \left(c \times T_{sample} + T_{measurement}\right) \\
&= \sum_k T_{measurement} \quad \text{if } T_{measurement} \text{ hides } c \times T_{sample} \\
&= \sum_k \left(T_{compilation} + T_{execution}\right)
\end{aligned}
$$

When measurement times k exceeds a certain threshold k_0 (e.g., 100), $T_{exploration}$ becomes the dominant factor in T_{total}, making $T_{post\ exploration\ compilation}$ negligible. Sampling and measurement processes can be optimized using a pipeline approach for parallel execution. Overall, measurement is a critical part of the entire compilation process, and reducing measurement times requires methodological innovation rather than just engineering efforts.

TLP uses the speedup on T_{total} to calculate the acceleration of compilation time, while Roller uses $\sum_k T_{compilation}$ to calculate the acceleration of compilation time. Although they indeed reflect the speedup of compilation time to some extent, these metrics are unstable due to the influence of system load. In this paper, we use measurement times k to calculate the speedup of compilation time.

B Subgraph Performance

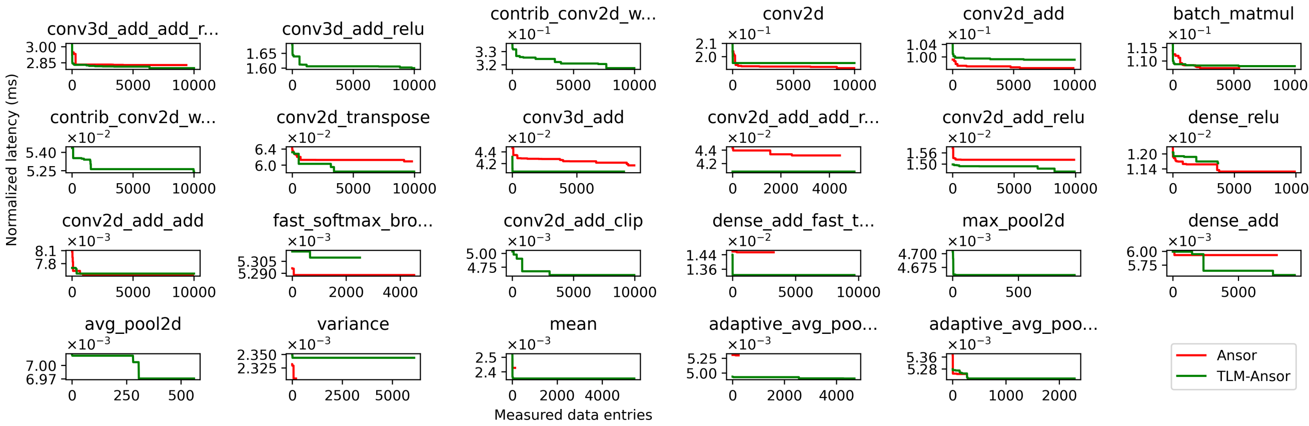


Figure 14: Subgraph latency comparison curves for TLM-Ansor vs. Ansor. To enhance the comparison of subtle details, the curve only includes latencies that do not exceed 1.1 times the lowest latency of either TLM-Ansor or Ansor. In the figure, if one is not visible throughout, it indicates that its lowest latency exceeds that of the other by 10%.

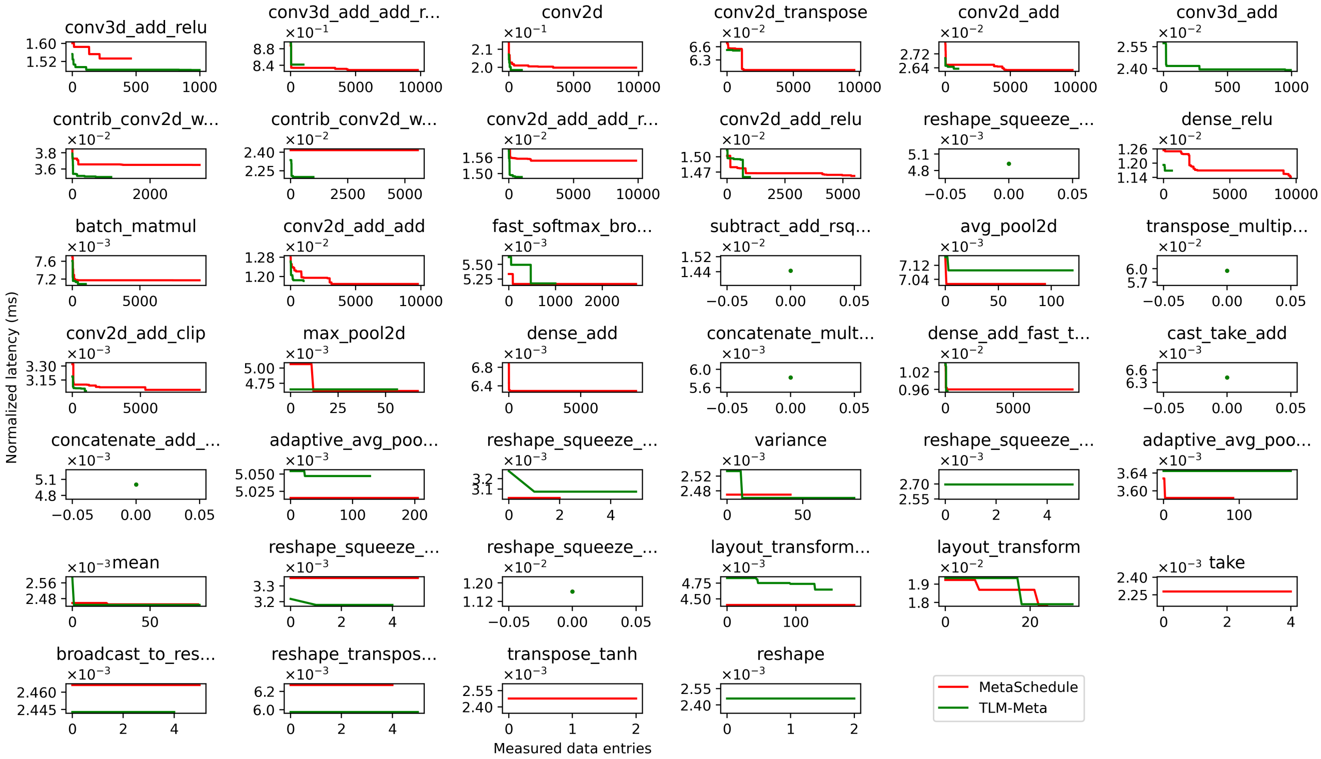


Figure 15: Subgraph latency comparison curves for TLM-Meta vs. MetaSchedule. To enhance the comparison of subtle details, the curve only includes latencies that do not exceed 1.1 times the lowest latency of either TLM-Meta or MetaSchedule. In the figure, if one is not visible throughout, it indicates that its lowest latency exceeds that of the other by 10%.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

LADDER: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation

Lei Wang[†◇*] Lingxiao Ma[◇] Shijie Cao[◇] Quanlu Zhang[◇] Jilong Xue[◇] Yining Shi[‡◇*]
Ningxin Zheng[◇] Ziming Miao[◇] Fan Yang[◇] Ting Cao[◇] Yuqing Yang[◇] Mao Yang[◇]

[†]*University of Chinese Academy of Sciences* [‡]*Peking University* [◇]*Microsoft Research*

Abstract

The increasing demand for improving deep learning model performance has led to a paradigm shift in supporting low-precision computation to harness the robustness of deep learning to errors. Despite the emergence of new low-precision data types and optimization approaches, existing hardware and software have insufficient and inefficient support for those evolving data types, making it challenging to achieve real performance gains through low-precision computing.

This paper introduces LADDER, a novel compiler designed to bridge the gap between evolving custom data types and the fixed precision formats supported by current hardware. Leveraging a general type system, *t*Type, and an extended tensor expression, LADDER transforms deep neural network (DNN) computations into optimized computing pipelines with custom data types as the first-class citizen, exposing an optimization space for efficiently handling data storage, accesses, and type conversions. LADDER employs a new set of tensor scheduling primitives and a hardware-aware optimization policy to navigate the complex transformation space, ensuring optimal performance across different memory layers and DNN operators. Our evaluation demonstrates LADDER's capability to systematically support a wide array of low-bit precision custom data types, significantly enhancing the performance of DNN computations on modern accelerators without necessitating hardware modifications. This innovation empowers model designers with the ability to explore data type optimizations and offers hardware vendors a flexible solution to expand their support for diverse precision formats.

1 Introduction

Building on the recent advancements in scaling up deep learning models [11, 17, 26], there's a growing demand for more powerful computing performance in hardware accelerators like GPUs. The inherent robustness of deep learning to errors enables the use of lower precision arithmetic, setting it apart from traditional workload like scientific computing, which necessitate high precision like float64. In line with this trend, cutting-edge accelerators are increasingly integrating more low-precision computational units, such as 32-bit, 16-bit, and even 8-bit floating-point operations, into their new generations. At the same time, model developers are vigorously investigating various custom low-precision data types, such as mixed precision formats, to strike an optimal balance between model accuracy and training efficiency. Moreover, during the model deployment phase, computations can be converted to even more compact data representations to achieve extreme efficiency, such as 2 bits fixed-point precision in LLM [12] or group-based types where multiple values share the same scaling factor [41].

However, hardware accelerators are challenging in keeping pace with the diverse and rapidly evolving requirements for supporting various data precision formats, i.e., custom data types. This difficulty arises because each accelerator can only integrate a few types of computing units for standard data types, given the limited chip area and high hardware cost. Even for those recently supported low-precision data types, such as those under 16 bits in width, existing software is generally inefficient due to the complexity of aligning fine-grained low-bit data access with the coarse-grained memory system. For instance, NVIDIA GPU's shared memory bank size is 4 bytes in width, and simply loading or storing 8-bit data elements can easily lead to bandwidth waste. This often necessitates non-trivial optimizations, such as packing multiple data values together to align with the features of different memory hierarchies. Consequently, optimizing kernel libraries for all these new data types, combined with different operators and shapes, becomes a challenging task. For instance, the highly-optimized cutlass library for NVIDIA GPUs only achieves 422 tflops (68% utilization) on INT8 matrix multiplication. The inadequacy and inefficiency in supporting these new custom data types significantly hinder the innovation for both models and accelerators.

To address these challenges, we make the following observations: First, despite hardware accelerators lacking comput-

[*] Work is done during the internship at Microsoft Research.

ing instructions for those custom data types, their memory system can be utilized to store arbitrary data types by casting them into an opaque data chunk with a fixed bit width. Second, most custom data type can be losslessly converted to a wider-bits standard data type supported by the computing units in existing hardware. For example, NF4 tensors can be computed with an FP16 or FP32 operation by converting their data types. These observations inspire us a general approach to support all custom data types by separating data storage and computation. That is, store and transmit tensors in custom data types and compute in standard data types through type conversion. Given that modern DNN models tend to be memory-intensive and the latest hardware faces the memory wall issue [43], such an approach is increasingly critical as it can effectively exploit the performance benefits of low-bits data types by saving memory traffic and footprint.

However, efficiently supporting such computing pipeline for general custom data types on existing accelerators is non-trivial. A typical tensor computation pipeline involves loading data from multiple layers of memory hierarchy, such as DRAM, L2 cache, shared memory, register, etc. First, converting tensor data types in different layers could significantly impact the performance factors like memory footprint, data access traffic, hardware cost, etc., which is complex to optimize. For example, converting a low-bit data chunk to a higher-bit type in a register could lead to register spill, causing a dramatic performance drop. Second, pipelines involving different data types usually require different data layout optimization to align with memory system, e.g., align with memory bank, to maximize the data access throughput. Existing optimizations like swizzling memory accesses [6] are mostly designed for a few specific data types, which is hard to be generalized.

To address these challenges, we present LADDER, a compiler for efficient deep learning computation on general custom data types. To facilitate the implementation of quickly-evolving custom data types, such as block-wise data types like MXFP, LADDER first introduces a general type system called *t*Type. *t*Type is inherently a tile-wise data type, which can define all common custom types by explicitly specifying type width, element shape, and the type-converting functions. Based on *t*Type, LADDER extends the existing tensor expression, used to express a DNN operator, to natively support annotating *t*Type for each tensor. This way, LADDER can systematically translate a DNN computation with custom data types into a standard computation pipeline.

To optimize the computation pipeline involving custom data storage, access, and type conversions, we observe that tensor storage and access in a pipeline can be transformed into various logically equivalent formats, each with dramatically different performance impacts. For instance, a sub-tensor can be stored in row-major, column-major, block-wise, or even custom-defined layouts, padded to a certain shape to match computing instructions, and accessed in different granularities (e.g., different tile shapes) by the upper-layer memory. All these factors significantly affect overall performance. To facilitate such transformations, LADDER introduces a set of tensor scheduling primitives, including *slice*, *map*, *pad*, and *convert*, that can be used to transform a default computing pipeline into optimized ones.

Deriving optimal tensor transformations for a specific computing pipeline requires holistic consideration of inter-memory layer and inter-operator optimizations. For example, a specific data layout can be propagated to adjacent operators to avoid explicit layout conversation costs. Moreover, the data layout in a specific memory layer needs to consider both the memory feature and upper-layer access pattern. Both cross-layer or cross-operator optimizations form a vast optimization space. LADDER optimizes such transformation space through a layer-wise hardware-aware optimization policy: a lower-layer memory provides the preferred data access granularity as a hint, and the upper layer decides the optimal compute granularity by aligning with the data access granularity. Thus, LADDER first models a DNN computation into a tile-level data flow graph and then optimizes the transformation scheduling using a granularity-aware scheduling policy.

LADDER is implemented on top of TVM [13], Roller [57] and Welder [43]. We have open-sourced LADDER [1]. Furthermore, the DNN operation compilation in LADDER has also been released as BitBLAS [2], a library that can be integrated into existing DNN and LLM frameworks to empower efficient low-precision computing in existing deep learning ecosystem. Our evaluation of DNN inference on NVIDIA A100, NVIDIA V100, NVIDIA RTX A6000 and AMD Instinct MI250 GPUs shows that LADDER outperforms state-of-the-art DNN compilers on native-supported data types, while efficiently supports custom data types that GPUs do not support with up to 14.6× speedup. As a result, LADDER is the first system to systematically support general low-bit precision represented by custom data types for DNN computation on modern hardware accelerators. It opens the door for both model designers to explore more flexible data type optimization with real performance feedback and hardware vendors to support a large range of types without hardware modification.

2 Background and Motivation

2.1 Precision Requirements in Deep Learning

The increasing demand to scale deep learning models to larger sizes, such as Large Language Models (LLM), enhances the requirement of computing in lower bits and mixed precision to increase computation efficiency and save memory. This section introduces some new data type requirements in deep learning.

[1] https://github.com/microsoft/BitBLAS/tree/osdi24_ladder_artifact

[2] https://github.com/microsoft/BitBLAS

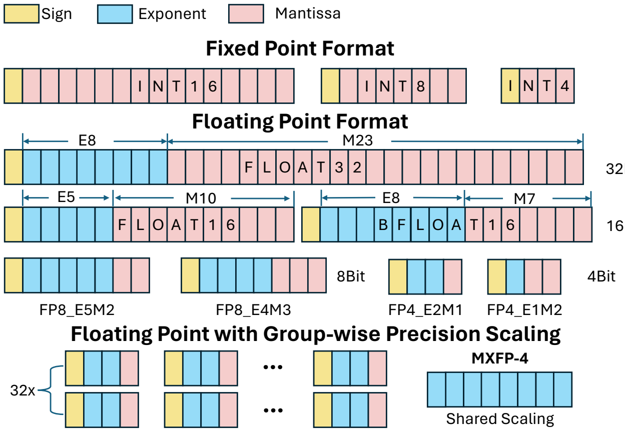


Figure 1: Diverse narrow-precision data types in deep learning training and inference.

Lower-bit numeric precision. FP32 (32-bit float) has been the go-to choice for data representation in deep learning models. However, recent practices suggest that the high precision of FP32 isn't always necessary. Lower precision can deliver the same level of effectiveness while simultaneously reducing costs. A pivotal example of this precision shift is the FP16/BF16 computation in Automatic Mixed Precision (AMP) training [36]. More aggressively, systems like *Transformer Engine* [37] and *MS-AMP* [39] have begun to employ FP8 for weight, gradient, and even optimizer tensors, pushing the boundaries of precision reduction in deep learning. During inference, models are frequently quantized to significantly lower precision, typically down to 8 or 4 bits [14,22,48]. Contemporary cutting-edge research is challenging these limits further, aiming to decrease weight quantization to a remarkable 2 or even 1 bit [12, 46]. This is primarily due to the redundancy inherent in pretrained weights and the fact that computations are mostly forward passes. Figure 1 highlights various data formats used in deep learning models, marking the notable shift from high-precision formats to low-bit alternatives.

Group-wise precision scaling. To improve the accuracy and robustness of low-precision deep learning models, a common approach is to use a scaling factor to rescale the values for a more accurate representation of the data distribution. Traditional methods typically employ a tensor-wise or channel-wise scaling factor. However, group-wise scaling, by virtue of its finer granularity, can better capture the distribution of sub-tensors or groups, leading to improved performance. For instance, in Post-training Quantization (PTQ) [22], group sizes of 128 and 64 are typically preferred, with each group scaled using FP16. In OCP-MXFP [41], an 8-bit shared scale is applied to a group of 32 elements.

Mixed-precision operations. Mixed-precision operations emerge in data quantization due to the varying sensitivity of different tensors to lower bit quantization. For example, mixed-precision training employs a combination of higher and lower bit tensors, such as FP32, FP16, and FP8. This strategic utilization of precision levels strikes a balance between computational efficiency and precision, thereby optimizing performance. Similarly, in Large Language Model (LLM) quantization, weights that are more receptive to quantization can be represented using lower bits. On the other hand, activations, which pose more substantial quantization challenges, require higher bit representations. This divergence leads to mixed-precision operations, including W4A16 (i.e., weight values are represented in 4-bit data types, and activations are represented in 16-bit data types), W2A16, W1A8, and others [12,22,46].

Data Type	$W_{FP16}A_{FP16}$			$W_{INT8}A_{INT8}$			$W_{FP8}A_{FP8}$	$W_{NF4}A_{FP16}$
GPU	V100	A100	MI250	V100	A100	MI250	V100/A100/MI250	
cuBLAS	78%	87%	X	X	68%	X	X	X
rocBLAS	X	X	46%	X	X	75%	X	X
AMOS	64%	38%	X	X	45%	X	X	X
TensorIR	67%	56%	22%	X	X	X	X	X
Roller	50%	70%	29%	X	X	X	X	X

Table 1: MatMul [M,N]=[M,K]x[N,K] where M,N,K=16384. "X" indicates not supported in tensor core or matrix core.

2.2 Insufficient Precision Supports in GPUs

Hardware accelerators like GPUs are constantly adapting to the evolving data type requirements in deep learning. Early generations of GPUs, such as NVIDIA's Fermi, supported standard data types like FP32 and FP64. As deep learning workloads gained relevance, lower precision formats like FP16 were introduced in the Pascal architecture. The Turing architecture further expanded support by introducing INT4 and INT8 for inference workloads. The Ampere architecture later introduced BF16, striking a balance between performance benefits and numerical range for machine learning applications. The latest architecture, NVIDIA's Hopper, extends this trend by supporting FP8, showcasing the ongoing pursuit of efficiency by adjusting the precision-performance trade-off. This evolution highlights the increasing versatility of GPUs in handling diverse computing workloads.

However, hardware typically lags behind the requirements of algorithms or models. When encountering unsupported data types, we must convert or simulate them in higher-precision supported data types. This could lead to significant performance issues and inefficiencies.

2.3 Inefficiency of Low-precision Computing

Low-precision computing is particularly challenging to optimize due to the fine-grained data access granularity and special hardware units, such as TensorCore. We tested the performance of a standard matrix multiplication benchmark with different precisions, using the latest software libraries and compilers on three of the latest GPUs: NVIDIA V100 and A100, and AMD MI250, as shown in Table 1. We make the following observations. First, the hardware utilization of low-precision computing is generally low, i.e., less than 60% on

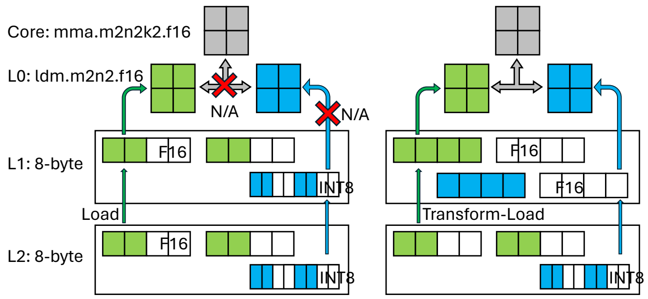


Figure 2: MatMul: $C_{FP16}[2,2]=A_{FP16}[2,4]\times B_{INT8}[2,4]$

average. Even for the most dominant precision in today's deep learning workload, like FP16, the average utilization is around 60%. Second, some hardware-supported precisions are not well supported by the software. For example, while INT8 is supported in both A100 and MI250, most existing deep learning compilers do not support INT8 computing on these GPUs. Third, meeting new precision requirements is challenging for hardware to support in a timely manner. For instance, FP8 is only available in the next generation of NVIDIA Hopper architectures. Mixed precision computing, such as F16 × NF4, is not supported by all the latest GPUs.

2.4 Our Insights

We use mixed-precision matrix multiplication, specifically FP16×INT8, as an example to illustrate our key insights, as shown in Figure 2. A DNN operation is often implemented as a computing pipeline, which continuously loads small data tiles from input tensors across multiple layers of memory hierarchy to compute in the top-level cores. Each memory layer usually has its preferred minimum access granularity, such as an 8-byte transaction length in the L1 layer. Some of the latest GPUs even introduce built-in instructions for highly efficient data loading, which load a two-dimensional data tile at a time—for instance, the *ldmatrix.2x2.f16* loads a 2x2 tile. Given that a data tile is typically stored in a strided memory space, data access often becomes unaligned with the transaction length or instruction shape, potentially leading to low bandwidth utilization. For example, the left figure illustrates that each memory access from L1 only achieves half utilization for both tensors. Furthermore, due to the absence of computing instructions for FP16×INT8, the operation cannot be supported, even if we manage to load the corresponding data into the register.

To address these issues, we observe that the alignment issue can be circumvented by transforming the tensor layout into a well-optimized one based on the data type width, memory transaction length, and instruction shape. For instance, in the right figure, we store each 2x2 tile in contiguous memory space in the L1 layer so that the load instructions at the upper layer can fully utilize the bandwidth. Moreover, given that the computing instruction only supports the FP16 data format, we can convert the second tensor from INT8 to FP16 during the data loading from the L2 to the L1 memory layer. Consequently, the data loading from L2 to L1 efficiently leverages the low traffic due to the low-bit data type, the data loading from L1 to L0 fully utilizes the memory bandwidth through transaction alignment, and the computation is ultimately accelerated in the hardware computing unit by type conversion. This example demonstrates that a DNN computation on a custom data type not supported by hardware can still be scheduled and optimized through a well-designed tensor transformation on its layout and data types.

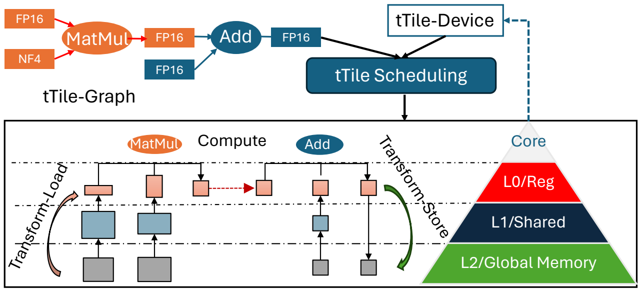


Figure 3: The system overview of LADDER

3 LADDER Design

The observations in §2 motivate LADDER, a DNN compiler that treats data type as a first-class citizen and introduces tensor transformations to support efficient DNN computation on custom data types. Figure 3 shows the system architecture.

The core of LADDER is the *TypedTile* (*t*Tile) abstraction, which augments the tile-based tensor abstraction with data type (i.e., *t*Type, §3.1). Specifically, the algorithm designer can use commonly-used data type (e.g., FP16) or define a custom data type (e.g., MXFP8, NF4) as a *t*Type, and define the DNN computation at this data type. Then, LADDER takes the DNN model as input and converts it into a *t*Tile-based data-flow graph (i.e., *tTile-graph*) where operators are defined as *t*Tile-based computing tasks (i.e., *tTile-operator*) (§3.1).

Besides, LADDER abstracts a hardware accelerator as a multi-layer hierarchy where the requirement of each layer is represented as a *t*Tile (*t*Tile-device, §3.1). *t*Tile-device explicitly describes the requirements of each layer, e.g., supported data type, transaction size, etc. By aligning *t*Tiles in the *t*Tile-graph with the *t*Tile-device, the *t*Tile-graph represented DNN computation can be executed on the hardware accelerator.

Given the initial *t*Tile-graph and the hardware specifications, LADDER will compile the DNN model into an efficient execution plan on the accelerator. To schedule the *t*Tile-graph on the *t*Tile-device and satisfy the requirements of the hardware hierarchy, LADDER separates the scheduling mechanism from its policy. On the mechanism side, LADDER proposes four *t*Tile transformation primitives: *slice*, *map*, *pad*, and *convert*, enabling the transformation from a *t*Tile to an equivalent *t*Tile (§3.2).

Then, the scheduler will schedule the initial *t*Tile-graph

```
class tType {
 TileShape shape;
 size_t nElemBits;
 struct metadata;
 map<TileType, c_func> c_tTypes;
};
```

(a)

```
class tTile {
 TileShape shape;
 tType type;

 struct metadata;
};
```

(b)

```
class tTile-Operator {
 TensorExpr expr;
 TileShape shape;
 vector<tTile> get_input_tTiles();
 vector<tTile> get_output_tTiles();
 void compute();
};
```

(c)

Figure 4: The definition of *t*Type, *t*Tile and *t*Tile-operator

into a *t*Tile-graph of fine-grained control over *t*Tile configurations, transformations and *t*Tile placement on the hardware hierarchy. The *t*Tile abstraction enlarges the scheduling space for DNN computation and opens a new trade-off between memory footprint efficiency and latency efficiency. On the policy side, LADDER plays heuristics based on observations and provides a hardware-aware layer-wise policy optimizing for latency efficiency (§3.3).

Finally, the compiled plan represented as a *t*Tile-graph is then generated as an executable code for the given hardware accelerator.

3.1 The *t*Tile Abstraction

***t*Type** According to the observations in §2, data types in DNN computation are usually defined at either element-wise granularity or block-wise granularity. To express these data types, LADDER introduces the concept of *t*Type (Figure 4(a)). Specifically, the *t*Type represents a data type that consists of a group of homogeneous elements. The layout of these elements is a n-dimensional array `shape`. Each element shares the same type with `nElemBits` bits to store an element. This group of elements also share the same `metadata`. As described in §2, data types usually can be losslessly represented by some higher-bit data types. The `c_tTypes` represents a *t*Type can be losslessly converted to another *t*Type with the `c_func` function.

Both existing commonly-used data types and new customized data types can be represented with *t*Type. For example, the FP16 type can be expressed as a *t*Type of `shape=[1]` with `nElemBits=16`. The element-wise granulated NF4 type can be expressed as a *t*Type of `shape=[1]` with `nElemBits=4` and the shared value map in `metadata`. The NF4 type can be losslessly represented as FP16, and therefore there could be a `<FP16, NF4_to_FP16_func>` entry in `c_tTypes`. The block-wise granulated OCP-MXFP8 type can be expressed as a *t*Type of `shape=[32]` with `nElemBits=8` and the shared scaling factor in `metadata`.

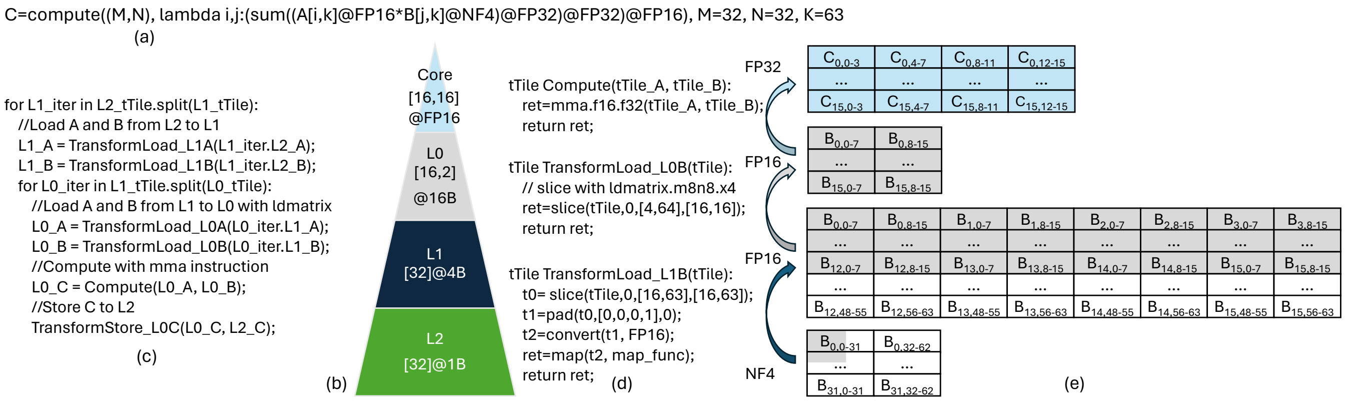
***t*Tile** Based on the *t*Type that represents a data type, LADDER proposes *t*Tile to represent a tensor of a specific data type at the fine-grained tiles. Specifically, as shown in Figure 4(b), a *t*Tile is defined as a group of homogeneous elements with the same data type `dtype` and a layout of a n-dimensional array `shape`. Elements in a *t*Tile share a `metadata`. Besides, elements in a *t*Tile are stored as row-major.

***t*Tile-Operator** A DNN operator (e.g., MatMul) are commonly implemented as a group of independent and homogeneous tasks, where each task processes a tile of the input tensor and outputs a tile of the output tensor. With the *t*Tile abstraction, a tensor of a specific data type is represented at the fine-grained tile granularity. Therefore, LADDER can leverage *t*Tile to representent a DNN operator of custom data types as a group of independent and homogeneous fine-grained tasks, i.e., *t*Tile-operator. Specifically, as Figure 4(c) shown, a *t*Tile-operator explicitly represents the tensor computation task over elements of `shape`. `get_input_tTiles()` and `get_output_tTiles()` return the input and output *t*Tiles of this computation task. `compute()` executes the computation defined in the tensor expression `expr` for the input and output *t*Tiles.

The computation of a *t*Tile-operator is defined as an index-based lambda expression `expr` [13,40,57]. However, the tensor expression in existing tensor compilers [13,21,52,56,57] focuses on describing the index and the shape and cannot flexibly indicate the data type during computation. For example, it cannot express a tensor in FP16 multiples a tensor in FP16 with FP32 as the accumulation. To support expressing computation over mix-ed data types, it requires the expression of data types during computation. Therefore, LADDER introduces the *t*Type annotation in tensor expression to explicitly indicate the data type during computation, including inputs, outputs and intermediate data, to represent computation over mix-ed data types. For example, a tensor A[M,K] of FP16 type multiplies a tensor B[N,K] of NF4 type with FP32 as the accumulation and outputs a tensor C[M,N] of FP16 type can be expressed as Figure 5(a). With the *t*Type-annotated tensor expression, given the `shape`, LADDER can infer the corresponding input and output *t*Tiles.

With the *t*Tile-based fine-grained representation for DNN operators, a DNN model can be represented as a fine-grained ***t*Tile-graph**, where each node is a *t*Tile-operator and each edge represents the dependency of two *t*Tile-operators.

***t*Tile-based Hardware Abstraction** Modern hardware accelerators usually have a hardware hierarchy, including memory layers (e.g., DRAM, register) and computing units. Each layer in the hardware hierarchy has its preference for data accessing. Specifically, a memory layer usually requires accesses via transactions where a transaction is a sequential or a shape of data at a granularity. For example, the shared memory of NVIDIA GPUs requires a transaction of 32 4-byte banks. A compute unit also usually requires processing a shape of data at a granularity. For example, the `hfma2` instruction in NVIDIA GPUs processes at the granularity of two FP16 value.



```
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16), M=32, N=32, K=63
```

```
for L1_iter in L2_tTile.split(L1_tTile):
  //Load A and B from L2 to L1
  L1_A = TransformLoad_L1A(L1_iter.L2_A);
  L1_B = TransformLoad_L1B(L1_iter.L2_B);
  for L0_iter in L1_tTile.split(L0_tTile):
    //Load A and B from L1 to L0 with ldmatrix
    L0_A = TransformLoad_L0A(L0_iter.L1_A);
    L0_B = TransformLoad_L0B(L0_iter.L1_B);
    //Compute with mma instruction
    L0_C = Compute(L0_A, L0_B);
    //Store C to L2
    TransformStore_L0C(L0_C, L2_C);
```

```
tTile Compute(tTile_A, tTile_B):
  ret=mma.f16.f32(tTile_A, tTile_B);
  return ret;

tTile TransformLoad_L0B(tTile):
  // slice with ldmatrix.m8n8.x4
  ret=slice(tTile,0,[4,64],[16,16]);
  return ret;

tTile TransformLoad_L1B(tTile):
  t0= slice(tTile,0,[16,63],[16,63]);
  t1=pad(t0,[0,0,0,1],0);
  t2=convert(t1, FP16);
  ret=map(t2, map_func);
  return ret;
```

$C_{0,0-3}$	$C_{0,4-7}$	$C_{0,8-11}$	$C_{0,12-15}$
...	...	...	...
$C_{15,0-3}$	$C_{15,4-7}$	$C_{15,8-11}$	$C_{15,12-15}$

$B_{0,0-7}$	$B_{0,8-15}$
...	...
$B_{15,0-7}$	$B_{15,8-15}$

$B_{0,0-7}$	$B_{0,8-15}$	$B_{1,0-7}$	$B_{1,8-15}$	$B_{2,0-7}$	$B_{2,8-15}$	$B_{3,0-7}$	$B_{3,8-15}$
...	...	...	...	...	...	...	...
$B_{12,0-7}$	$B_{12,8-15}$	$B_{13,0-7}$	$B_{13,8-15}$	$B_{14,0-7}$	$B_{14,8-15}$	$B_{15,0-7}$	$B_{15,8-15}$
...	...	...	...	...	...	...	...
$B_{12,48-55}$	$B_{12,56-63}$	$B_{13,48-55}$	$B_{13,56-63}$	$B_{14,48-55}$	$B_{14,56-63}$	$B_{15,48-55}$	$B_{15,56-63}$

$B_{0,0-31}$	$B_{0,32-62}$
...	...
$B_{31,0-31}$	$B_{31,32-62}$

Figure 5: MatMul of FP16 tensor A and NF4 tensor B: (a) *t*Type-annotated tensor expression, (b) *t*Tile-device for NVIDIA A100, (c) Pseudo code of computing pipeline, (d) `Tramsform-Load` with *t*Tile transformation primitives, (e) Tensor B transformations

These requirements can be described as *t*Tiles.

Therefore, LADDER abstracts a hardware accelerator as a hierarchy of multiple layers described as *t*Tiles, i.e., ***t*Tile-device**. Each layer is a memory layer or the compute unit, whose requirement represented as a shape on a granularity is described as a *t*Tile and the granularity is described as *t*Type.

Figure 5(b) shows the *t*Tile-device for the NVIDIA A100 GPU with the FP16 tensor cores. The FP16 tensor core MMA instruction [3] requires processing at a granularity of [16,16] and [8, 16] for two inputs, respectively. This can be expressed as a *t*Tile of `shape[16,16]` with the `dtype=FP16`. The FP16 tensor core data loading instruction [4] requires loading [16,2] data at a granularity of `half8` (i.e., 8 FP16 value), and can be expressed as a *t*Tile of `shape[16,2]` with the `dtype=16B`. Besides, the requirement of fully utilizing the shared memory can be expressed as a *t*Tile of `shape[32]` with the `dtype=4B`. The 32-byte transaction requirement of the global memory can be expresses as a *t*Tile of `shape[32]` with the `dtype=1B`.

3.2 *t*Tile Transformation

*t*Tile explicitly describes the fine-grained tensor storage and the requirements of the hardware hierarchy. The *t*Tile-represented DNN computation in the *t*Tile-graph should align with the *t*Tile-device for efficient execution. Fortunately, according to our observations in §2, the tensor storage and access in a pipeline can be transformed into logically equivalent formats, where each has different performance impacts in the hardware hierarchy. Therefore, LADDER proposes *t*Tile transformation mechanism to enable transforming the layout or the *t*Type of a *t*Tile to an equivalent *t*Tile. Specifically, LADDER augments the computation pipeline of a *t*Tile-operator as three stages on the hardware hierarchy: `Transform-Load`, `Compute`, and `Transform-Store`. `Transform-Load` loads the *t*Tiles from the lower memory layer to a higher memory layer with *t*Tile transformations. `Compute` executes the computation task of the *t*Tile-operator on the compute units. `Transform-Store` stores the *t*Tiles from the higher memory layer to a lower memory layer with *t*Tile transformations.

```
tTile slice(tTile_input, index, shape, out_shape);
tTile map(tTile_input, map_func);
tTile pad(tTile_input, pad_shape, pad_value);
tTile convert(tTile_input, new_tType);
```

Figure 6: *t*Tile transformation primitives

LADDER provides four primitives to transform a *t*Tile to an equivalent *t*Tile, as shown in Figure 6.

Slice The `slice` primitive slices a group elements of `shape` from the address `index` of the `tTile_input` and returns them as a new *t*Tile of `out_shape`. The `slice` primitive is usually used to represent the data tiling.

Map The `map` primitive modifies the layout of the elements in a *t*Tile. Given the `map_func`, the `map` primitive maps the address of each element to the expected address. For example, in Figure 5(d), the `TransformLoad_L1B` from the L2 memory layer to the L1 memory layer leverages the `map` primitive to modify the elements' addresses with the `map_func`.

Pad The `pad` primitive pads the `tTile_input` with the `pad_value` on each border given in `pad_shape`. The length of the `pad_shape` is 2 times of that of the `tTile_input`'s `shape`, and describes the left and the right borders of each dimension, respectively.

Convert The `convert` primitive converts the *t*Type of the `tTile_input` to the given `new_tType`. The given `new_tType` should be in the `c_tTypes` of the `tTile_input`'s *t*Type. `convert` will call the corresponding `c_func` of the given `new_tType` on each element in the `tTile_input`, and return the expected *t*Tile of `new_tType`. For example, in Figure 5(d), the `TransformLoad_L1B` converts the *t*Type from NF4 to FP16 with the `convert` primitive to satisfy the cores' FP16 *t*Type requirement.

With the above four primitives, a *t*Tile can be transformed to another equivalent *t*Tile by changing the `shape` with `slice` and `pad`, modifying the elements' layout with `map`, or con-

[3] mma.sync.aligned.m16n8k16.row.col.f16.f16.f32.f32

[4] ldmatrix.sync.aligned.m8n8.x4.shared.b16

verting the *t*Type with `convert`. This enables transforming the *t*Tiles of a *t*Tile-operator to align with the *t*Tile-device, so that these *t*Tiles can be efficiently processed in the hardware hierarchy.

Figure 5 shows an example that a FP16 tensor A[32,63] multiplies a NF4 tensor B[32,63] with FP32 as the accumulation and outputs a FP16 tensor C[32,32] (Figure 5(a)) on a four-layered *t*Tile-device (i.e., from L2 to core, Figure 5(b)). Specifically, Figure 5(c) shows the pseudo code of the execution. *t*Tiles of A and B are transformed and loaded from L2 to L1 as FP16 type. Then, the *t*Tiles are loaded to L0 with `ldmatrix` and processed by the `mma` instruction which accumulates intermediates as FP32 in L0. Finally, the *t*Tiles of C in L0 are transformed and stored to L2 as FP16. Figure 5(d)(e) shows the detailed transformations of the NF4 tensor B to align with the *t*Tile-device, while the transformations of tensor A are similar. Specifically, the `mma` and `ldmatrix` instructions require the FP16 data type in L1. Each layer also has its transaction requirement shown as Figure 5(b). Therefore, `TransformLoad_L1B` `slices` [16,63] and `pads` it to [16,64], which aligns with the L2's transaction requirement. Then, `TransformLoad_L1B` `converts` it to FP16 and `maps` it to another elements layout to align with the transaction requirements of L1 and L0. We get the FP16 `L1_B`[16,64] in L1. Then, `TransformLoad_L0B` leverages the `ldmatrix` to `slice` the `L1_B` and gets the FP16 `L0_B`[16,16] on L0, which aligns with the requirements of L1, L0 and the `mma` core.

3.3 Hardware-Aware *t*Tile-Graph Scheduling

Given the DNN computation represented as a *t*Tile-graph, to schedule it to a *t*Tile-device, we can map each *t*Tile-operator's computation pipeline for *t*Tiles (i.e., `Transform-Load`, `Compute`, and `Transform-Store`) to the *t*Tile-device. Specifically, we can partition each *t*Tile-operator into multiple *t*Tiles to fit the capacity of each memory layer, schedule *t*Tile transformations to align the *t*Tiles with the requirements of hardware layers, and coordinate inter-operator *t*Tile configurations and transformations for holistic optimizations. Finally, the entire *t*Tile-graph is scheduled as a data pipeline where *t*Tiles of a *t*Tile-operator node move up and down on the hardware hierarchy and are passed cross the edge to the successor *t*Tile-operator node.

The scheduling space of the *t*Tile-graph becomes much larger because *t*Tile opens another dimension (i.e., tensor transformation) in DNN computation scheduling. Furthermore, the *t*Tile transformations introduce a new trade-off between memory footprint efficiency and latency efficiency, which brings more complexities and challenges in scheduling. Take the MatMul of a FP16 tensor and a NF4 tensor on NVIDIA GPU as an example, it requires to `convert` the NF4 type to FP16 due to the hardware support limitation. This conversion should be finished before the `Transform-Load` from L1 to L0, and therefore can be scheduled to either L2 or L1. When the conversion is on L2, it will take more memory on L2 and L1, but it will not occupy the compute unit in later *t*Tile movement from L2 to L1 and to L0. When the conversion is on L1, it will save memory on L2 and save the memory bandwidth of L2, but it will occupy the compute unit for the type conversion. When the operator is bounded by the compute unit, the previous option can achieve lower latency but more memory footprint. When the operator is bounded by the memory IO, the latter one can achieve better performance on both latency and memory footprint. Additionally, as the `convert` is only required to be finished before the `Transform-Load` from L1 to L0, this `convert` can be fused into the previous operator for execution to achieve better end-to-end performance.

Given such a large scheduling space, LADDER provides a latency-oriented policy that targets at minimizing the end-to-end latency. Specifically, LADDER proposes a layer-wise scheduling policy based on hardware-awareness: a lower-layer memory provides the preferred data access granularity as a hint represented as a *t*Tile, and the upper layer decides

Algorithm 1: Hint-based layer-wise scheduling

```
Data: g: tTile-graph; D: tTile-device
Result: $g_{ret}$: scheduled tTile-graph
Function GetDeviceHint(g, D):
    D = SelectDeviceConfig(g, D);
    HintShape = None, HintGranularity = None;
    for layer ∈ D.layers do
        HintGranularity = LCM(HintGranularity, layer.tTile.type);
    for layer ∈ D.layers do
        layer.tTile = convert(layer.tTile, HintGranularity);
        HintShape = LCM(HintShape, layer.tTile.shape);
    for layer ∈ D.layers do
        layer.tTile.shape = HintShape;
    return D;
Function ScheduleTransform(op,D,$l_{id}$):
    $tTile_h$ = op.tTile[$l_{id}$-1];
    $tTile_l$ = op.tTile[$l_{id}$];
    ScheduleSlice($tTile_l$, $tTile_h$);
    if LCM($tTile_l$.shape, $tTile_h$.shape) $\neq$ $tTile_l$.shape then
        SchedulePad($tTile_l$, $tTile_h$, D);
    if $tTile_l$.type $\neq$ $tTile_h$.type then
        ScheduleConvert($tTile_l$, $tTile_h$, D);
    if nBits($tTile_h$.shape[-1]) $\neq$ nBits(D.layers[$l_{id}$].shape[-1]) then
        ScheduleMap($tTile_l$, $tTile_h$, D);
    return op.transform[$l_{id}$-1];
Function ScheduleConnectedGraph(g, D):
    D = GetDeviceHint(g, D);
    for $l_{id}$ in length(D.layers) do
        for op ∈ g[$l_{id}$] do
            op.tTile[$l_{id}$] = ScheduleTiling(op,D,$l_{id}$);
            if $l_{id}$ > 0 then
                op.transform[$l_{id}$] = ScheduleTransform(op,D,$l_{id}$);
    g = ProfileAndSelect(g);
    return g;
Function Schedule(g,D):
    g = ExtractConnectedGraph(g, D);
    for $g_{conn}$ ∈ g do
        $g_{conn}$ = ScheduleConnectedGraph($g_{conn}$, D);
    return g;
```

the optimal compute granularity by aligning with this *t*Tile represented granularity with transformations. To reduce the large scheduling space and schedule a proper plan within reasonable time, LADDER plays heuristics based on our observations.

Scheduling policy. Algorithm 1 describes the hint-based layer-wise scheduling policy. It takes a DNN model `g` represented as a *t*Tile-graph and the hardware specifications represented as a *t*Tile-device `D`, and returns the scheduled *t*Tile-graph $\mathtt{g}_{ret}$. Initially, this policy schedules the graph into sub-graphs (line 33). Each sub-graph represents as a computation pipeline that loads *t*Tiles from the lowest memory layer to the core and then stores the results to the lowest memory layer. A sub-graph could be a *t*Tile-operator or a group of *t*Tile-operators that can be fused. The `ExtractConnectedGraph` can leverage existing DNN compiler work [13, 43].

Given a sub-graph, it first infers the hints from the hardware. Specifically, it first selects the proper hardware configurations (e.g., the compute cores) (line 2), which prefers the **bit-nearest *t*Type** supported by the hardware. Because numeric types of more bits usually require more transistors to implement the hardware instructions [] and usually result in lower performance. For example, in NVIDIA A100 GPU, the NF4 type can be converted to FP16 or FP32 for processing, and LADDER will select the FP16 core (312 TFlops) rather than FP32 (19.5 TFlops). Then it finds the aligned granularity and shape for each hardware layer by **bit-alignment**, and configures the hints (line 1-11). Take the NVIDIA A100 as an example (Figure 5(b)), the `HintGranularity` is 16B required by `ldmatrix` and the `HintShape` is [4,8], where the inner dimension is 128B and aligns with the 32B transaction of global memory and the 128B transaction of shared memory. Then, the policy schedules this sub-graph from the top layer (i.e., core) to the bottom layer (i.e., DRAM) layer by layer (line 25-29). In each layer, the policy first schedules the *t*Tile-operator tiling via `ScheduleTiling` with hint (line 27), and then schedules the *t*Tile transformation (line 29). If the `ScheduleTiling` (line 27) schedules the operator tiling as multiple of [4,8] with 16B, the later `ScheduleTransform` can align this scheduling with the *t*Tile-device. Additionally, the `ScheduleTiling` can leverage existing tensor compilers [13,52,57]. In `ScheduleTransform`, the policy will check the alignment of both `shape` and `type` with the *t*Tile-device, and schedule corresponding transformations to align *t*Tiles (line 12-22). There may be some candidates after the scheduling, which will be profiled and returned the best (line 30).

ScheduleMap. The `map_func` in scheduling the `map` transformation is non-trivial. LADDER proposes a method to infer the `map_func`, i.e., mapping the elements in the *t*Tile to the required transaction size in row-major order. Figure 5(e) shows an example: at the granularity of 16B, to map the shape[16,2] in L0 to the required shape[8] in L1, elements are flatten in row-major order, resulting in shape[4,8]. `map` can also support other `map_funcs`.

This scheduling policy is not guaranteed as optimal. However, as shown in §5, this scheduling policy can already outperform state-of-the-arts and enable efficient low-precision DNN computing on GPUs. We also hope that this optimization space from the proposed scheduling mechanism could be further explored by future research on more advanced scheduling policies.

4 Implementation

LADDER is implemented by about 5K lines of code, including Python and C++, based on open-source DNN compilers: TVM [13], Welder [43], and Roller [57]. LADDER modifies TVM for implementing kernel schedules and generating kernel code, while Roller is leveraged to infer efficient *t*Tile configurations. Welder is the state-of-the-art DNN compiler that can holistically optimize DNN models, and is leveraged for end-to-end graph optimizations.

The input of LADDER is a PyTorch program. For PyTorch built-in data types, LADDER does not require any modifications on the DNN model program. Additionally, for new data types that PyTorch does not support, LADDER extends the PyTorch with custom operators for expressing tensor expressions on the user-defined data types. Given the PyTorch program, LADDER exports it to an ONNX graph. LADDER also extends ONNX to represent computation on new data types, where the *t*Type-annotated tensor expression is saved in the attribute of an ONNX graph node. With the exported ONNX graph and the *t*Tile-based specification file of the targeted hardware accelerator, LADDER automatically converts the ONNX graph into the *t*Tile-graph and performs the scheduling. Then, LADDER generates the device code for the targeted hardware accelerator.

We implemented LADDER for NVIDIA GPUs and AMD GPUs, recognizing their widespread use as the most popular accelerators for DNNs. In the rest of this section, we describe the LADDER implementation on NVIDIA GPUs in detail and briefly describe the implementation on AMD GPUs. Additionally, LADDER can be ported to new hardware instructions (e.g., FP8 tensor cores in the latest Hopper GPUs) and other hardware accelerators (e.g., Graphcore IPU) if they align with the *t*Tile-based hardware abstraction and provide programming interfaces of loading and storing data on the hardware hierarchy.

4.1 LADDER on NVIDIA CUDA GPUs

4.1.1 *t*Type and *t*Tile

LADDER has implemented the *t*Types for common data types, e.g., FP32, FP16, INT8, FP8, MXFP, INT4, NF4, INT1.

A GPU is a single instruction multiple threads (SIMT) architecture, and it prefers a group of threads process the same instruction on different data. Therefore, LADDER separately

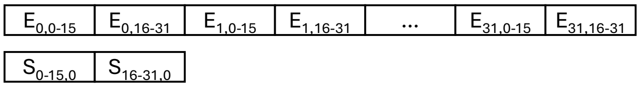


Figure 7: The storage of a MXFP8 *t*Tile of `shape[32, 32]` on NVIDIA GPU. E: elements. S: shared scaling in metadata.

stores the elements and each of the `metadata` in the *t*Tile. Figure 7 shows the storage of a MXFP8 *t*Tile of `shape[32,32]` on NVIDIA GPU. The elements are stored in an array, while the shared scaling factors are stored in another array. To access a *t*Tile, consecutive threads process consecutive elements, resulting in coalesced accesses.

Note that, there may be some data types that the `nElemBits` is not 2^n, e.g., 3-bit [22]. To support these data types, Ladder stores at the granularity of 4B due to the GPU specifications, e.g., 10 3-bit value can be stored in a 4B (32-bit) granularity.

4.1.2 Optimizing Code Generation with PTX Instruction

NVIDIA does not provides the assembly instructions for programming. Instead, NVIDIA introduces the Parallel-Thread-Execution (PTX) as a low-level virtual machine for NVIDIA GPUs, where the ISA (Instruction Set Architecture) on the PTX virtual machine can be considered as the instruction-level APIs for NVIDIA GPUs [8]. CUDA C++ code is first compiled to the PTX code and then compiled to the machine code for execution. CUDA provides both the C++ APIs and the PTX APIs for some units. For example, the tensor cores provides both the WMMA C++ APIs and the MMA PTX APIs, where a WMMA API is compiled as a group of MMA instructions by the nvcc compiler. The MMA PTX APIs have more flexibility and better performance than the WMMA C++ APIs. Ladder uses the MMA PTX APIs for codegen on tensor cores, and uses `cp.async` instructions for the new asynchronous memory copy feature on Ampere GPUs [3]. Additionally, we observed converting low-bit integers (e.g., INT4) to floats (e.g., FP16) may introduce significant overheads. Ladder implements the conversion of integers of lower than 4 bits with the `LOP3` instruction [8]. We modified the code generation module in TVM to implement these optimizations.

4.2 Ladder on AMD ROCm GPUs

AMD GPUs are similar to NVIDIA GPUs, which also have a hardware hierarchy of global memory shared by all CUs, local data store in each CU (similar to the shared memory), registers, and cores. Therefore, similar to NVIDIA GPUs, an AMD GPU can be abstracted as a four-layer *t*Tile-device with different *t*Tile configurations. ROCm provides the HIP programming model [1] for AMD GPUs, which is similar to CUDA's functionality and supports most CUDA statements. We implemented a new code generation backend for HIP in TVM to support AMD ROCm GPUs. Additionally, we use the `MFMA` (Matrix Fused-Multiply Add) ISA-level APIs to utilize the matrix core (the equivalent of the NVIDIA tensor core).

5 Evaluation

5.1 Evaluation Setup

Hardware platforms. We evaluate Ladder on a diverse range of GPUs from both NVIDIA and AMD to ensure a comprehensive assessment of performance across different hardware ecosystems. Our evaluation comprises three high-performance NVIDIA GPUs: Tesla V100 (16GB), A100 (80GB), and RTX A6000 (48GB), utilizing the CUDA toolkit version 12.1 for optimal performance. We extend to the AMD ecosystem with the inclusion of the AMD Instinct MI250 GPU (128GB), utilizing the ROCm toolkit version 5.7.0. The operating systems remain consistent, utilizing Ubuntu 20.04.

DNN models. We evaluate the effectiveness of Ladder by benchmarking the inference on a suite of state-of-the-art DNN models that span various domains and architectures. These models encompass large language models, such as LLAMA-70B [45] and BLOOM-176B [47], computer vision models, including ResNet-50 [24], ShuffleNet-V2 [34], and ViT-Base [19], as well as audio models like transducer Conformer-l [23]. The data type configurations used in these models are all from state-of-the-art research literature and have been evaluated by the deep learning community. Ladder follows these configurations and does not introduce additional model quality loss. The data type configurations, representing both weights and activations and denoted as $W_{\text{type}}A_{\text{type}}$, for the evaluated models are detailed below:

- **LLAMA-70B and BLOOM-176B:** Evaluated with data type configurations of $W_{\text{FP16}}A_{\text{FP16}}$ [45, 47], $W_{\text{INT4}}A_{\text{FP16}}$ [22, 31], $W_{\text{NF4}}A_{\text{FP16}}$ [15], $W_{\text{FP8}}A_{\text{FP8}}$ [37], $W_{\text{MXFP8}}A_{\text{MXFP8}}$ [41], and $W_{\text{INT1}}A_{\text{INT8}}$ [46].
- **ResNet-50:** Evaluated with data type configurations of $W_{\text{FP16}}A_{\text{FP16}}$ [24], $W_{\text{FP8}}A_{\text{FP8}}$ [37], $W_{\text{MXFP8}}A_{\text{MXFP8}}$ [41], and $W_{\text{INT1}}A_{\text{INT4}}$ [25].
- **ShuffleNet-V2:** Evaluated with data type configurations of $W_{\text{FP16}}A_{\text{FP16}}$ [34] and $W_{\text{FP8}}A_{\text{FP8}}$ [42].
- **ViT-Base:** Evaluated with data type configurations of $W_{\text{FP16}}A_{\text{FP16}}$ [19], $W_{\text{FP8}}A_{\text{FP8}}$ [27], and $W_{\text{INT4}}A_{\text{INT4}}$ [29].
- **Conformer-L:** Evaluated with data type configurations of $W_{\text{FP16}}A_{\text{FP16}}$ [23], $W_{\text{INT8}}A_{\text{INT4}}$ [18], and $W_{\text{INT4}}A_{\text{INT4}}$ [18].

We configure various batch size (BS) and sequence length (SEQ) settings to cover diverse deployment scenarios. For large language models such as LLAMA-70B and BLOOM-176B, we conduct tests with (BS, SEQ) settings of (1, 1), (32, 1), and (1, 4096) to comprehensively represent online and offline inference scenarios, as well as pre-fill and decoding stages. Additionally, models like ResNet-50, ShuffleNet-V2, ViT-Base, and Conformer-L are evaluated with batch sizes of both 1 and 128 to assess performance across online and offline inference scenarios.

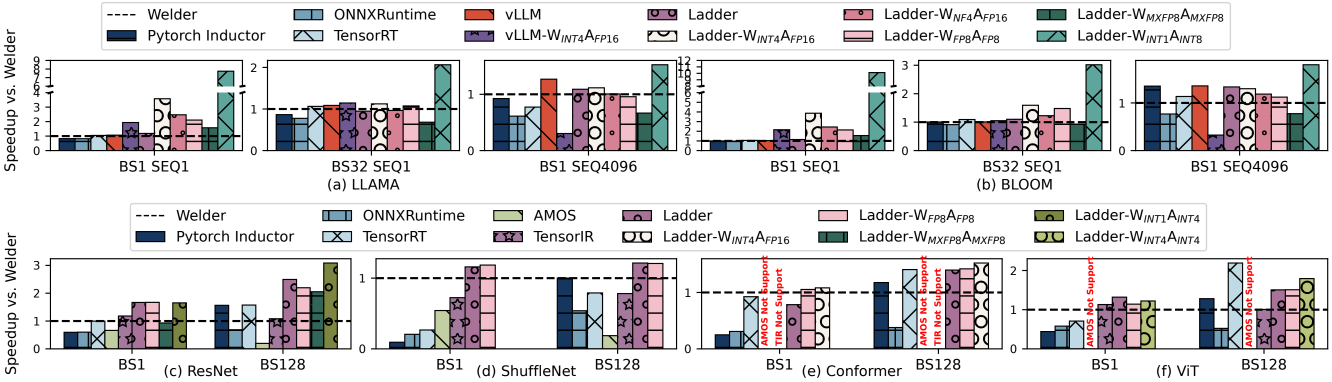


Figure 8: End-to-end performance on the NVIDIA A100 GPU.

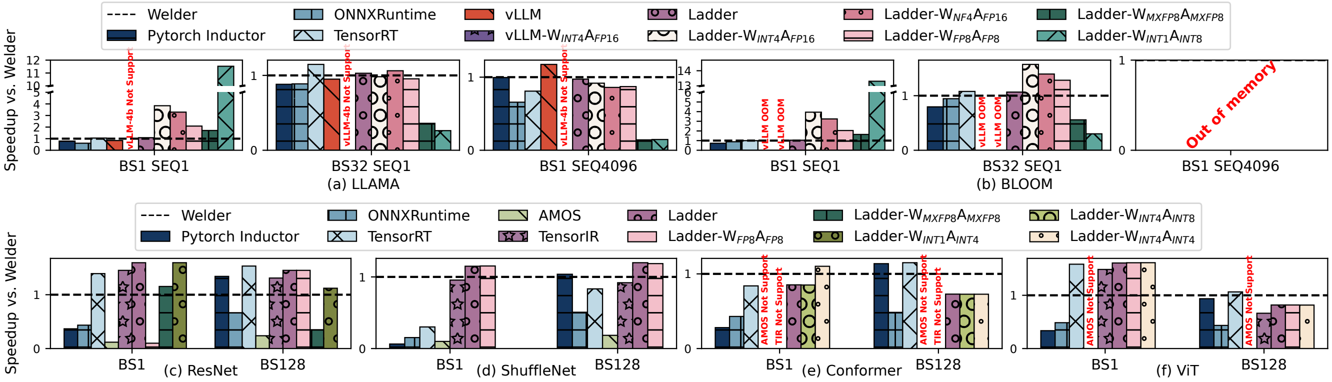


Figure 9: End-to-end performance on the NVIDIA V100 GPU.

Baselines. We compare LADDER against various well-established compilers and frameworks across different GPU platforms. For NVIDIA GPUs, we include comparisons with Welder [43], PyTorch-Inductor [38], ONNXRuntime [7], TensorRT [9], AMOS [56], TensorIR [21], vLLM [28], vLLM-$W_{\text{INT4}}A_{\text{FP16}}$ (the 4-bit quantized model support in vLLM) [28]. On AMD GPUs, we compare LADDER with Welder [43], PyTorch-Inductor [38], ONNXRuntime [7] and TensorIR [21]. To harness the MatrixCore capabilities on ROCm devices, we have integrated MIOpen and rocBLAS into Welder, and we have also enhanced TensorIR with rocWMMA Auto Tensorize support. For operator benchmarks, LADDER is evaluated against cuBLAS [4], CUTLASS [6], vLLM [28], cuDNN [5], AMOS [56] and TensorIR [21].

5.2 Evaluation on NVIDIA GPUs

5.2.1 End-to-End Performance

Inference latency. Our inference latency evaluation targets the previously detailed DNN models, executed on the Tesla A100, V100, and RTX A6000 GPUs. For large language models, such as LLAMA-70B and BLOOM-176B, due to GPU memory constraints, we evaluate the inference latency using one decoder layer of these models, which serves as a proxy for the full model's performance because each layer is the same and the latency is linear with the number of layers.

Figure 8 summarizes the inference latency results on the A100 GPU. In the data type configuration of $W_{\text{FP16}}A_{\text{FP16}}$, LADDER achieves notable performance enhancements. Compared to Welder, we report an average speedups of 1.0×, 1.2×, 2.0×, 1.2×, 1.1×, and 1.4× for LLAMA, BLOOM, ResNet, ShuffleNet, Conformer, and ViT, respectively. The reason is because Welder leverages Roller [57], cuBLAS [4] and CUTLASS [6] for kernel generations and suffers from kernel performance issues like shared memory bank conflicts, especially in ResNet where Conv2D operations introduce more irregular shapes. LADDER can achieve higher efficiency by resolving these kernel performance issues with tensor transformation scheduling, e.g., 1.1 ms and 7.6 ms latency on BS1 and BS128 of ResNet. In the data type configuration of $W_{\text{INT4}}A_{\text{FP16}}$ which is widely used in LLMs, LADDER achieves a remarkable 2.3× speedup on average over vLLM. Moreover, LADDER exhibits robust versatility by supporting custom data types not traditionally accommodated by other systems. For instance, in the case of $W_{\text{INT1}}A_{\text{INT8}}$ configuration, LADDER achieves an impressive speedup of up to 10× relative to Welder on one layer of BLOOM-176B-BS1SEQ1 with 0.32 ms latency.

Our inference latency evaluation extends to the Tesla V100 and RTX A6000 GPUs, with results shown in Figures 9 and Figure 10. The results on these platforms align closely with those observed on the A100. It is important to note that the V100, equipped with 16GB of memory, encounters limitations when handling even a single decoder layer of the BLOOM

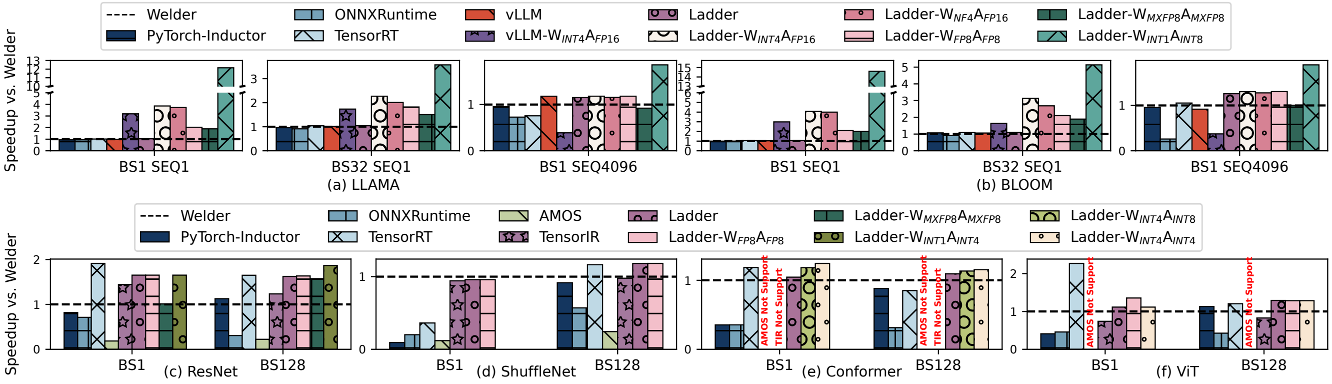


Figure 10: End-to-end performance on the NVIDIA RTX A6000 GPU.

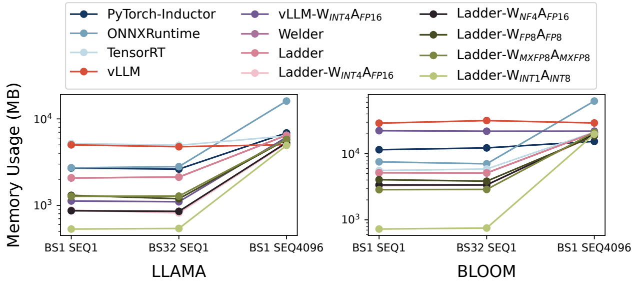


Figure 11: Memory usage of LLM inference on the NVIDIA A100 GPU across varying data type configurations.

Model (BS)	AMOS	TensorIR	Welder	LADDER
ResNet (1)	3852	156	11	31
ResNet (128)	2191	836	18	44
ShuffleNet (1)	3328	128	13	17
ShuffleNet (128)	3121	400	12	29

Table 2: Compilation time (in minutes) comparison of end-to-end models on NVIDIA A100 GPU.

model, resulting in out-of-memory errors. In terms of performance gains, with the $W_{\text{FP16}}A_{\text{FP16}}$ configuration, LADDER delivers an average speedup of 1.1× on the V100 and 1.2× on the A6000, compared to Welder on both platforms. With the $W_{\text{INT4}}A_{\text{FP16}}$ configuration, LADDER achieves an average speedup of 2.0× compared to vLLM on the A6000 GPUs, and also enables effective $W_{\text{INT4}}A_{\text{FP16}}$ inference on V100 GPUs. In scenarios utilizing the $W_{\text{INT1}}A_{\text{INT8}}$ configuration, LADDER reaches up to 13.3× speedup on the V100 and 14.6× speedup on the A6000, compared to Welder.

Memory usage. Employing reduced-precision data types is a critical strategy for alleviating the substantial memory requirements of large language models (LLMs). To quantify the benefits of this approach, we conduct a thorough investigation of memory usage across various data type configurations during LLM inference on the A100 GPU. The results are shown in Figure 11, which illustrates a near-linear decrease in memory usage corresponding to the reduction in bit width. This trend is particularly pronounced during the decoding phase with a sequence length of 1, highlighting the advantages of precision scaling in the memory-intensive decoding stage of inference. In the most extreme scenario, employing a weight precision of 1-bit and activation precision of 8-bit ($W_{\text{INT1}}A_{\text{INT8}}$), we observe substantial memory savings. Specifically, when compared to the full precision ($W_{\text{FP16}}A_{\text{FP16}}$) configuration, the memory footprint for LLAMA model inference is reduced by 74%, 74%, and 24% across three different batch size and sequence length combinations, respectively. For the BLOOM model, the memory footprint is reduced by 85%, 85%, and 6% for the corresponding settings.

Compilation time. To assess the efficiency of our system, we present a comparative analysis of compilation times in Table 2. Our evaluation compares LADDER against other prominent systems: AMOS, TensorIR, and Welder. The compilation times are measured for the end-to-end compilation of two representative neural network models, ResNet and ShuffleNet, with different batch sizes (1 and 128) on an NVIDIA A100 GPU. The results highlight that on average, LADDER demonstrates a significant reduction in compilation time compared to both AMOS and TensorIR. Notably, LADDER is an order of magnitude faster than TensorIR, and two orders of magnitude faster than AMOS. As LADDER enables supporting low precision arithmetic through tensor transformation and thus, inherently, a broader schedule space. While it allows LADDER to capitalize on the performance benefits of low-precision arithmetic, it also imposes additional overhead during the compilation process. Consequently, LADDER exhibits slightly higher compilation times compared to Welder.

5.2.2 Operator Benchmark

To assess kernel performance within LADDER, we constructed an operator benchmark incorporating commonly utilized operators from the LLAMA and ResNet models. The benchmark is composed of six matrix multiplication (MatMul) operators, labeled M0-M5, and eight 2D convolution (Conv2d) operators, labeled C0-C7. We tested each operator under a variety of data type configurations, including $W_{\text{FP16}}A_{\text{FP16}}$, $W_{\text{INT4}}A_{\text{FP16}}$, $W_{\text{NF4}}A_{\text{FP16}}$, $W_{\text{FP8}}A_{\text{FP16}}$, $W_{\text{INT1}}A_{\text{INT8}}$, $W_{\text{MXFP8}}A_{\text{MXFP8}}$, and $W_{\text{INT4}}A_{\text{INT4}}$. All experiments were executed on an NVIDIA A100 GPU to ensure consistency and

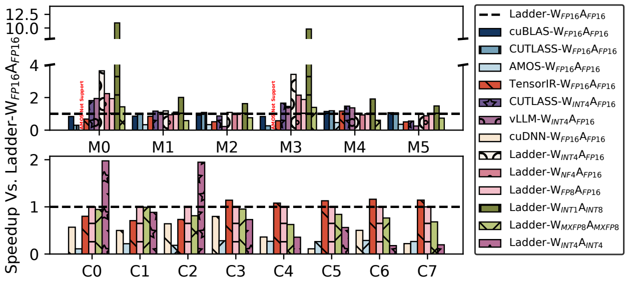


Figure 12: Operator benchmark on NVIDIA A100 GPU.

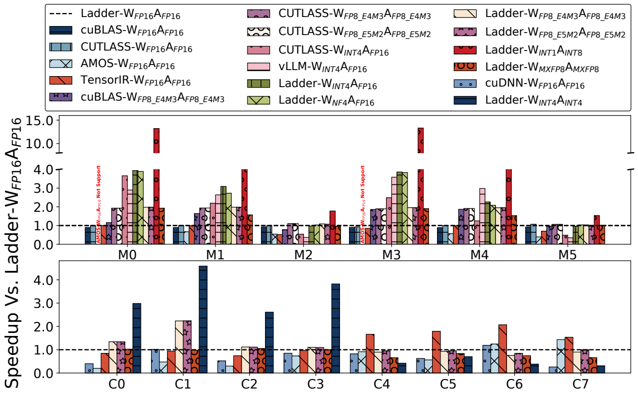


Figure 13: Operator benchmark on NVIDIA RTX 4090 GPU.

reliability in performance evaluation. As depicted in Figure 12, LADDER demonstrates optimal performance with the $W_{\text{FP16}}A_{\text{FP16}}$ configuration. Transitioning to $W_{\text{INT4}}A_{\text{FP16}}$, LADDER achieves an average speedup of 1.8×, while the $W_{\text{INT1}}A_{\text{INT8}}$ configuration enables an even further average speedup of 4.5×.

The Ada Lovelace, Hopper and Blackwell GPUs support $W_{FP8}A_{FP8}$ tensor core. We also conducted the operator benchmark on a NVIDIA RTX 4090 GPU with CUDA 12.4 to evaluate the hardware-supported $W_{FP8}A_{FP8}$ performance. Figure 13 shows the results. For $W_{FP8\_E4M3}A_{FP8\_E4M3}$, LADDER outperforms cuBLAS and achieves comparable performance over CUTLASS. For $W_{FP8\_E5M2}A_{FP8\_E5M2}$, LADDER achieves comparable performance over CUTLASS, while cuBLAS does not support this case. RTX 4090 only enables the $W_{FP8}A_{FP8}$ with FP32 accumulation which has the same theoretical performance as $W_{FP16}A_{FP16}$. Therefore, cuBLAS, CUTLASS and LADDER of $W_{FP8}A_{FP8}$ is similar to that of $W_{FP16}A_{FP16}$ on large matrices like M2 and M5. Although $W_{FP8}A_{FP8}$ with FP16 accumulation has double theoretical performance, it is not exposed by NVIDIA currently. LADDER achieves higher speedup on data types like $W_{NF4}A_{FP16}$ and $W_{INT1}A_{INT8}$ than those on A100, because RTX 4090 has more powerful cores for transforming data types.

5.2.3 Optimization Breakdown

Figure 14 illustrates the step-by-step optimizations LADDER applied to the LLAMA-70B model's kernels for both single (BS1 SEQ1) and large batch sequences (BS1 SEQ4096)

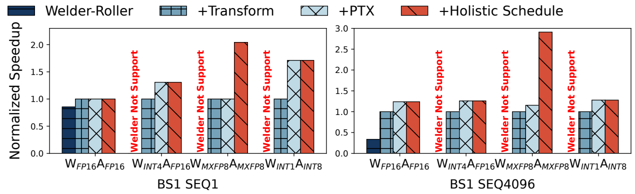


Figure 14: Optimization breakdown

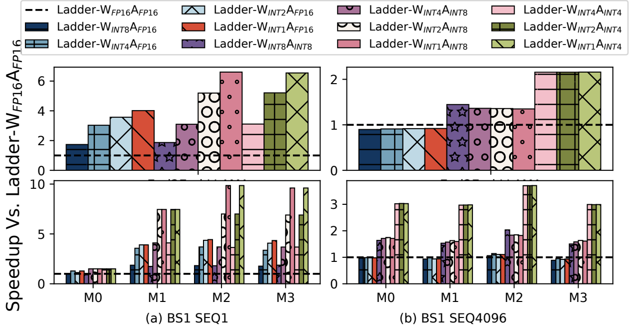


Figure 15: Scaling the bit width of weight and activation.

across different data formats. Tile-aware kernel transformation led to smoother data handling and a 2.0× speed boost over the Roller baseline, also enabling support for various data types. PTX-level optimizations reduced GPU memory load, and with advanced control over tensor operations and layout, LADDER achieved a further up to 1.7× speedup. A comprehensive scheduling strategy yielded a up to 2.5× speedup, especially benefiting memory-constrained types like MXFP8, by optimizing transformations. Overall, LADDER's optimizations enhance computational efficiency and adaptability, delivering marked performance gains across multiple operations.

5.2.4 Scaling Bit Width

Leveraging the versatile capabilities of LADDER, we are able to support a wide range of data types with arbitrary bit widths for both weights and activations. To thoroughly evaluate the performance implications of precision scaling, we conducted experiments across data type settings that progressively decrease bit widths. Our evaluation encompasses end-to-end performance as well as individual operator performance across two distinct batch size and sequence length configurations. The experimental outcomes are detailed in Figure 15. As we scale down the bit widths of W and A, we observe a corresponding escalation in speedup, reflecting the efficiency gains of lower precision arithmetic. In decoding scenarios with sequence length of 1, which are memory-bound, our experiments show a clear speedup increase with reduced W bit width (from $W_{\text{INT4}}A_{\text{INT4}}$ to $W_{\text{INT2}}A_{\text{INT4}}$, to $W_{\text{INT1}}A_{\text{INT4}}$). However, during encoding at sequence length of 4096, which is compute-bound, speedup remains unchanged across these configurations due to the reliance on higher-precision computations in mixed-precision operations.

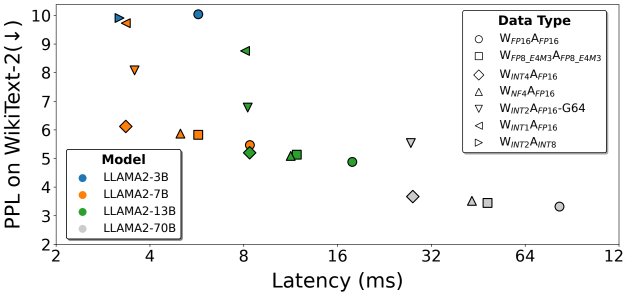


Figure 16: PPL on WikiText-2 (↓) and latency (ms) of decoding single token on A100 for different low-precision methods on LLMs. G64 of $W_{INT2}A_{FP16}$ indicates a group-wise scaling of 64 elements. LLAMA2-70B uses pipeline parallelism.

5.2.5 Efficiency and Accuracy of Low-Precision LLMs

Low-precision computing focuses on both model quality and model efficiency, thus there is usually an efficiency-accuracy trade-off in designing low-precision models. We take LLMs (i.e., LLAMA2-3B, LLAMA2-7B, LLAMA2-13B and LLAMA2-70B) as the example to evaluate both the efficiency and the accuracy of state-of-the-art low-precision methods. Specifically, we evaluated PTQ for $W_{FP8\_E4M3}A_{FP8\_E4M3}$ [2, 37], GPTQ for $W_{INT4}A_{FP16}$ [22], PTQ for $W_{NF4}A_{FP16}$ [16], BitDistiller for $W_{INT2}A_{FP16}$ [20], OneBit for $W_{INT1}A_{FP16}$ [49], and BitNet-b1.58 for $W_{INT2}A_{INT8}$ [35]. Both PTQ and GPTQ are post-training quantization methods, which does not include model training. BitDistiller and OneBit are quantization-aware training methods, leveraging distillation to achieve 2-bit and 1-bit weight quantization. BitNet-b1.58 trains LLMs from scratch to achieve ternary weights represented in $W_{INT2}A_{INT8}$.

Figure 16 shows the perplexity (PPL) on WikiText-2 and the latency of decoding single token on A100. Note that the lower PPL indicates the better model quality. The PPL of $W_{INT4}A_{FP16}$, $W_{NF4}A_{FP16}$ is reported by AFPQ [51]. The PPL of $W_{INT1}A_{FP16}$ is reported by OneBit [49]. The PPL of $W_{FP16}A_{FP16}$ on LLAMA2-3B is reported by BitNet-b1.58 [35]. The PPL of other models are evaluated with open-sourced model checkpoints and open-sourced implementations. $W_{FP8\_E4M3}A_{FP8\_E4M3}$, $W_{NF4}A_{FP16}$ and $W_{INT4}A_{FP16}$ show little affects on PPL, while achieve 1.6×, 1.7×, 2.5× on average, respectively. Quantizing LLMs to 2-bit weights with PTQ and GPTQ will result in NaN PPL [20, 49], while BitDistiller and OneBit leverage distillation to achieve stable results in 2-bit and 1-bit quantization. However, the group-wise scaling introduces extra computation cost to $W_{INT2}A_{FP16}$-G64, resulting in similar speedup as $W_{INT4}A_{FP16}$.

It is noticeable that BitNet-b1.58 achieves even better PPL with 1.8× speedup on the LLAMA2-3B configuration, when compared to the $W_{FP16}A_{FP16}$ model trained on the same dataset with same tokens [35]. This speedup does not achieve the theoretical speedup because the LLAMA2-3B is too small to saturate the GPU. We further evaluated BitNet-b1.58's $W_{INT2}A_{INT8}$ on the LLAMA2-70B configuration and achieved 4.6× speedup over $W_{FP16}A_{FP16}$, thus BitNet-b1.58 shows a good potential on both accuracy and efficiency.

When comparing across different model configurations, the model size has significant impact on both accuracy and efficiency. It is noticeable that LLAMA2-13B with $W_{INT4}A_{FP16}$ achieved better performance than LLAMA2-7B with $W_{FP16}A_{FP16}$ on both accuracy and efficiency, and the quantized LLAMA2-7B models also outperform LLAMA2-3B with $W_{FP16}A_{FP16}$ on both accuracy and efficiency. This shows the power of low-precision computing.

The community is actively exploring low-precision computing, and we hope LADDER can help researchers to explore this direction by providing feedback on efficiency.

5.3 Evaluation on AMD GPUs

We evaluate the efficient LADDER on AMD Instinct MI250 GPU by comparing it with Welder, PyTorch-Inductor and ONNXRuntime. Figure 17 shows the end-to-end performance of 6 models. In the data type of $W_{FP16}A_{FP16}$, LADDER achieves an average 2.1×, 2.35×, 1.5×, 10.5×, 1.6×, and 1.5× speedup over Welder for LLAMA, BLOOM, ResNet, ShuffleNet, Conformer, and ViT, respectively. Welder does not perform well on ShuffleNet because it leverages rocBLAS and MIOpen for matrix core and thus breaks fusion opportunities. LADDER not only generates efficient computing kernel for matrix core but also enables more fusion opportunities, resulting in 14.1× speedup over Welder on ShuffleNet-BS1 with 0.43 ms latency. In the data type configuration of $W_{INT4}A_{FP16}$ for LLMs, LADDER achieves up to 3.8× speedup on LLAMA with 0.73 ms latency on BS1SEQ1 and 4.5× speedup on BLOOM with 1.75 ms latency on BS1SEQ1 over Welder.

6 Discussion

LADDER's current implementation mainly focuses on model inference. We discuss some LADDER's limitations and future work in this section.

Multi-GPU serving. Multiple GPUs are required to deploying some large-scale models like BLOOM-176B and LLAMA2-70B, becuase these models cannot fit into a single GPU. Multi-GPU support is complementary with LADDER. LADDER focuses on supporting low-precision computing on a hardware accelerator. Multi-GPU frameworks [28, 30, 32, 53] focus on partitioning model and scheduling parallel computation across multiple GPUs. LADDER can collaborate with multi-GPU frameworks to enable parallel computation for low-precision models on multiple GPUs that multi-GPU frameworks partition a model and schedule the partitioned computation to LADDER on a device for execution. We leave integrating LADDER with multi-GPU frameworks to our future work.

Low-precision training. LADDER's design is not limited to inference. Both training and inference of low-precision

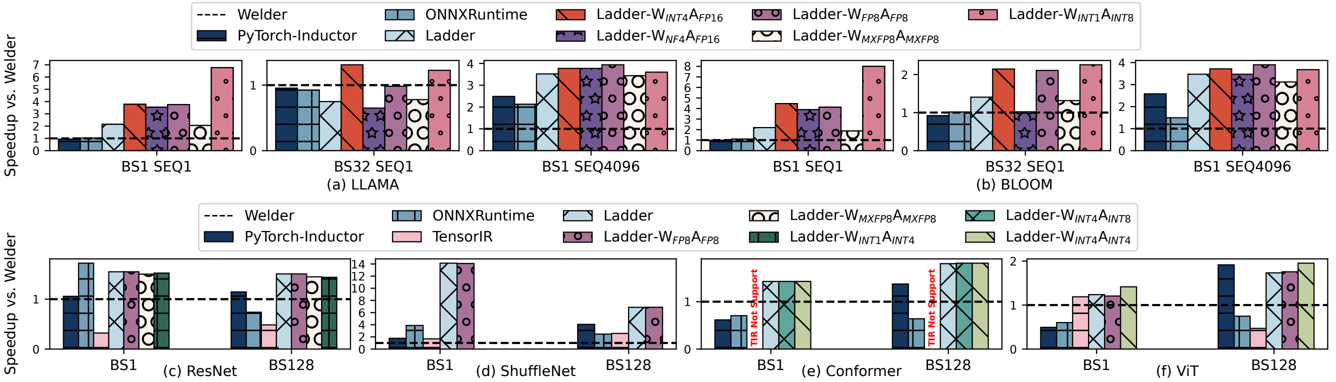


Figure 17: End-to-end performance on AMD Instinct MI250 GPU.

models require low-precision support of system and hardware. And the backward computation in training is similar to the forward computation. Low-precision model training can achieve gains from: 1) leveraging more efficient low-precision computation units, e.g., $W_{INT8}A_{INT8}$ tensor core supported on A100 has 2× throughput than that of $W_{FP16}A_{FP16}$, while $W_{INT4}A_{INT4}$ tensor core has 4×; and 2) less memory footprint from low-precision model representation enabling larger batch sizes which may improve the hardware utilization. We leave low-precision training to our future work.

7 Related Work

Deep learning compilers and frameworks. Most existing deep learning compilers, such as [10, 13, 33, 38, 43, 50, 52, 54, 57], focus on operator or model computation optimizations for mainstream data types, e.g., FP16 or FP32, with little emphasis on low-precision data types. However, many optimizations are complementary with low-precision computing, for example, Roller [57] is leveraged to infer efficient *t*Tile configurations, and Welder [43] is leveraged for end-to-end graph optimization in Ladder. SparTA [55] treats model pruning and quantization as model sparsity to holistically optimize sparse model inference and training, and Ladder can provide efficient low-precision kernels to further improve the performance. AMOS [56] has optimized for TensorCore computation, covering FP16 and INT8 types, but it is specific to NVIDIA GPUs. In comparison, Ladder is the first compiler to optimize for general low-precision computations that support general custom data types on different GPUs. Deep learning libraries or frameworks like ONNXRuntime [7] and TensorRT [9] support some low-bit operators for inference scenarios, but their coverage is still limited due to the significant effort required to implement those combinatorial cases. Some recent compilers like Triton [44] and TensorIR [21] allow users to directly write the computation pipeline of a DNN operator, providing flexibility in specifying scheduling in each stage. However, these compilers mostly focus on computation scheduling and have little support in data scheduling for custom data types, which is the primary focus of Ladder.

Model-specific low-precision optimization. Given the lacking efficient support of low-precision in existing compilers and frameworks, many works have conducted workload-specific low-precision optimizations. For example, some quantization and model training on low-precision types are optimized for Large Language Models (LLMs) [22, 28, 31, 35, 41, 45–47]. Previous work like [23–25, 34, 42] optimizes other models like ShuffleNet, Conformer, etc., into FP8 or FP16 precision. In comparison, Ladder provides a mechanism to allow one to more easily implement custom data types and optimization policies. Thus, these optimization approaches are complementary to Ladder, as they can be implemented or automatically optimized in Ladder.

8 Conclusion

In conclusion, this paper introduces Ladder, the first deep learning compiler designed to optimize general low-precision computation on accelerators like GPUs. Ladder exposes a general type system (*t*Type) and an extended tensor expression, enabling users to easily implement and express new data types in deep learning. It introduces a set of new tensor scheduling primitives to facilitate optimizations like tensor storage, access, and type conversions in a computing pipeline. The layer-wise hardware-aware optimization policy of Ladder navigates the complex transformation space, showcasing its capability to systematically support a wide array of low-bit precision custom data types. This enhances DNN computation performance on modern accelerators without requiring hardware modifications. This innovation empowers model designers to explore data type optimizations and offers hardware vendors a flexible solution to expand support for diverse precision formats.

Acknowledgement

We thank anonymous reviewers and our anonymous shepherd for their extensive suggestions.

References

[1] AMD ROCm Platform. https://github.com/RadeonOpenCompute/ROCm.

[2] AutoFP8. https://github.com/neuralmagic/AutoFP8.

[3] NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[4] NVIDIA cuBLAS. https://developer.nvidia.com/cublas.

[5] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[6] NVIDIA cutlass. https://github.com/NVIDIA/cutlass.

[7] Onnx runtime. https://www.onnxruntime.ai.

[8] PTX ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[9] Tensorrt. https://github.com/NVIDIA/TensorRT.

[10] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[12] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. Quip: 2-bit quantization of large language models with guarantees. *arXiv preprint arXiv:2307.13304*, 2023.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11(20), 2018.

[14] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[15] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[16] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[18] Shaojin Ding, Phoenix Meadowlark, Yanzhang He, Lukasz Lew, Shivani Agrawal, and Oleg Rybakov. 4-bit conformer with native quantization aware training for speech recognition. *arXiv preprint arXiv:2203.15952*, 2022.

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[20] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. *arXiv preprint arXiv:2402.10631*, 2024.

[21] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[22] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[23] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer:

Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Kun Huang, Bingbing Ni, and Xiaokang Yang. Efficient quantization for neural networks with binary weights and low bitwidth activations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3854–3861, 2019.

[26] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[27] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. Fp8 quantization: The power of the exponent. *Advances in Neural Information Processing Systems*, 35:14651–14662, 2022.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[29] Yanjing Li, Sheng Xu, Baochang Zhang, Xianbin Cao, Peng Gao, and Guodong Guo. Q-vit: Accurate and fully quantized low-bit vision transformer. *Advances in Neural Information Processing Systems*, 35:34451–34463, 2022.

[30] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[31] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[32] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 2024.

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[35] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

[36] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[37] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[39] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, et al. Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313*, 2023.

[40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[41] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.

[42] Haihao Shen, Naveen Mellempudi, Xin He, Qun Gao, Chang Wang, and Mengni Wang. Efficient post-training quantization with fp8 formats. *arXiv preprint arXiv:2309.14592*, 2023.

[43] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, Boston, MA, July 2023. USENIX Association.

[44] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[46] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

[47] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[48] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[49] Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. Onebit: Towards extremely low-bit large language models. *arXiv preprint arXiv:2402.11295*, 2024.

[50] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, 2023.

[51] Yijia Zhang, Sicheng Zhang, Shijie Cao, Dayou Du, Jianyu Wei, Ting Cao, and Ningyi Xu. Afpq: Asymmetric floating point quantization for llms. *arXiv preprint arXiv:2311.01792*, 2023.

[52] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[53] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[54] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, et al. PIT: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 331–347, 2023.

[55] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-learning model sparsity via tensor-with-sparsity-attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, 2022.

[56] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. Roller: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

CARAVAN: Practical Online Learning of In-Network ML Models with Labeling Agents

Qizheng Zhang, Ali Imran[†], Enkeleda Bardhi[‡], Tushar Swamy, Nathan Zhang,
Muhammad Shahbaz[†⋆], Kunle Olukotun
Stanford University [†]*Purdue University* [‡]*Sapienza University of Rome* [⋆]*University of Michigan*

Abstract

Recent work on in-network machine learning (ML) anticipates offline models to operate well in modern networking environments. However, upon deployment, these models struggle to cope with fluctuating traffic patterns and network conditions and, therefore, must be validated and updated frequently in an online fashion.

This paper presents CARAVAN, a practical online learning system for in-network ML models. We tackle two primary challenges in facilitating online learning for networking: (a) the automatic labeling of evolving traffic and (b) the efficient monitoring and detection of model performance degradation to trigger retraining. CARAVAN repurposes existing systems (e.g., heuristics, access control lists, and foundation models)—not directly suitable for such dynamic environments—into high-quality labeling sources for generating labeled data for online learning. CARAVAN also introduces a new metric, *accuracy proxy*, to track model degradation and potential drift to efficiently trigger retraining. Our evaluations show that CARAVAN's labeling strategy enables in-network ML models to closely follow the changes in the traffic dynamics with a 30.3% improvement in F1 score (on average), compared to offline models. Moreover, CARAVAN sustains comparable inference accuracy to that of a continuous-learning system while consuming 61.3% less GPU compute time (on average) via accuracy proxy and retraining triggers.

1 Introduction

Machine learning (ML) is being increasingly leveraged to better manage and operate networks today [27,30,35,37,45,49,54,77,80,82,97,101,104,105,111,114,116]. In academia, several proposals make a case for using ML to improve systems security through anomaly and intrusion detection [30,54,105] and to optimize systems performance through inference, diagnosis, and forecasting of systems' behavior [49,76,80,81]. Correspondingly, in industry, ML is being deployed to detect threats and bots in public and enterprise-scale cloud networks [1,9,10] for securing physical and virtual infrastructure and for providing better user experience by predicting network incidents and congestion early on [8]. Moreover, to operate at scale, with high throughput and low latency, the model inference is being offloaded to the data plane (e.g., programmable switches [15,114] and SmartNICs [17,101]) in the network (i.e., in-network ML)—to perform decision-making on a per-packet basis [37,104,122].

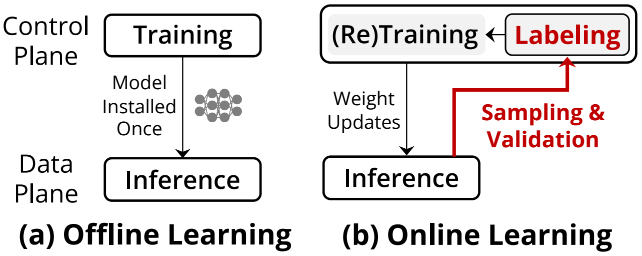


Figure 1: Comparison of in-network model learning. (a) Offline learning: trained and deployed once; (b) Online learning: trained and updated over time—requires iterative sampling, labeling, and validation.

Unlike conventional approaches (e.g., hand-crafted heuristics and static rulesets), ML models are better at revealing hidden patterns and characteristics in vast amounts of high-dimensional data—such as network traffic [35,54,71,104,111,116,117]. However, most efforts on replacing traditional approaches (e.g., heuristics and access control lists) with ML [35] are limited to using static models (aka offline learning, Figure 1a) [30,35,37,49,54,82,97,101,104,105,111,114,116]. These models are trained once using synthetic or controlled network traces and are expected to operate well in the real environment without further guidance (or retraining). While showing significant promise in stable (less volatile) environments, these static models perform poorly in the presence of fluctuations and unforeseen events—not captured by the traffic during the initial training phase [33,115,117]. These manifest as model drift either (a) when the network environment gradually evolves or suddenly changes due to traffic bursts, time-of-day, or rare events (called concept drift) [117] or (b) when new data patterns arrive or data distribution changes (called data drift) [33]. This model drift is shown to be prevalent in many online ML applications [33,94,100,115].

To keep these models up-to-date with new patterns and network behavior, one approach is to train and update them continuously on the incoming traffic—referred to as online learning or continuous learning [33,94]. For example, a (re)training pipeline in the control plane can continuously sample packets from the network (e.g., using INT [68] or NetFlow [7]), label them, and pass them to the model for retraining (Figure 1b). It then updates the weights on the data-plane device, performing model inference. As we show in our evaluations (§5), keeping the model current through online training allows it to handle new incidents with much higher accuracies compared to the static offline models (i.e., the average difference in accuracy is as high as 67%).

However, there are a number of challenges when it comes to enabling continuous learning in modern networking environments (processing Tbps of traffic for varying tenants and workloads) [32, 96, 119, 121]. First, unlike traditional online learning systems in other domains (e.g., recommendation systems and financial systems) where the new retraining data either contains labels (ground truth) [59, 103, 113] or can be easily labeled using existing approaches (like Data Programming [93] or Weak Supervision [91,92]), in networking the incoming data is raw (sampled) traffic with no labels. *Challenge #1: How can we prepare (and label) traffic data for retraining in-network models?* Second, we cannot rely on fixed interval-based or periodic retraining to ensure the installed models perform well. The network conditions are highly dynamic and erratic; a large interval will miss such variations, whereas frequent updates would be too costly in terms of resource usage (CPU/GPU cycles and network bandwidth). *Challenge #2: How to decide when to trigger retraining?*

In this paper, we present CARAVAN, an online learning system for in-network ML to tackle these challenges. To label new incoming network traffic, CARAVAN relies on labeling agents that use different user-defined knowledge sources to assist with labeling. In networking, many existing systems, such as heuristics, access control lists, deep learning, or even foundation models (e.g., GPT-4 [87], Gemini [106], Llama 3 [16]), fare poorly when used for real-time decision-making—they either fail to adapt to changing network conditions or take too much time to process. However, we observe that these can be used as knowledge sources to label incoming traffic for online learning of in-network models. For instance, using foundation models (which encode a broad spectrum of information about the environment [89,95,112]) and guidance from users (e.g., prompts [28] and document retrievals [72]), we can generate application-specific, weak-supervision labels to (re)train these models. We also introduce a new metric, *accuracy proxy*, to decide when to trigger retraining. Instead of relying on ground-truth labels to compute model accuracy, we compute accuracy proxy based on generated labels we receive from the labeling agents for model (re)training. Doing so allows CARAVAN to track degradation in model behavior through relative changes in the accuracy level on a temporal scale, and to trigger retraining. More specifically, if there is an abrupt change in the accuracy proxy (i.e., model drift exceeds a certain threshold), CARAVAN uses this as a signal to trigger retraining. This limits CARAVAN from excessively retraining the model under normal conditions.

We evaluate our CARAVAN system both in simulation (for microbenchmarks) and with a Taurus FPGA-based switch [104] (for end-to-end results). Our simulation results show that labels generated using knowledge sources perform similarly to ground-truth labels in terms of inference accuracy when used to label incoming traffic for retraining. Moreover, our accuracy proxy and retraining triggers save up to 74.55% GPU compute time compared to continuous online training while sustaining similar accuracy gains. With our Taurus FPGA testbed, we show that CARAVAN maintains 30% higher accuracy on average compared to offline models while using 56.23% less CPU and with similar memory footprint compared to continuous retraining baselines—with CARAVAN, the model operates at line rate while adapting to changing traffic dynamics.

In summary, we make the following contributions:

- We present CARAVAN, a practical online learning system for in-network ML. CARAVAN's labeling-agent strategy allows the use of existing network systems (e.g., heuristics, access control lists) and emerging foundation models (e.g., GPT-4, Gemini, and Llama 3) as knowledge sources to label incoming traffic. Using accuracy proxy further allows CARAVAN to efficiently retrigger the training pipeline while closely tracking changes in the network conditions.
- We implement CARAVAN as a software logic running in the control plane, and test it both in a simulation setting and using a real testbed with Taurus FPGA-based switches. Our CARAVAN prototype is available as open-source.[1]
- Our evaluations show that CARAVAN allows in-network models to track changes in the network at line rate while sustaining 30.3% higher F1 score (on average) compared to offline systems. Moreover, it consumes 61.3% less GPU compute time (on average) than a continuous-learning system by selectively triggering retraining via accuracy proxy.

2 Background & Motivation

In-network Machine Learning. Network operators face many challenges with managing the size and complexity of modern networks while maintaining their stringent (and ever-increasing) performance requirements [32,96,119,121]. Over time, the networking community has developed a plethora of hand-tuned heuristics permeating the network, which continuously introduce new parameters that must then be tuned to the given network (and workload). We see this with the constant iterations of congestion-control variants [55,73], active-queue management [23], load balancing algorithms [26,66], anomaly detection [25, 34, 38] and more. Relying on network developers and researchers to keep adding new parameters to each algorithm being used throughout the network, as the workloads change and evolve, has limited scalability as networks grow. Networking researchers have, therefore, begun to turn toward data-driven algorithms, in the form of ML, particularly deep-learning and neural networks [35, 40, 63, 80, 83, 105, 111, 116]. Rather than tuning individual model weights by hand, ML algorithms take training data as input and learn model weights to optimize for performance metrics (e.g., prediction accuracy).

To operate at scale with high throughput and low latency, these models are further offloaded to the network data plane

[1] Artifact: https://github.com/Per-Packet-AI/Caravan-Artifact-OSDI24

(e.g., programmable switches and SmartNICs) [17, 101, 104, 114, 125]. Doing so allows more fine-grained control over the traffic, with decision-making (and model inference) taking place at or near the packet level. For example, programmable switches (e.g., Intel Tofino series [15]) with match-action tables (MATs) can perform ML algorithms (such as SVMs and decision trees) [37, 114], with more recent data-plane devices incorporating MapReduce-based processing blocks to run deep neural networks (DNNs) directly in the network [104]. Likewise, emerging SmartNIC devices (e.g., Marvell Octeon 10 [17] and Xilinx SN1000 [2]) come equipped with on-board ML inference engines for per-packet inference. Similarly, data/infrastructure processing units (DPUs/IPUs) from Nvidia [6], AMD [4], and Intel [14] also provide computational resources capable of running ML inference alongside the packet-processing pipelines.

Online Learning and Model Drifts. Recent work on applying online learning in networking domains (such as video analytics and edge monitoring) shows promising results. For example, Ekya [33] and RECL [67] demonstrate that retraining computer vision models for video analytics applications with new video frames can effectively mitigate data drift for compressed ML models. Nazar [58] features online monitoring and adapts various ML models on mobile devices to relieve the problem of potential model drifts.

Through retraining ML models with new incoming data, online learning addresses two common issues these models face post-deployment: concept drift [117] and data drift [33]. Concept drift occurs as networks and traffic are subjected to dynamic signal interference due to environmental changes [123] (e.g., weather, temperature, or time-of-day), as well as changes in the network and user behavior (e.g., increased online activity during COVID-19 [46], addition/removal of devices and software due to upgrades or failures [75]). For example, a large file download may be classified as benign during the day when networks are more active but are marked as malicious during the night when the number of high-volume flows is smaller. On the other hand, data drift happens when the live traffic (or data) distribution diverges from the training data distribution after the model is deployed [33, 94]. For classification models, in particular, the arrival of new data classes (not already present in the offline training set) or a change in data class distribution could cause an ML model to perform poorly [33, 94]. For example, in network security, new attacks come up without warning, and it becomes challenging for a static ML model to detect such an attack since it was not trained on data featuring the new attack.

Network Data Labeling. The emerging interest in training and testing ML models for networking applications sparks extensive research in the area of obtaining labeled network data [42, 60, 98, 99]. Most recent work falls into three different categories: generating labeled network data in a controlled environment, synthetic data generation, and manual labeling through domain experts (i.e., network operators).

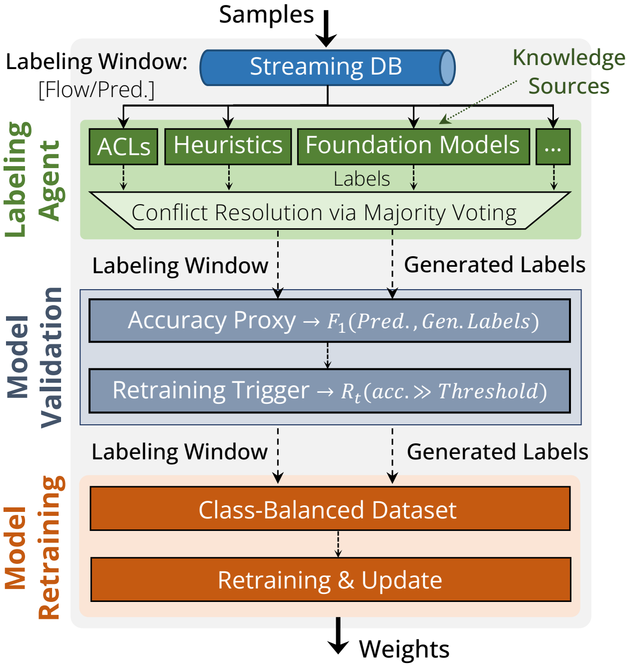


Figure 2: High-level design of CARAVAN. The three key components, Labeling Agent (§3.1), Model Validation (§3.2), and Retraining, work in tandem to keep the in-network ML model up-to-date.

Efforts like NetUnicorn [31] propose to collect and actively label network data in a controlled environment where operators can access different nodes (switches and hosts) in the network. Though labeling accuracy would be high since operators can choose to generate and collect selected traffic classes, this approach might not offer representative labeled data in real networks [53]—limiting its use in online learning. Other efforts feature synthetic data generation, where models like GANs [118] or diffusion models [65] are used to produce packet traces that match the feature distribution of input network data. However, the generation process takes a lot of time and cannot explicitly label new incoming data, making it impractical in an online setting. Also, it is unclear how closely the synthetic data reflects the traffic in a real environment (an open area of research [57, 107, 109]). The last resort is to ask human experts with domain knowledge to label all or selected network data. Though there are many efforts featuring selecting sampled data for human experts to label [53], this still requires a human-in-the-loop and may not operate at the timescales needed for automatic data labeling in networks.

3 Design of CARAVAN

We present CARAVAN, a system for practical online learning of in-network ML models deployed in the data plane. CARAVAN is designed to satisfy the following requirements: (1) generation of effective label datasets for retraining and (2) efficient monitoring and detection of model performance.

Overview. Figure 2 shows the high-level architecture of CARAVAN. The system periodically collects a window of samples, arriving from the data-plane device running in-network ML inference. Each sample contains a set of header fields (called flow) along with the prediction made by the deployed model. Once a window is full, a labeling agent (§3.1) generates application-specific (e.g., security) labels for each sample in the window in the form of class predictions (e.g., type of network attack) or confidence scores (e.g., the likelihood a flow being malicious). The agent relies on a collection of knowledge sources (§3.1.1), each generating its own labels. The label with the most votes (i.e., occurrences) is added to the final label set. Next, the validation stage (§3.2) monitors and detects the degradation in the model performance using a new metric, called *accuracy proxy* (§3.2.1), which uses predicted values and generated labels to measure the model's accuracy on the received samples in the window. Based on the accuracy values (e.g., exceeding a certain threshold), the retraining trigger stage decides whether retraining is necessary (e.g., in the presence of new types of attacks missed by the in-network ML model) for the current window of incoming samples. If an update is required, the final model retraining stage will generate a balanced dataset from the window of samples received, i.e., a mix of malicious and benign flows and generated labels. After training is complete, the in-network model is updated with the new weights to detect the new types of missed attacks.

3.1 Labeling Agent

The first component in CARAVAN is the labeling agent. It generates application-specific labels that can be used in the later stages of model validation and online retraining for: (1) computing an accuracy proxy that can signal potential model accuracy degradation to efficiently trigger retraining, and (2) generating a class-balanced labeled dataset for retraining when necessary. To generate labels for new incoming network traffic automatically and accurately, the labeling agent relies on external knowledge sources. Knowledge sources (§3.1.1) are defined to be entities or applications that can be repurposed to assist with data labeling (e.g., access control lists, heuristics, foundation models). They can be defined and provided by users through a user interface (§3.1.2).

When a full window of samples from the data plane is available, the labeling agent reads these samples and the associated inference results from a streaming database (e.g., InfluxDB [13] or Apache Kafka [5]). Then, it sends a labeling request to every knowledge source it relies on. With one label from each knowledge source, the labeling agent would do a majority voting to determine the final set of labels to be used (also called *generated labels*) for the current window of data samples. These generated labels will be sent to the next stage of CARAVAN for model validation (Figure 2).

3.1.1 Knowledge Sources. The labeling agent relies on knowledge sources for labeling. We define knowledge sources to be any entities or services that contain useful information about the user-defined application and can be repurposed to assist data labeling. Take network intrusion detection as an example. A common knowledge source is IP-based access control lists (ACLs) [52] that can block network flows or packets from certain source IP addresses considered to be of malicious origins. With IP-based ACLs as a knowledge source, the labeling agent is able to label a flow as *malicious* if its source IP is on the list. Another example is foundation models. With appropriate adaptation, a foundation model can assist with downstream tasks in networking, such as traffic classification, by functioning as a multi-class classifier [54].

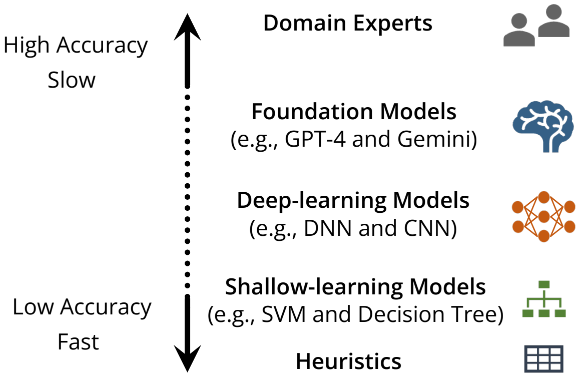


Figure 3: Classifying network knowledge sources across a spectrum based on accuracy and speed.

CARAVAN repurposes different knowledge sources in different ways for them to be used in the system for accurate and efficient labeling. In particular, CARAVAN focuses on two metrics of a knowledge source: (1) accuracy, which refers to how accurate the knowledge source is after being repurposed for labeling, and (2) speed (throughput and latency), which refers to how fast a knowledge source can be used to label data. Different knowledge sources can vary dramatically in these two metrics, and as illustrated in Figure 3, there exists a trade-off between these two metrics for a given knowledge source: sources with high accuracy (e.g., domain experts and foundation models) usually operate with lower throughput due to extra time needed for in-depth analysis, while high-speed sources (e.g., heuristics, IP-based ACLs) might not be able to provide accurate labels. The aim of CARAVAN is that through online learning, an in-network ML model can be turned into a model with both *high accuracy* and *high speed*, so it would be a good fit for real-time decision making in the data plane.

Low-Accuracy, Fast Knowledge Sources. Knowledge sources with a low accuracy but high speed (e.g., heuristics, IP-based ACLs) are a good fit for labeling large volumes of incoming data [24, 51, 108]. However, the main issue is that generated labels would be extremely noisy in this case. If we use these labels to retrain the in-network ML model, the accuracy of the retrained model can be even worse than that of the existing one in the data plane. To tackle this challenge, CARAVAN adopts the following solution: Instead of letting these low-accuracy knowledge sources provide a label for every unit of data in the current window, CARAVAN's labeling agent will ask these knowledge sources to label parts of data. These generated labels that cover a part of the dataset are usually called weak-supervision labels in the machine-learning community, and can reduce the amount of noise in the labels [92]. For example, when we use an IP-based ACL as a knowledge source for labeling, a straightforward way of repurposing it into a labeler is that we would label every flow with IP not on the blacklist as a benign flow. However, we could have mislabeled a lot of malicious flows as benign in this case and manually introduced a lot of noise into the final set of labels. With CARAVAN's solution of generating weak-supervision labels, we would only generate labels for flows whose IPs are on the blacklist (as we are more confident they would be malicious). Even though we would only be able to obtain a much smaller set of labeled data for model validation and retraining, with a large volume of incoming data, we would end up with a reasonable amount of labeled data with good accuracy (§5).

> **Insight 1:** *Low-accuracy but fast knowledge sources, such as heuristics and IP-based ACLs, can provide weak-supervision labels for training high-accuracy models.*

High-Accuracy, Slow Knowledge Sources. Knowledge sources characterized by low speed but high accuracy (e.g., domain experts and foundation models) are well-suited for labeling a small to medium amount of incoming data considered important or representative of the network (e.g., sampled data with network telemetry algorithms). For example, foundation models, like NetFound [54] and ChatGPT [20], are shown to be capable of solving downstream traffic analysis tasks with high accuracy and generalizing well across diverse network environments with no extra retraining. The main issue, however, is that they might either be too slow or use too many system resources (e.g., GPU/CPU memory and API costs) and thus cannot be activated frequently (for instance, at the end of every window of sampled data).

One insight that CARAVAN takes advantage of is that these knowledge sources can usually be transformed into cheaper rulesets or heuristics that are able to offer much higher throughput due to low latency or cost-effectiveness. In the machine-learning community, this insight was originally used to interpret black-box ML models [48, 62]. In CARAVAN, to avoid the costs associated with calling expensive knowledge sources at every labeling window, we introduce a *labeling rule cache*. Each time the knowledge source is activated for labeling, it is also asked to generate an ensemble of rulesets that will be stored in the labeling rule cache for fast and cheap labeling at the end of the next few labeling windows. For example, though foundation models, like GPT-4 [87], can be repurposed as a labeling source, the fees incurred by calling the GPT-4 APIs for inference can grow prohibitively expensive if we call these APIs at the end of every labeling window—GPT-4 turbo [11] can cost as high as $144 an hour for 1000-token labeling request/response per second (on average). CARAVAN specifically asks the language model to generate rules it relies on for decision-making and stores these in the rule cache for labeling the next few windows of data. Note that, in the evaluation section (§5), though we demonstrate that using the rule cache for labeling could save a lot on cost and system resources with little performance penalty, we also find that these rules could go stale quickly (Figure 7) and, therefore, must be updated occasionally.

> **Insight 2:** *High accuracy but slow knowledge sources, such as ChatGPT [20] and NetFound [54], can transform into rulesets or heuristics to facilitate fast and resource-efficient labeling for a limited duration, before becoming stale.*

3.1.2 User Interface. CARAVAN's labeling agent exposes an interface where users can conveniently specify what knowledge sources they would like to use for the labeling agent and how the labeling sources should be defined. To support a new knowledge source, the user only needs to complete a function called `label()`, which takes a window of data samples as input and returns a set of labels on this window as output.[2]

3.2 Model Validation

The model validation stage periodically monitors and evaluates the performance of the in-network ML model. It is also responsible for triggering online training when necessary, e.g., in the case of a potential concept drift or data drift when the performance of the model degrades due to changes in the network environment or due to new incoming classes. These actions take place at the end of a labeling window, after the labeling agent has generated labels for all data (samples) in the current window.

Next, we introduce two components for model validation that the user can define to express their intent or performance goal of the chosen application. (a) Accuracy proxy (§3.2.1) allows the user to specify what *signals* they would like to capture on a temporal scale from the generated labels and the inference results (e.g., drop in overall classification accuracy, the appearance of a particular type of new class, and more). (b) Retraining trigger (§3.2.2) allows the user to specify at what *occasion* they would like to initialize online training based on the observed signals through the accuracy proxy (e.g., retrain when model performance degrade or retrain when certain types of attacks show up).

3.2.1 Accuracy Proxy. We define accuracy proxy as the inference accuracy computed with generated labels as the reference ground truth, which we describe in detail below.

Ideally, for a given sample of incoming data (e.g., a network flow or a packet), the corresponding inference result

[2] As a case study, we show how to construct a new knowledge source for intrusion detection using LLMs in §4.1.

(e.g., in the form of a class label prediction, noted as ML_{labels} below) from the in-network ML model would be compared with the ground truth label in the validation stage. Ground truth labels (noted as $GndT_{labels}$ below), also called "golden" labels, are objectively correct reference results for the given application and are usually used to compute the performance accuracy of ML models. Acquiring such labels is typically challenging in practice as it necessitates domain knowledge from human experts, requiring a costly and time-consuming labeling process [53]. Moreover, during the online stage of in-network ML, where the volume of data for validation is immense, it is infeasible to obtain the ground truth labels for all new incoming data and calculate the actual performance accuracy of the in-network ML model. In CARAVAN, we instead utilize generated labels ($GenL_{labels}$) and compute the accuracy proxy for the current window of incoming data. For instance, in the intrusion detection case, using F1 score [50] as the performance metric, the real accuracy Acc_{real} is computed as follows:

$$Acc_{real} = F_1(ML_{labels}, GndT_{labels}) \tag{1}$$

The accuracy proxy, on the other hand, is computed with generated labels as ground-truth labels:

$$Acc_{proxy} = F_1(ML_{labels}, GenL_{labels}) \tag{2}$$

The accuracy proxy does not need to be defined in terms similar to the real accuracy. The user has the flexibility to define accuracy proxy to be any function as long as its definition is consistent with the user's intent or the application's performance goal, e.g., to signal potential concept or data drifts.

Without access to real accuracy values, we are unable to know the absolute accuracy level of the in-network ML model at the end of a labeling window. However, in our design, the primary responsibility of the validation stage is monitoring: it is expected to reveal potential model performance degradation and trigger online training, instead of giving users or operators the exact accuracy numbers of the in-network ML model.

In particular, we observe that accuracy proxy, though not numerically the same as the real performance accuracy, could signal a potential change in data distribution or class distribution based on its trend on a temporal scale. In our evaluation using the intrusion-detection example (§5.2.2), we observe that the arrivals of new types of attacks (unseen by the in-network ML model before) cause a drop in the relative level of accuracy on a temporal scale, and the values from accuracy-proxy can reveal that incident (Figure 8).

> **Insight 3:** *The accuracy proxy reveals potential concept and data drifts by capturing similar patterns of relative changes in accuracy levels as observed in real accuracy.*

3.2.2 Retraining Trigger. The goal of continuous model validation is to enable updating the in-network ML model through online training as and when necessary. The model validation stage will activate online training through a user-defined retraining trigger. A retraining trigger can take one of the following three forms, as pre-specified by the user of CARAVAN:

- **Window-based:** Retrain periodically once every X labeling windows. When $X = 1$, CARAVAN will perform continuous training for every window, similar to the approach in prior works [33, 85]. For window-based triggers, the validation stage will skip accuracy proxy since the trigger does not use it.
- **Accuracy-based:** Retrain if the values of accuracy proxy satisfy a certain pattern on a temporal scale. For example, users can set certain accuracy thresholds, and the retraining trigger will initialize retraining if the values of accuracy proxy continuously stay below the threshold.
- **Event-based:** Retrain when a particular event takes place, e.g., when the labeling agent or the human operator detects a particular type of attack.

The retraining trigger should ideally be defined together with accuracy proxy by the user: While accuracy proxy is able to catch meaningful signals (e.g., F1 score drop) on a temporal scale, the retraining trigger explicitly expresses at what occasions the user would like online training to happen, which can be very different given the particular user application in consideration.

In CARAVAN, we mainly focus on accuracy-based retraining triggers, in which we use values of accuracy proxy to determine if online training should occur. There are two types of decisions that the retraining trigger will need to make: (a) If we do not retrain at the end of the last labeling window, should we retrain for this window? CARAVAN's retraining trigger will initialize retraining if it observes an abrupt drop in the value of accuracy proxy in the current labeling window compared to the last one, since that could be an explicit signal of potential concept or data drifts. (b) If we retrain at the end of the last labeling window, should we stop retraining for this window? As we demonstrate in the evaluation section (Figure 8), if we continuously retrain for several windows, the marginal inference accuracy gain would gradually decrease, assuming that there are no new drifts that show up in this period. As a result, the retraining trigger stops retraining if we have retrained for the last few windows and obtained decent inference accuracy gain.

> **Insight 4:** *The marginal inference accuracy gain of online training would quickly diminish if no new sources of drifts are present (i.e., the network is stable).*

3.3 How to Select CARAVAN's Elements?

For knowledge sources, we can choose existing systems (e.g., IDS) or construct new ones (e.g., fine-tuned foundation models). It is important to evaluate the labeling accuracy and approximate speed of a knowledge source using an offline labeled dataset before deploying it in Caravan. When selecting accuracy proxy and retraining trigger, we should consider the application's performance objectives (e.g., low false-positive rate) and identify signals or events from the system that might indicate performance degradation (e.g., increased rebuffering events in video streaming).

4 Implementation

We implement an end-to-end version of Caravan using Python. To interact with in-network ML models, Caravan stores the samples of the arriving flows in a streaming database, InfluxDB [13]. We initialize InfluxDB with a predefined schema consisting of various header/feature fields and metadata of the arriving packet (e.g., duration, data rate, and 5-tuple) as well as the inference results (prediction) from the deployed in-network ML model (for validation purposes). Upon the arrival of a labeling window's worth of samples, the labeling agent queries these data samples from InfluxDB to generate labels.

For knowledge sources (e.g., heuristics, DNN-based classifiers, and foundation models), we define how it labels data by completing its `label()` function (as described in §3.1). Heuristics come in the form of labeling functions [91] and are easily defined by the user. The DNN-based classifiers load a pre-trained DNN classifier, and run batched inference upon calls of `label()` for labeling. For foundation models, we use GPT-4 API [87] for sending labeling requests in the form of prompts. In this setup, labeling is modeled as a text completion task, and we explicitly prompt the language model to produce a label for each input data sample. We present a case study of implementing a foundation model, LLM-based knowledge source in §4.1. With individual knowledge sources defined, we build a labeling agent by specifying what knowledge sources it will be using. The labeling agent calls each knowledge source's `label()` function to obtain all labels and selects the best ones (with the most occurrences) as final labels.

For model validation and retraining, we define a model validator that runs `compute_accuracy_proxy()` to compute the accuracy proxy (in §3.2.1) with generated labels and inference results (from InfluxDB) as input arguments. The retraining trigger is defined as a function that checks if we have retrained for the last window. If not, we check if there is a significant drop in accuracy proxy value to initialize retraining; if yes, we then check if the increase in accuracy proxy value is small enough to stop retraining. If retraining is necessary, we go on to form a class-balanced dataset based on iCaRL [94], keeping the same number of data samples from each class and maintaining a fixed upper bound for the size of the dataset (which can be specified by the user). For training ML models, we use PyTorch [88] and one Nvidia V100 Tensor Core GPU from AWS.

Caravan maintains a busy-waiting process for data labeling, model validation, and online learning. This process will periodically read data from InfluxDB and initialize data labeling as well as model validation at the end of a labeling window (determined by time or number of data samples). If retraining is necessary, it will conduct retraining and send out the weights to the in-network ML model as gRPCs [12] or PCIe writes.

4.1 `label()` with Foundation Model (LLM)

We now present a case study of developing a new knowledge source using large language models (LLMs). We use commercial off-the-shelf LLM, more specifically ChatGPT [20]. ChatGPT is not explicitly fine-tuned on network traffic data; but, as a foundation model, may have been trained on openly available data from the Internet. Please refer to §A.1 for details on the specific model (and snapshot) we use for labeling.

• **Instruction Following.** To ensure the LLM understands the structure of input data and properly follows the subsequent instructions, we compose system prompts §A.2 that are shared by all incoming inference requests (including labeling and rule extraction). The system prompts precisely state the objective of the application (e.g., flag malicious traffic for intrusion detection) and enumerate the names and meanings of each feature in the network dataset.

> **In-context Learning Prompt (P1):** To begin with, here are some labeled flows for your reference later. The last field is the binary label (0 for benign and 1 for malicious): `[Flows, their features and labels go here]`. Next, I will give you some unlabeled flows for labeling. Please let me know if you understand the requirement by answering yes or no.

• **In-context Learning.** We take advantage of in-context learning [36, 90] to improve LLM's ability to label network data (or packets) with higher accuracy without (re)training or fine-tuning the original model. We provide a few labeled examples from the CIC-IDS2018 dataset [98]. The network traffic in these examples contains similar attack types (such as brute force attacks and DDoS attacks) to those present in the evaluation dataset (CIC-IDS2017). However, it is collected from a different network and at a different time. Using these labeled examples, we construct an in-context learning prompt (P1) shared by all subsequent inference requests.

> **Data Labeling Prompt (P2):** Please give me a label for each of these unlabeled flows. No explanation or analysis needed, label only; one flow on each line. Format for each line: (flow number) label. `[Flows and their features go here]`.

• **Data Labeling.** Whenever we invoke the `label()` function, we first compose a labeling prompt (P2). This prompt specifies the expected response format, facilitating easy parsing of responses for per-packet labels. Additionally, it includes all the data to be labeled, and structured in accordance with the system prompt. We concatenate the system prompt, the in-context learning prompt, and the labeling prompt, and submit an API request to the LLM.

> **Rule Extraction Prompt (P3):** To begin with, here are some example input flows for your reference later. `[Flows and their features go here]`. Based on these example input flows, can you do some analysis and help me come up with some rules or heuristics (in the form of a Python function) to determine if an unlabeled flow is benign or not? Make sure that in the Python function, you label a flow as malicious only when you are very confident. Name the function `label_flow_with_rule_cache()`, and pass it in a format that can be executed by `exec()`. The input of the function should be the 16 features in the system prompt (in order), and the output should be 0 (benign) or 1 (malicious).

• **Rule Extraction.** To extract rules to store in the labeling rule cache for fast and resource-efficient labeling, we construct a rule-extraction prompt (P3). This prompt explicitly requests the LLM to generate rules and heuristics for data labeling as a Python function, specifying the expected input/output formats to simplify the parsing of the generated responses. §A.3 shows an example function generated by the LLM for fast labeling.

5 Evaluation

In our evaluation, we show: (a) using three different choices of knowledge sources, Caravan is able to efficiently label new incoming network traffic for the purpose of model validation and retraining, and can achieve almost the same level of inference accuracy gains compared to using ground-truth labels (§5.2.1). (b) Caravan's accuracy proxy and retraining trigger allow us to efficiently determine when to initialize or stop retraining. As compared to continuous retraining, the use of accuracy proxy and retraining trigger has the potential to reduce GPU compute cost by an average of 74.55% without significantly hurting inference accuracy gain (§5.2.2). (c) In software simulation (§5.3.1), Caravan is able to achieve a 30.3% improvement in F1 score (on average) compared to static offline models across three chosen applications. Caravan's accuracy proxy and retraining trigger enable 61.3% saving in GPU compute time (on average) for retraining without significantly compromising inference accuracy gains. (d) In the end-to-end Taurus FPGA testbed (§5.3.2), Caravan continuously keeps in-network ML models up-to-date with changing traffic dynamics and maintains high inference accuracy at network line-rate. With accuracy proxy and retraining trigger, Caravan improves over static models in terms of F1 score by an average of 30% with 56.23% less CPU usage and similar memory footprint as continuous retraining baselines.

5.1 Experiment Setup

Use Cases. To evaluate Caravan, we select two network traffic analysis applications widely used and evaluated by prior work in the domain of in-network ML (Table 1). (a) *Network Intrusion Detection*: The goal is to flag network flows or network packets regarding whether they involve malicious activities. We expect the in-network ML model to offer a preliminary analysis of the network flows through binary classification before running more expensive downstream security analysis instead of providing complete end-to-end protection of a networked system. This application is an example of how in-network ML could improve the *security* of networked systems. (b) *IoT Traffic Classification*: The goal is to assign an IoT device type to a network flow or packet. Classification results from the in-network ML model enable operators to know what different flows might entail (e.g., application or data type) early in the network, and to act correspondingly based on different devices, applications, or data types to optimize for the quality of service (QoS) or user quality of experience (QoE). For example, network flows from video cameras might require allocation to a less congested network path, since the user will likely be in a live video conference. In this case, fewer packet retransmissions and lower latency are critical to good user perception of video and service quality. This application is an example of how in-network ML could improve the *performance* of networked systems.

Datasets and In-network ML Models. We closely follow prior work in the domain of in-network ML when choosing datasets and in-network ML models. A summary of these datasets and related statistics is available in Table 1.

For network intrusion detection, we follow prior work [101, 125] to use CIC-IDS2017 [98] and UNSW-NB15 [84]. With CIC-IDS2017, we use the same features from pForest [37] and a deep neural network with similar architecture to the one from Taurus [104]. With UNSW-NB15, we use the same features and one of the deep neural networks from the intrusion detection example of N3IC [101]. For IoT traffic classification, we follow prior work [101,125] to use UNSW-IoT [102]. For the in-network ML model, we follow the IoT traffic classification example of N3IC [101] in terms of feature selection and model architecture. For multi-class classification, we use one of N3IC's four-layer deep neural networks, which have 16, 64, 32, and 10 neurons on each layer, respectively; we replace the binary weights with 32-bit weights.

Choices and Configuration of Knowledge Sources. We choose three knowledge sources for the labeling agent to use. (a) A *DNN-based classifier* for intrusion detection on CIC-IDS2017: The DNN-based classifier has 8 layers and 13,222 parameters in total. The architecture is similar to a stacked autoencoder in DeepPacket [78]. It is trained on a small part of CIC-IDS2017 (a subset that is not used during testing) and a small part of CIC-IDS2018 [98] (a different intrusion detection dataset from the same publisher). (b) A *large language model* for intrusion detection on UNSW-NB15: The large language model is based on GPT-4 [87] text completion APIs. We program the user prompts properly so the language

Application	Dataset	# Samples	# Features	# Classes	# Drifts
Network Intrusion Detection	CIC-IDS2017 [98]	7,000	16	2	7
	UNSW-NB15 [84]	5,000	20	2	5
IoT Traffic Classification	UNSW-IoT [102]	108,000	16	10	9

Table 1: Network applications and datasets used in our evaluation with input features listed in §A.2 and [101]. A drift occurs in intrusion detection with the arrival of new attack traffic, and in IoT classification with unseen IoT device traffic.

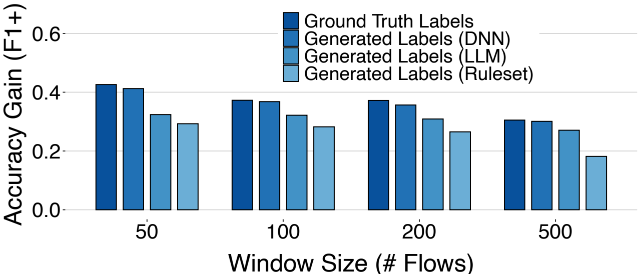


Figure 4: Caravan's labeling agent generates labels for online training, bringing comparable levels of accuracy gain as ground-truth labels across three different knowledge sources.

model can understand the particular format of our input network flows and generate labels in a format easily parsed by the labeling agent. To improve labeling accuracy, we take advantage of *in-context learning* and give the language model 10–20 labeled flows (not used during testing) for reference. (c) An *IoT device list* for IoT traffic classification on UNSW-IoT: We use the device list provided by the original dataset publishers. To ensure that the device list will generate strictly worse labels than the ground-truth labels, we modify the MAC address of some network flows so that the device list is unable label them. Overall, the device list can identify and label 10% of all the network flows in the dataset.

Quality and Usage Metrics. For accuracy, we use the F1 score [50] as the performance metrics for evaluating the quality of an in-network ML model. In machine learning, the F1 score is often preferred over basic metrics like classification accuracy. It provides a more nuanced measure of a model's performance, especially when class distributions are imbalanced or when the costs of false positives and false negatives differ. This preference for accuracy metrics aligns with previous research in the field [37, 101, 104, 122]. To better model the performance gain of the validation and online learning processes, we use the metrics of *accuracy gain*, defined as the increase in the F1 score of the retrained in-network ML model compared to that of the offline one. To determine the accuracy of a specific experiment, we first calculate an F1 score based on the model predictions and ground-truth labels at the end of each labeling window using the data from that window. Ultimately, we report the average F1 score, or the average increase in F1 score (i.e., F1+) compared to the offline model, as the final accuracy metric or accuracy gain.

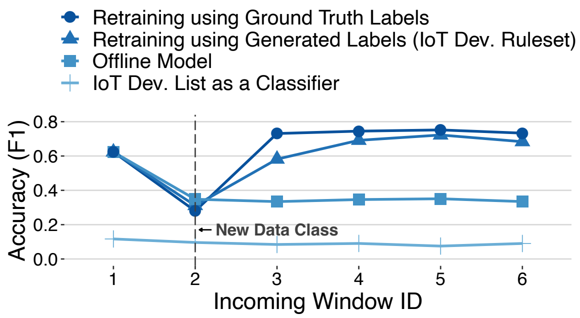


Figure 5: A comparison of generated labels using Caravan's labeling agent versus ground-truth labels for a low-accuracy and fast knowledge source (IoT device list) during data drift (i.e., encountering new data classes).

To quantify the system resource usage for online training, we use the metrics of *GPU compute time*, defined as the time spent on the GPU during online training. When using large language models as a knowledge source, we also use *tokens used for labeling* to demonstrate the cost of using an expensive knowledge source for labeling, defined to be the aggregate number of tokens (an addition of prompt tokens by the user and completion tokens by the language model) used for the labeling task.

End-to-End Testbed. We use the Taurus FPGA-based testbed [104] for end-to-end evaluation. A 32-port programmable Tofino Wedge100BF-32x switch [21] is used to sample packets for the control plane and manage the Taurus ML core, which is emulated as a bump-in-the-wire FPGA. The switch bypasses its internal traffic through the Xilinx Alveo U250 FPGA [3], which is used to emulate the in-network ML model. The control plane runs a process to perform model validation and retraining on the sampled packets and update the model weights in the FPGA via PCIe. It also runs the ONOS controller [18] and a Python REST API to install forwarding rules on the switch. Two 80-core Intel Xeon servers generate and receive traffic via ScaPy [19] or MoonGen [44]. The in-network ML model has been compiled to Verilog using the Spatial [70] compiler and installed on the FPGA for evaluations.

5.2 Microbenchmarks

5.2.1 Effectiveness of the Labeling Agent. We find that noisy labels and partial-coverage labels generated by imperfect knowledge sources can still lead to decent inference accuracy gains after online training.

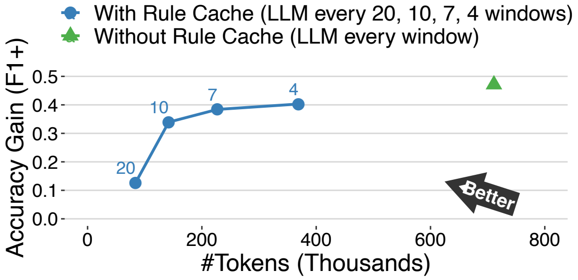


Figure 6: With a validation rule cache, CARAVAN conserves language model request tokens used for labeling, without significantly compromising the accuracy gains from retraining.

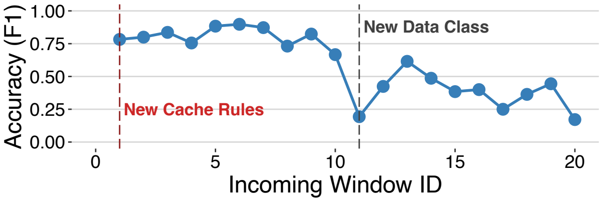


Figure 7: Though labeling rule cache generated by LLMs are subject to data drift, they generate accurate labels in a short local period of time.

Effectiveness of Noisy Labels. Noisy labels are defined to be labels that might be incorrect, and may be generated by knowledge sources like DNN-based classifiers or large language models in our case. Though these two knowledge sources (DNN-based classifier and language model) can generate a label for every sample of new incoming window when requested, we find that the overall quality of generated labels is around 0.7 to 0.8 in terms of F1 score on a small development set, indicating that there is a non-trivial level of noise in generated labels. We use these generated labels for a simple experiment of continuous online training, in which we skip validation and retrain at the end of every labeling window. We find that even with noisy labels, we are able to obtain a level of inference accuracy gain that is similar to the gain if we retrain with ground-truth labels under different labeling window sizes (Figure 4). The reason accuracy gain tends to decrease as window size increases is that we use a fixed number of 30 epochs for training; with larger training data sizes, it generally takes longer for the model to converge.

Effectiveness of Weak Supervision Labels. In the case of CARAVAN, weak supervision labels are defined to be labels that only cover a subset of all the samples in a labeling window and can be generated by low-accuracy but fast knowledge sources (e.g., an IoT device list) as discussed in §3.1.1. In our setup, the IoT device list can only label around 10% of all network flows in the dataset. To verify whether such a knowledge source can effectively mitigate model drift, we continuously retrain an ML model when new types of devices are present in the incoming data. We find that even with weak supervision labels that have partial coverage, we can achieve a comparable level of inference accuracy gain when data drift occurs (after the arrival of a new class) to that of retraining with ground-truth labels (Figure 5). At the same time, we find that the device list cannot be used independently to classify incoming data with high accuracy due to partial coverage, as depicted in Figure 5.

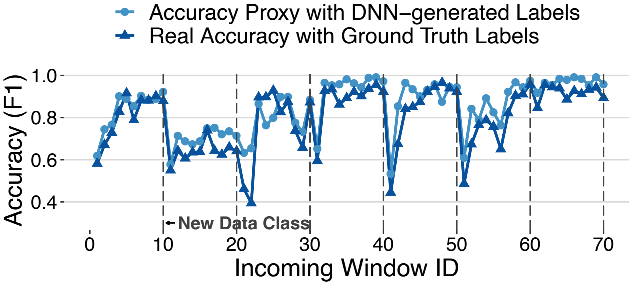


Figure 8: CARAVAN's accuracy proxy F1 scores align with the real F1 scores in terms of relative changes in accuracy on a temporal scale, particularly in instances of data drift.

5.2.2 Effectiveness of Labeling Rule Cache, Accuracy Proxy, and Retraining Trigger.

Labeling Rule Cache. As discussed in §3.1.1, when using expensive knowledge sources like large language models, we can request the knowledge source to generate temporary rules or heuristics in a rule cache that can be used for fast and cost-effective labeling for the following few labeling windows. In our experiment, we call language models for labeling and rule generation (in the form of a simple executable function) every 4, 7, 10, and 20 labeling windows. We use the generated function as the rule cache for labeling at the end of all other windows. By invoking the language model every 4, 7, or 10 windows, we achieve nearly the same level of inference accuracy gain after online training compared to employing language models for labeling at every window, while utilizing 65.4% fewer tokens on average (Figure 6). Note that the rules or heuristics in the rule cache can quickly go stale, especially in the case of a concept or data drift (Figure 7), so the rule cache should be updated frequently to avoid the generation of highly noisy labels.

Accuracy Proxy. We set up accuracy proxy in the same way as defined in §3.2.1, and verify if it is consistent with our insight that it can be used to reveal potential concept or data drifts even though it is not numerically equivalent to the real accuracy. In an incremental-class learning setup, where a new data class shows up in the incoming data every 10 labeling windows, we find that accuracy proxy is consistent with the real accuracy in terms of overall trend and relative changes in accuracy level on a temporal scale (Figure 8).

Retraining Trigger. To demonstrate the potential of using retraining triggers to avoid excessive retraining and save GPU compute time, we set up a window-based retraining trigger

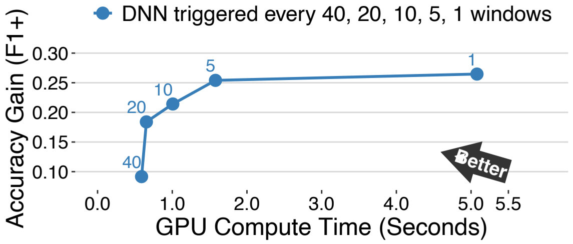


Figure 9: With a window-based retraining trigger, Caravan saves GPU compute time without significantly compromising retraining accuracy gain.

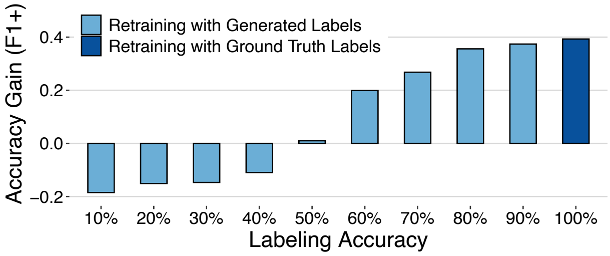


Figure 10: The relationship between Caravan's retraining accuracy gain and the labeling accuracy of the knowledge source. (Labeling accuracy is the percentage of data that can be correctly labeled by the knowledge source, compared to ground truth labels.)

that reduces the frequency of retraining from once every labeling window to once every 5, 10, 20, and 40 labeling windows. We observe that even with this straightforward retraining trigger, we manage to save an average of 74.55% GPU compute time, with at most a 0.05 reduction in inference accuracy gain in terms of F1 score when retraining occurs every 5 or 10 windows (Figure 9).

5.2.3 Sensitivity to External Knowledge Sources. Caravan assumes that users will be able to provide reliable knowledge sources that be adapted for data labeling. When inaccurate knowledge sources are used, the accuracy gain from Caravan's retraining may decrease and sometimes even drop below zero, as illustrated in Figure 10. We discuss potential solutions to this issue in §6.

5.3 End-to-End Improvement

We evaluate the end-to-end improvements of Caravan in software simulation and on the Taurus FPGA testbed [104].

5.3.1 Software Simulation. In software simulation, we find that Caravan is able to achieve a 30.3% improvement in F1 score (on average) as compared to static offline models across three chosen applications (Figure 12). We also find that the gap between inference accuracy gain of continuous online learning with ground-truth labels and with labeling-agent generated labels stays as little as 0.4–2.1% for intrusion detection with DNNs as knowledge source, and 0.5–1.8% for IoT traffic classification with device lists as knowledge source. Though that gap can be as large as 11% for intrusion detection with a large language model as a knowledge source, we believe that performance can be further improved when specialized network foundation models are used as knowledge sources in the future. Moreover, Caravan's accuracy proxy and retraining trigger enable 61.3% savings in GPU compute time (on average) for retraining without significantly compromising inference accuracy gain.

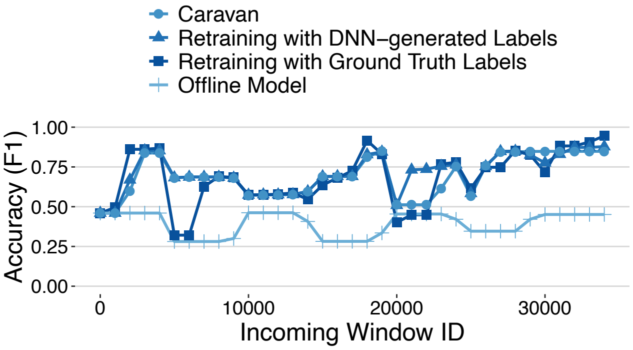


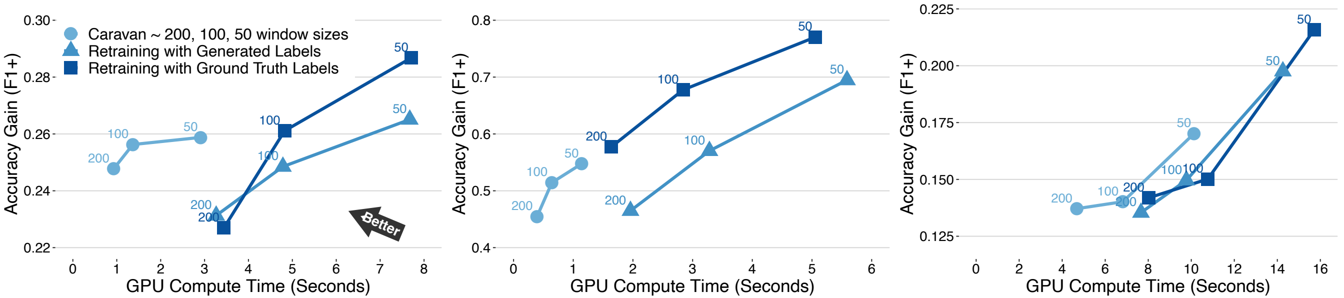
Figure 11: End-to-end results on the Taurus FPGA testbed. Caravan keeps in-network ML models up-to-date against changing traffic when operating at line rate.

System	LUT%	FFs%	BRAM%	Power (W)
Taurus: Offline	6.49	4.35	4.15	16.86
Caravan	6.81	4.71	4.15	17.16

Table 2: Resource usage of Caravan's in-network model for intrusion detection on the Taurus FPGA testbed.

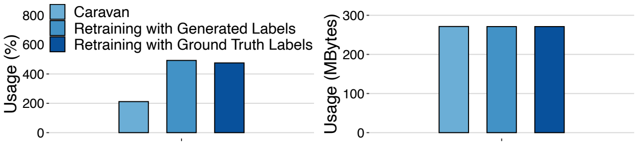
5.3.2 FPGA-based Experiments. In the Taurus FPGA testbed [104], we run an intrusion detection application with the same in-network model as in software simulation, programmed with Spatial [70]. We generate traffic by sampling 35 M packets from the CIC-IDS2017 dataset, while ensuring a uniform distribution of the seven attacks present in the dataset (i.e., 5 M packets for each attack). We preserve the order of the attacks as in the original dataset. Using Moongen [44], we send packets at 0.5 Million packets per second, and set the sampling rate to about 0.1%. Each labeling window receives about 500 packets. We find that Caravan can continuously keep in-network ML models up-to-date with changing traffic dynamics and maintain high inference accuracy under network line rate on a temporal scale (Figure 11). With accuracy proxy and retraining trigger, Caravan further improves upon static models in terms of F1 score by an average of 30%. It is worth noting that at times, the accuracy of Caravan can surpass that of the continuous retraining baselines. This is because continuous retraining for small in-network models may lead to overfitting, whereas Caravan's retraining trigger helps mitigate this issue.

5.3.3 Resource Usage & Latency Breakdown. Table 2 shows the FPGA's percentage resource count in terms of lookup tables (LUTs), flip flops (FFs), on-chip memory



(a) IDS with a DNN labeler on CIC–IDS2017 (b) IDS with an LLM labeler on UNSW–NB15 (c) IoT Classif. with Ruleset on UNSW–IoT

Figure 12: Tradeoff between inference accuracy gain and GPU compute time for CARAVAN and continuous retraining baselines across two applications (intrusion detection and IoT classification), three datasets, and three knowledge sources.



(a) CPU (b) Memory

Figure 13: CPU and memory usage of CARAVAN's busy-waiting process for labeling data, retraining model, and updating model weights.

Retraining Step	Latency (ms)
- Retrieving data from InfluxDB	6.041 ± 1.114
- Labeling data with DNN-based IDS	1.015 ± 1.238
- Computing accuracy proxy	1.732 ± 0.073
- Retraining in-network ML model	14.775 ± 0.982
- Installing new model weights	46.145 ± 0.507

Table 3: Latency breakdown of CARAVAN's retraining steps on a window of 100 packets for the network intrusion detection application.

(BRAM), and power (W). In contrast to the vanilla Taurus implementation (i.e., Offline ML) [104], which lacks support for online weight updates, CARAVAN introduces minimal additional overhead in FPGA resource usage while supporting live weight updates.

We also measure the CPU and memory usage of the CARAVAN's busy-waiting process that retrieves incoming data from a streaming database (i.e., InfluxDB), labels it, computes accuracy proxy, retrains models, and issues weight updates (§4). We see that CARAVAN reduces CPU usage by an average of 56.23% compared to continuous retraining baselines, without incurring any additional memory overhead (Figure 13). Table 3 further shows a breakdown of each these retraining steps in CARAVAN.

6 Limitations & Future Work

Optimizing Sample Selection for Online Learning. CARAVAN employs random sampling to reduce the volume of input data sent to the labeling agent. Existing research in online learning systems shows that network traffic is heavy-tailed and empirically variable [115], which could undermine the effectiveness of online learning in real-world deployments if training samples are not carefully selected [43]. While it is straightforward to modify the input data sampling and retraining data formation logic in CARAVAN, developing efficient and effective algorithms for online sample selection remains a future research direction that requires further understanding of both machine learning techniques and the characteristics of network traffic.

Reverting In-network ML Models. CARAVAN focuses on updating and improving models using continuous sampling of and selective retraining on the network's data, which lets them adapt to new events. However, in scenarios where data is compromised, it would be necessary to revert or reset these models to a previous good state. If online data (such as network traffic) is being used to retrain and update models, bad actors can poison training data by intentionally feeding bad traffic in the network. Future research may detect and protect against these attacks and restore models to a clean state.

Network Telemetry Data for ML. CARAVAN focuses on retraining models with sampled data but does not dictate how the collection of such data is performed. However, extensive research is needed on how to collect and sample data for the express purpose of retraining ML models. For instance, some data may not contribute to an increase in the fidelity of the model, even with further training iterations. In these cases, the data may simply be orthogonal to the task the ML model is built for. On the other hand, the system may need to sample more frequently in cases where notable network events are detected. For example, a server running out of resources may indicate a network attack that breaches security. Packets must be collected so as to classify and inoculate future ML models to these attacks. In short, collection systems for online training need to leverage dynamic sampling rates at various points throughout the network in order to ascertain when and where to get the best training data.

Creating Domain-Specific Knowledge Sources. In this paper, we repurpose GPT-4 as a knowledge source for network intrusion detection. We recognize that GPT-4 was not originally designed or trained for cybersecurity applications; instead, it is used primarily as a proof-of-concept foundation model for data labeling. An emerging research direction involves pre-training or fine-tuning domain-specific foundation models for networking or security on larger traffic traces (e.g., NetLLM [112], NetFound [54], Lens [110]). Another direction from the machine-learning community aims to enhance foundation models to better follow human intents and self-improve through feedback, whether human-generated or model-generated (e.g., constitutional AI [29] and self-improving LLMs [56, 61, 120]). These efforts could lead to developing knowledge sources that can generate accurate labels and better align with human expertise and intentions.

Evaluating and Validating Knowledge Sources. CARAVAN assumes that the provided labeling sources are sufficient to cover the space of input data for a given networking use case. As a next step, these labeling sources must be vetted further to ensure high-quality label generation. Common accuracy metrics such as F1, precision, or recall are all valid for assessing how well these labeling sources are performing (on a given dataset), but additional metrics are required to assess the full coverage of application space. For example, in a security context, how many of the commonly seen network attacks can the labeling source cover? Furthermore, the network community should start making its labeling sources public to allow retraining systems more effectively—similar to how various ML communities have put forth public collections of data and benchmarks. For instance, in the case of foundation models, public benchmarks feature open and comprehensive evaluations of models on specific applications, such as chat [124], code generation [74], and question answering [41]; these benchmarks help users select the best model for their particular use case. Finally, as suggested in Snorkel [92], multiple labeling sources can be aggregated for greater coverage and fidelity. In this way, aggregate labeling sources can generate more accurate labels than individual sources, effectively allowing a given source to cover the blind spots of another source.

Generalizing to Larger Control-Plane ML Models. Although CARAVAN is designed for online learning of in-network ML models, we believe that its core insights and techniques—such as using weak supervision for labeling data in an online setup, employing accuracy proxies, and utilizing retraining triggers to detect and mitigate model quality degradation—can be generalized to larger ML models deployed in the control plane. These control-plane ML models also face similar challenges like data or concept drifts [75] and a lack of labels for model monitoring and retraining in an online setup [53].

7 Related Work

Systems for Online Learning. Ekya [33] and RECL [67] discuss how online learning can be done for computer vision models on an edge server jointly with inference, while CARAVAN studies the case of in-network ML models in which data-plane inference does not interfere with control-plane online learning. Nazar [58] features how to mitigate data drift for ML models on mobile devices, and differs from CARAVAN as it does not address essential components of online learning (e.g., data labeling).

Data Collection and Generation for Networking. The emerging need to train ML models for networking tasks and design new network telemetry algorithms sparks extensive research in designing better tools for network data collection and network data generation. NetUnicorn [31] is a platform for collecting and actively labeling network data for developing offline generalizable ML models. It features a human-in-the-loop approach where users can select what data to collect and label, and it is different from our focus since CARAVAN features automatic online data labeling after ML models have been deployed. NetShare [118] enables synthetic IP-header generation for network flows but has a different focus from CARAVAN and does not study data labeling for downstream traffic analysis tasks.

Interpretability of ML Models. With the growth of ML models in networking, many existing efforts focus on the interpretation of these black-box models to make their decision-making logic transparent to network operators. For example, Trustee [62] proposes a framework that determines whether or not a given ML model suffers inductive biases by extracting a high-fidelity decision tree from the model being analyzed. However, such diagnosis of the ML models is not yet automatized and needs a human-in-the-loop. Indeed, CARAVAN can use Trustee as an orthogonal system component for diagnosing the behavior of the online learning model.

Programmatic Data Labeling. CARAVAN complements and augments (rather than competes with) existing data programming systems, such as Snorkel [92]. Snorkel uses generative models to estimate the accuracies of different knowledge sources, and can potentially be used for conflict resolution in CARAVAN's labeling agent. CARAVAN is similar to Snorkel in the aspect that both point out that weak knowledge sources can be used for labeling data and training ML models instead of using them for independent decision-making. However, CARAVAN focuses on how automatic data labeling helps online learning of ML models and mitigates drifts (by incorporating knowledge sources, accuracy proxy, and retraining trigger), while Snorkel focuses on enabling users to label datasets with multiple knowledge sources for training better ML models offline.

Weak Supervision in Networking. The concept of weak supervision has been extensively applied in networking, particularly in cybersecurity and internet measurement applications [47, 69, 86]. CARAVAN differs from these works by focusing on enabling weak supervision in an online setup to detect model quality degradation and retrain outdated models.

Label-free Data Drift Mitigation. Recent efforts in networking and security domains feature data drift mitigation with no need for labels. For example, CADE [117] proposes to train a neural network that can help determine if new incoming data has drifted away from training data. However, Caravan focuses on the continuous adaptation of an online model, where the training data are constantly evolving. Moreover, CADE uses root cause analysis to fix drifted models offline when there is no explicit requirement on how fast model update needs to happen, which is in contrast to Caravan's focus on the online setting when model updates must be done fast and automatically to keep up with the high inference rate. In summary, Caravan aims to be a more generalized framework designed for various in-network ML applications.

8 Conclusion

Once deployed online, in-network machine learning (ML) models can experience accuracy degradation owing to fluctuations in traffic patterns and changes in online data distribution. While online learning is a promising solution, it is challenging in practice due to the need for automatic labeling of evolving network traffic and the efficient monitoring of model performance degradation. To overcome these challenges, we present Caravan, the pioneering system for practical online learning of in-network ML models. Caravan addresses the issue of labeling new incoming traffic data for retraining by leveraging diverse knowledge sources that, otherwise, are unsuitable for real-time decision-making. Moreover, Caravan introduces the accuracy proxy metric to monitor model degradation and potential data drifts, providing an effective signal to trigger model retraining. Our evaluation shows that Caravan can keep in-network ML models up-to-date, achieving a 30.3% improvement in F1 score (on average) and reducing GPU compute time for training by 61.3% (on average), while achieving similar accuracy gains as continuous retraining. We hope the development of such a system will not only contribute to the domain of ML for networking and traffic analysis applications but also influence the design of practical and efficient machine-learning systems in general.

Acknowledgements

We thank our anonymous shepherd and reviewers for their invaluable feedback, which significantly enhanced the quality of this paper. We also thank Gerry Wan, Gautam Akiwate, Shinan Liu, Shiv Sundram, Tian Zhao, Alex Ratner, James Hong, Azalia Mirhoseini, Haijie Wu, Junchen Jiang, and Vyas Sekar for their insightful discussions. This research was supported by ACE, one of the seven centers in JUMP 2.0, an SRC program sponsored by DARPA; NSF awards CAREER-2338034, CNS-2211381, and CNS-2211384; and "SERICS" (PE00000014) under the NRRP MUR program funded by the EU-NGEU. Support also came in part from affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Intel, Microsoft, NEC, SAP, Teradata, and VMware.

References

[1] AI and ML: The New Frontier for Data Center Innovation and Optimization. https://www.techradar.com/news/data-centres-in-an-ai-and-ml-driven-future.

[2] Alveo SN1000 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html.

[3] Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[4] AMD Pensando. https://www.amd.com/en/accelerators/pensando.

[5] Apache Kafka. https://kafka.apache.org/.

[6] Bluefield Data Processing Units (DPUs). https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[7] CISCO NetFlow. https://www.cisco.com/c/en/us/tech/quality-of-service-qos/netflow/index.html.

[8] Creating a Predictive Network for the Human Mind. https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2022/m05/creating-a-predictive-network-for-the-human-mind.html.

[9] Data Centers in an AI and ML Driven Future. https://www.techradar.com/news/data-centres-in-an-ai-and-ml-driven-future.

[10] Every Request, Every Microsecond: Scalable Machine Learning at Cloudflare. https://blog.cloudflare.com/scalable-machine-learning-at-cloudflare/.

[11] GPT-4 Turbo in the OpenAI API. https://help.openai.com/en/articles/8555510-gpt-4-turbo-in-the-openai-api.

[12] gRPC. https://grpc.io/.

[13] InfluxDB. https://www.influxdata.com/.

[14] Infrastructure Processing Unit (Intel IPU) and SmartNICs. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[15] Intel Tofino 2. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html.

[16] Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date. https://ai.meta.com/blog/meta-llama-3/.

[17] Marvell OCTEON 10 DPU Platform. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdfe/.

[18] ONOS: Open Network Operating System. https://opennetworking.org/onos/.

[19] Scapy. https://scapy.net/.

[20] Three LLMs Walk into a Network Operations Center.... https://www.bigpanda.io/blog/three-large-language-models-walk-into-a-network-operations-center/.

[21] WEDGE 100BF-32X: 100GBE Data Center Switch. https://www.edge-core.com/cloud-data-center-100g/.

[22] What Is LLM Temperature? https://www.iguazio.com/glossary/llm-temperature/.

[23] Richelle Adams. Active Queue Management: A Survey. *IEEE Communications Surveys & Tutorials*, 15(3):1425–1476, 2012.

[24] Kazeem B Adedeji, Adnan M Abu-Mahfouz, and Anish M Kurien. DDoS Attack and Detection Methods in Internet-Enabled Networks: Concept, Research Perspectives, and Challenges. *Journal of Sensor and Actuator Networks*, 12(4):51, 2023.

[25] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A Survey of Network Anomaly Detection Techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.

[26] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*, 2014.

[27] Nahla Ben Amor, Salem Benferhat, and Zied Elouedi. Naive Bayes vs Decision Trees in Intrusion Detection Systems. In *ACM Symposium on Applied Computing*, 2004.

[28] Simran Arora, Avanika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Re. Ask Me Anything: A Simple Strategy for Prompting Language Models. In *ICML*, 2022.

[29] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from AI Feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[30] Diogo Barradas, Nuno Santos, Luıs Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-Based Network Security Applications. In *NDSS*, 2021.

[31] Roman Beltiukov, Wenbo Guo, Arpit Gupta, and Walter Willinger. In Search of netUnicorn: A Data-Collection Platform to Develop Generalizable ML Models for Network Security Problems. In *ACM CCS*, 2023.

[32] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC*, 2010.

[33] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *USENIX NSDI*, pages 119–135, 2022.

[34] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network Anomaly Detection: Methods, Systems and Tools. *IEEE Communications Surveys & Tutorials*, 16(1):303–336, 2013.

[35] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities. *Journal of Internet Services and Applications*, 2018.

[36] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.

[37] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. *arXiv preprint arXiv:1909.05680*, 2019.

[38] Brian Caswell, Jay Beale, and Andrew Baker. *Snort Intrusion Detection and Prevention Toolkit*. Syngress, 2007.

[39] Lingjiao Chen, Matei Zaharia, and James Zou. How Is ChatGPT's Behavior Changing over Time? *arXiv preprint arXiv:2307.09009*, 2023.

[40] David D Clark, Craig Partridge, J Christopher Ramming, and John T Wroclawski. A Knowledge Plane for the Internet. In *ACM SIGCOMM*, pages 3–10. ACM, 2003.

[41] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think You Have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[42] L. Dhanabal and S.P. Shantharajah. A Study on NSL-KDD Dataset for Intrusion Detection System Based on Classification Algorithms. *International Journal of Advanced Research in Computer and Communication Engineering*, 4(6):446–452, 2015.

[43] Alexander Dietmüller, Romain Jacob, and Laurent Vanbever. On Sample Selection for Continual Learning: a Video Streaming Case Study. *arXiv preprint arXiv:2405.10290*, 2024.

[44] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, 2015.

[45] Alice Este, Francesco Gringoli, and Luca Salgarelli. Support Vector Machines for TCP Traffic Classification. *Computer Networks*, 2009.

[46] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *ACM IMC*, 2020.

[47] Vojtech Franc, Michal Sofka, and Karel Bartos. Learning Detector of Malicious Network Traffic from Weak Labels. In *ECML PKDD*, pages 85–99. Springer, 2015.

[48] Nicholas Frosst and Geoffrey Hinton. Distilling a Neural Network into a Soft Decision Tree. *arXiv preprint arXiv:1711.09784*, 2017.

[49] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *USENIX NSDI*, 2019.

[50] Cyril Goutte and Eric Gaussier. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer, 2005.

[51] Vic Grout, John McGinn, and John Davies. Real-Time Optimisation of Access Control Lists for Efficient Internet Packet Filtering. *Journal of Heuristics*, 13:435–454, 2007.

[52] Andreas Grünbacher. POSIX Access Control Lists on Linux. In *USENIX ATC*, 2003.

[53] Jorge Luis Guerra, Carlos Catania, and Eduardo Veas. Datasets Are Not Enough: Challenges in Labeling Network Traffic. *Computers & Security*, 120:102810, 2022.

[54] Satyandra Guthula, Navya Battula, Roman Beltiukov, Wenbo Guo, and Arpit Gupta. netFound: Foundation Model for Network Security. *arXiv preprint arXiv:2310.17025*, 2023.

[55] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 2008.

[56] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language Models Can Teach Themselves to Program Better. *arXiv preprint arXiv:2207.14502*, 2023.

[57] James Halvorsen, Clemente Izurieta, Haipeng Cai, and Assefaw H Gebremedhin. Applying Generative Machine Learning to Intrusion Detection: A Systematic Mapping Study and Review. *ACM Computing Surveys*, 2024.

[58] Wei Hao, Zixi Wang, Lauren Hong, Lingxiao Li, Nader Karayanni, Chengzhi Mao, Junfeng Yang, and Asaf Cidon. Monitoring and Adapting ML Models on Mobile Devices. *arXiv preprint arXiv:2305.07772*, 2023.

[59] Bruno Miranda Henrique, Vinicius Amorim Sobreiro, and Herbert Kimura. Literature Review: Machine Learning Techniques Applied to Financial Market Prediction. *Expert Systems with Applications*, 124:226–251, 2019.

[60] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New Directions in Automated Traffic Analysis. In *ACM CCS*, 2021.

[61] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large Language Models Can Self-Improve. *arXiv preprint arXiv:2210.11610*, 2022.

[62] Arthur S Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A Ferreira, Arpit Gupta, and Lisandro Z Granville. AI/ML for Network Security: The Emperor Has No Clothes. In *ACM CCS*, 2022.

[63] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *ICML*, 2019.

[64] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[65] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Generative, High-Fidelity Network Traces. In *ACM HotNets*, 2023.

[66] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*, 2016.

[67] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *USENIX NSDI*, 2023.

[68] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-Band Network Telemetry via Programmable Dataplanes. In *ACM SIGCOMM*, 2015.

[69] Jared Knofczynski, Ramakrishnan Durairajan, and Walter Willinger. ARISE: A Multitask Weak Supervision Framework for Network Measurements. *IEEE Journal on Selected Areas in Communications*, 40(8):2456–2473, 2022.

[70] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A Language and Compiler for Application Accelerators. In *ACM PLDI*, 2018.

[71] Franck Le, Mudhakar Srivatsa, Raghu Ganti, and Vyas Sekar. Rethinking Data-Driven Networking with Foundation Models: Challenges and Opportunities. In *ACM HotNets*, 2022.

[72] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NeurIPS*, 2020.

[73] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[74] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct?

Rigorous Evaluation of Large Language Models for Code Generation. In *NeurIPS*, 2023.

[75] Shinan Liu, Francesco Bronzino, Paul Schmitt, Arjun Nitin Bhagoji, Nick Feamster, Hector Garcia Crespo, Timothy Coyle, and Brian Ward. LEAF: Navigating Concept Drift in Cellular Networks. *Proceedings of the ACM on Networking*, 1(CoNEXT2):1–24, 2023.

[76] Shinan Liu, Ted Shaowang, Gerry Wan, Jeewon Chae, Jonatas Marques, Sanjay Krishnan, and Nick Feamster. ServeFlow: A Fast-Slow Model Architecture for Network Traffic Analysis. *arXiv preprint arXiv:2402.03694*, 2024.

[77] Yingqiu Liu, Wei Li, and Yunchun Li. Network Traffic Classification Using K-Means Clustering. In *IMSCCS*, 2007.

[78] Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. Deep Packet: A Novel Approach for Encrypted Traffic Classification Using Deep Learning. *Soft Computing*, 24(3):1999–2012, 2020.

[79] Inbal Magar and Roy Schwartz. Data Contamination: From Memorization to Exploitation. *arXiv preprint arXiv:2203.08242*, 2022.

[80] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *ACM HotNets*, 2016.

[81] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*, 2017.

[82] Tahir Mehmood and Helmi B Md Rais. SVM for Network Anomaly Detection Using ACO Feature Subset. In *IEEE iSMSC*, 2015.

[83] Albert Mestres, Alberto Rodriguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J Hibbett, Giovani Estrada, Khaldun Ma'ruf, Florin Coras, Vina Ermagan, Hugo Latapie, Chris Cassar, John Evans, Fabio Maino, Jean Walrand, and Albert Cabellos. Knowledge-Defined Networking. *ACM SIGCOMM CCR*, 47(3):2–10, 2017.

[84] Nour Moustafa and Jill Slay. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.

[85] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online Model Distillation for Efficient Video Inference. In *ICCV*, 2019.

[86] Anirudh Muthukumar and Ramakrishnan Durairajan. Denoising Internet Delay Measurements Using Weak Supervision. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 479–484. IEEE, 2019.

[87] OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

[88] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

[89] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. Language Models as Knowledge Bases? *arXiv preprint arXiv:1909.01066*, 2019.

[90] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models Are Unsupervised Multitask Learners. *OpenAI Blog*, 1(8):9, 2019.

[91] Alex Ratner, Braden Hancock, Jared Dunnmon, Roger Goldman, and Christopher Ré. Snorkel Metal: Weak Supervision for Multi-Task Learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–4, 2018.

[92] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid Training Data Creation with Weak Supervision. *The VLDB Journal*, 29(2-3):709–730, 2020.

[93] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data Programming: Creating Large Training Sets, Quickly. In *NeurIPS*, 2016.

[94] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. iCaRL: Incremental Classifier and Representation Learning. In *CVPR*, 2017.

[95] Adam Roberts, Colin Raffel, and Noam Shazeer. How Much Knowledge Can You Pack into the Parameters of a Language Model? *arXiv preprint arXiv:2002.08910*, 2020.

[96] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*, 2015.

[97] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the Network Be the AI Accelerator? In *ACM NetCompute*, 2018.

[98] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. *International Conference on Information Systems Security and Privacy (ICISSP)*, 1:108–116, 2018.

[99] Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A Ghorbani. Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy. In *International Carnahan Conference on Security Technology (ICCST)*, pages 1–8. IEEE, 2019.

[100] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. Incremental Learning of Object Detectors without Catastrophic Forgetting. In *ICCV*, 2017.

[101] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-Architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX NSDI*, 2022.

[102] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[103] Linqi Song, Cem Tekin, and Mihaela Van Der Schaar. Online Learning in Large-Scale Contextual Recommender Systems. *IEEE Transactions on Services Computing*, 9(3):433–445, 2014.

[104] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for Per-Packet ML. In *ACM ASPLOS*, 2022.

[105] Tuan A. Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. Deep Learning Approach for Network Intrusion Detection in Software Defined Networking. In *IEEE WINCOM*, 2016.

[106] Gemini Team. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*, 2024.

[107] Saar Tochner, Giulia Fanti, and Vyas Sekar. Gen-T: Reduce Distributed Tracing Operational Costs Using Generative Models. In *Temporal Graph Learning Workshop@ NeurIPS 2023*, 2023.

[108] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. Retina: Analyzing 100GbE Traffic on Commodity Hardware. In *ACM SIGCOMM*, 2022.

[109] Minxiao Wang, Ning Yang, Nicolas J Forcade-Perkins, and Ning Weng. ProGen: Projection-Based Adversarial Attack Generation against Network Intrusion Detection. *IEEE Transactions on Information Forensics and Security*, 2024.

[110] Qineng Wang, Chen Qian, Xiaochang Li, Ziyu Yao, and Huajie Shao. Lens: A Foundation Model for Network Traffic in Cybersecurity. *arXiv preprint arXiv:2402.03646*, 2024.

[111] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *ACM SIGCOMM*, 2013.

[112] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. NetLLM: Adapting Large Language Models for Networking. *arXiv preprint arXiv:2402.02338*, 2024.

[113] Jun Xiao, Minjuan Wang, Bingqian Jiang, and Junli Li. A Personalized Recommendation System with Combinational Algorithm for Online Learning. *Journal of Ambient Intelligence and Humanized Computing*, 9:667–677, 2018.

[114] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *ACM HotNets*, 2019.

[115] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Alexander Levis, and Keith Winstein. Learning in situ: A Randomized Experiment in Video Streaming. In *USENIX NSDI*, 2020.

[116] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Alexander Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-Control Research. In *USENIX ATC*, 2018.

[117] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In *USENIX Security*, 2021.

[118] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical GAN-Based Synthetic IP Header Trace Generation Using NetShare. In *ACM SIGCOMM*, 2022.

[119] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM*, 2020.

[120] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation. *arXiv preprint arXiv:2310.02304*, 2024.

[121] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *ACM IMC*, 2017.

[122] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Transactions on Networking*, 2024.

[123] Gan Zheng, Ioannis Krikidis, Christos Masouros, Stelios Timotheou, Dimitris-Alexandros Toumpakaris, and Zhiguo Ding. Rethinking the Role of Interference in Wireless Networks. *IEEE Communications Magazine*, 52(11):152–158, 2014.

[124] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*, 2024.

[125] Zhizhen Zhong, Mingran Yang, Jay Lang, Christian Williams, Liam Kronman, Alexander Sludds, Homa Esfahanizadeh, Dirk Englund, and Manya Ghobadi. Lightning: A Reconfigurable Photonic-Electronic SmartNIC for Fast and Energy-Efficient Inference. In *ACM SIGCOMM*, 2023.

A Details on LLM-based Knowledge Source

A.1 Model Choice & Reproducibility

We use the `gpt-4-1106-preview` model snapshot from the OpenAI API service as the LLM—the latest model available at the time of implementation and evaluation of Caravan. We anticipate that future model snapshots released by OpenAI (such as `gpt-4-turbo` and `gpt-4o`) or Google (such as `Gemini Ultra` and `Gemini Flash`) could be adapted for data labeling using similar prompts, as long as the model supports a sufficiently large context window. The same would hold true for emerging open-source LLMs, such as those from Meta (e.g., Llama 3 series [16]) and Mistral AI (e.g., Mixtral 7B [64]).

The behavior of commercial LLM APIs may evolve over time, even using the same model snapshot and prompts [39]. To ensure reproducibility, one strategy would involve leveraging open-source LLMs instead of third-party APIs. However, these LLMs necessitate high-end GPUs or aggressive compression before deployment; we do not use these in our paper. Another approach is to decrease the temperature [22] during the generation process to minimize variability across different runs when utilizing third-party APIs.

A.2 System Prompts

a. System Prompt (UNSW-NB15): You are an expert in network security. The user is now labeling a network intrusion detection dataset, and he/she wants to assign a binary label (0 for benign or 1 for malicious) to each traffic flow in the dataset based on each flow's input features. He/She will give you a few labeled flows for reference, and you will then help him/her label another few unlabeled flows. Feel free to use your own expertise and any information the user gives you. These are the features of the input flows and meanings of the features: dur (record total duration), proto (transaction protocol, which will be categorized), sbytes (source to destination transaction bytes), dbytes (destination to source transaction bytes), sttl (source to destination time to live value), dttl (destination to source time to live value), sload (source bits per second), dload (destination bits per second), spkts (source to destination packet count), dpkts (destination to source packet count), smean (mean of the packet size transmitted by the src), dmean (mean of the packet size transmitted by the dst), sinpkt (source interpacket arrival time (mSec)), dinpkt (destination interpacket arrival time (mSec)), tcprtt (TCP connection setup round-trip time), synack (TCP connection setup time, the time between the SYN and the SYN_ACK packets), ackdat (TCP connection setup time, the time between the SYN_ACK and the ACK packets), ct_src_ltm (no. of connections of the same source address in 100 connections according to the last time), ct_dst_ltm (no. of connections of the same destination address in 100 connections according to the last time), ct_dst_src_ltm (no. of connections of the same source and the destination address in 100 connections according to the last time).

b. System Prompt (CIC-IDS2017): You are an expert in network security. The user is now labeling a network intrusion detection dataset, and he/she wants to assign a binary label (0 for benign or 1 for malicious) to each traffic flow in the dataset based on each flow's input features. He/She will give you a few labeled flows for reference, and you will then help him/her label another few unlabeled flows. Feel free to use your own expertise and any information the user gives you. These are the features of the input flows and meanings of the features: flow IAT min (minimum packet inter-arrival time in microseconds), flow IAT max (maximum packet inter-arrival time in microseconds), flow IAT mean(average packet inter-arrival time in microseconds), packet length min (minimum packet length), packet length max (maximum packet length), packet length mean (average packet length), total packet length (total packet length), number of packets (total number of packets in the flow), SYN flag count (number of TCP SYN flags), ACK flag count(number of TCP ACK flags), PSH flag count (number of TCP PSH flags), FIN flag count (number of TCP FIN flags), RST flag count (number of TCP RST flags), ECE flag count (number of TCP ECE flags), flow duration (duration of flow in microseconds), and DST port (destination port).

A.3 A Rule Extraction Output by LLM

In Figure 14, we show an example heuristics-based labeling function generated by the LLM, based on 500 unlabeled flows from the UNSW-NB15 dataset. The model generates all comments along with the function.

A.4 Concerns Regarding Data Contamination

In the domain of machine learning, data contamination [79] occurs when pre-trained language models, trained on extensive web-based datasets, inadvertently incorporate downstream test sets into their pretraining corpus. This phenomenon can lead to biases in evaluation metrics and diminish the generalization capacity of large language models. In Caravan, the issue of data contamination is also pertinent, as we adhere to previous methodologies by utilizing public datasets for evaluation.

While it is not feasible to entirely eradicate the issue of data contamination through retraining a new model from scratch, we implement the following measures to mitigate its potential impact on our evaluation: (a) We ensure that the LLM never generates *trivial* labels, meaning labels that precisely match the ground truth labels. (b) We carefully examine the labeling rule functions produced by the LLM to ensure they do not generate labels through *memorization* based on features, such as IP address.

B Artifact Appendix

B.1 Abstract

The artifact includes the software simulation and the FPGA/P4 testbed we built for implementing and evaluating Caravan. We also include a detailed guide with step-by-step

```python
def label_flow_with_rule_cache(flow):
    """
        Labels a network flow as benign (0) or malicious (1) based on certain heuristics.
        This function is designed to be conservative and only labels a flow as malicious when it is highly confident.

        :param flow: A tuple containing features of the flow.
        :return: An integer label, 0 for benign and 1 for malicious.
    """

    # Unpack the flow features
    dur, proto, sbytes, dbytes, sttl, dttl, sload, dload, spkts, dpkts, smean, dmean, sinpkt, dinpkt, tcprtt, synack,
    ackdat, ct_src_ltm, ct_dst_ltm, ct_dst_src_ltm = flow

    # Define thresholds and conditions for a flow to be considered malicious
    # These thresholds are derived from cybersecurity expertise and unlabeled data provided
    conditions = [
        # If the source to destination time to live value is maximum (255) and the destination to source packet count
        # is 0, it is likely to be a scan or a DoS attack.
        sttl == 254 and dpkts == 0,

        # If the number of connections to the same source address in the last 100 connections is high (above 30),
        # it might indicate a scanning activity or a distributed attack.
        ct_src_ltm > 30,

        # If the number of connections to the same destination address in the last 100 connections is high (above 30),
        # it might indicate a scanning activity or a distributed attack.
        ct_dst_ltm > 30,

        # If the number of connections to the same source and destination address in the last 100 connections is high
        # (above 30), it might indicate a scanning activity or a distributed attack.
        ct_dst_src_ltm > 30,

        # If the TCP connection setup round-trip time, the time between the SYN and the SYN_ACK packets,
        # and the time between the SYN_ACK and the ACK packets are all 0, it might indicate a SYN flood attack.
        tcprtt == 0.0 and synack == 0.0 and ackdat == 0.0,
    ]

    # If any of the malicious conditions are met, label the flow as malicious
    if any(conditions):
        return 1

    # Otherwise, label the flow as benign
    return 0
```

Figure 14: A heuristics-based labeling function generated by the LLM.

instructions for automatically running the key experiments and plotting the figures presented in the paper.

B.2 Scope

The `simulation/` folder contains the source code to automatically run key experiments from the paper and reproduce the corresponding figures (i.e., Figures 4–10, 12). The `testbed/` folder contains the new code changes and the instructions to set up and run the FPGA/P4-based evaluations for CARAVAN.

B.3 Contents

The artifact is provided as a self-contained repository available at https://github.com/Per-Packet-AI/Caravan-Artifact-OSDI24.

- **`simulation/`** contains the software code for reproducing evaluated figures, with automation scripts for generating data and producing figures located at `simulation/scripts/experiments.sh` and `simulation/scripts/plots.sh`, respectively.
- **`testbed/`** contains a modified version of the Taurus FPGA testbed [104] for testing CARAVAN's use cases.

B.4 Hosting

CARAVAN is hosted on GitHub: https://github.com/Per-Packet-AI/Caravan-Artifact-OSDI24.

B.5 Requirements

Hardware. CARAVAN requires at least an 8-core server with 16 GiB of RAM, one CUDA 12.1-compatible GPU (e.g., Nvidia V100), along with Internet connectivity to access OpenAI API endpoints. We recommend using a Google Compute Engine (g2-standard-8) instance.

Software. CARAVAN runs with Python version 3.10 or later with CUDA support. The complete list of dependencies is available in `simulation/pyproject.toml` and gets installed automatically using `pip install -e .` from the `simulation/` directory.

nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training

Zhiqi Lin[†*], Youshan Miao[‡], Quanlu Zhang[‡], Fan Yang[‡], Yi Zhu[‡], Cheng Li[†], Saeed Maleki[◊*],
Xu Cao[‡], Ning Shang[‡], Yilei Yang[‡], Weijiang Xu[‡], Mao Yang[‡], Lintao Zhang[△*], Lidong Zhou[‡]
[†]*University of Science and Technology of China,*
[‡]*Microsoft Research,* [◊]*xAI,* [△]*BaseBit Technologies*

Abstract

With the growing model size of deep neural networks (DNN), deep learning training is increasingly relying on handcrafted search spaces to find efficient parallelization execution plans. However, our study shows that existing search spaces exclude plans that significantly impact the training performance of well-known DNN models (e.g., AlphaFold2) under important settings, such as when handling large embedding tables in large language models.

To address this problem, we propose nnScaler, a framework that generates efficient parallelization plans for deep learning training. Instead of relying on the existing search space, nnScaler advocates a more general approach that empowers domain experts to construct their own search space through three primitives, `op-trans`, `op-assign`, and `op-order`, which capture model transformation and the temporal-spatial scheduling of the transformed model of any parallelization plans. To avoid space explosion, nnScaler allows the application of *constraints* to those primitives during space construction. With the proposed primitives and constraints, nnScaler can compose existing search spaces as well as new ones. Experiments show that nnScaler can find new parallelization plans in new search spaces that achieve up to 3.5× speedup compared to solutions such as DeepSpeed, Megatron-LM, and Alpa for popular DNN models like Swin-Transformer and AlphaFold2.

1 Introduction

Deep neural networks (DNN) have shown remarkable success [2, 27, 35]. However, training a large DNN model today requires resources far exceeding the capacity of a single computing device, such as a GPU. Therefore, a common practice has been to partition a large model, schedule the partitioned model to a large number of GPUs, and then construct a well-coordinated execution plan across the GPUs (i.e., a parallelization plan) for deep learning training [24, 26, 39].

It is challenging to find an efficient parallelization plan for DNN model training. A DNN model is often represented as a data flow graph (DFG) that can consist of thousands of nodes [56], with each node representing a DNN operator, e.g., matrix multiplication. A parallelization plan requires deciding on a partitioning choice for each operator, which can have many different partition choices [26]. Additionally, each partitioning choice for all operators in the DFG further requires the selection of a spatial-temporal scheduling scheme from many scheduling options designed for thousands of GPUs. This creates a vast *search space* with prohibitive combinatorial complexity for identifying an effective parallelization plan that dictates model partitioning and scheduling.

Due to the immense search space, model training often depends on carefully designed parallelization plans. For example, Megatron-LM [50] incorporates the well-known, parameterized parallelization plans known as data/tensor/pipeline parallelism to support GPT-like models (§2). This approach essentially constructs a few well-studied classes of parallelization plans within the large search space. More recently, Alpa [65] organized parallelization plan choices into a two-level hierarchical space, where the system first searches a parallelization plan on pipeline (inter-operator) parallelism and then on tensor (intra-operator) parallelism within each pipeline stage. This approach offers a larger search space and so often results in better parallelization plans. However, existing search spaces exclude several configurations in the parallelization plans (§2, §4.2). Our study shows that this limitation significantly impacts training performance on well-known models such as Swin-Transformer [35] and AlphaFold2 [27] (§8).

While existing work studies specific parallelization plans or searches within a carefully-crafted search space, we argue that domain experts should be empowered with the capability to *compose their own search space*. Given the wide variety of model architectures and the expansive domain knowledge of an expert, this approach could expose more parallelization opportunities. To this end, we propose three primitives, `op-trans`, `op-assign`, and `op-order`, that enable the com-

[*] This work was done when the authors were with Microsoft Research.

position of search space with arbitrary model partitioning (op-trans), as well as spatial (op-assign) and/or temporal scheduling (op-order) of the partitioned model. We show that existing parallelization plans or search spaces can be elegantly expressed by the three primitives with *constraints* on model partitioning and spatial-temporal scheduling. More importantly, with the new constraints, searching within the space composed by the three primitives can lead to new parallelization plans that significantly outperform those found in existing search spaces or specific parallelization plans. Essentially, the three primitives, along with the necessary constraints, represent a more general abstraction to characterize parallelization plans.

Based on the above insight, we built nnScaler, a framework that facilitates the search, generation, and optimization of parallelization plans for deep learning training. Domain experts first use nnScaler to construct the desired search space for parallelization plans through the three primitives (§3). Specifically, given a model, op-trans designates how each operator can be partitioned; op-assign denotes the placement of each partitioned operator on GPUs; and op-order specifies the preferred temporal order across multiple operators when they are assigned to the same GPU.

nnScaler also allows the application of *constraints* to the primitives (§4). An example of a constraint applied to op-trans is one that only evenly splits an operator into 2, 4, 8, and 16 partitions. The use of constraints, especially those leveraging the characteristics of DNN models (e.g., the large embedding table in §4.2), greatly reduces the search space. As a result, with proper search policies applied to such a constrained search space (§5), nnScaler can discover unconventional parallelization plans that significantly outperform existing ones.

Given the sophisticated model partitioning and spatial-temporal scheduling enabled by nnScaler, a parallelization plan may deviate significantly from the original dataflow graph representing the DNN model. To ensure the correctness of a generated plan, nnScaler introduces vTensor-pTensor (§6), a tensor abstraction that tracks the "lineage" across operators before and after partitioning. This allows nnScaler to maintain correct data dependency during graph partitioning and detect cycles in the graph that could lead to deadlocks, thereby excluding invalid plans. Moreover, vTensor-pTensor also enables automatic communication adaptation when an operator is split and assigned across multiple devices. Finally, nnScaler lowers the discovered parallelization plan into executable code, enabling parallel deep learning training on each device.

Implemented based on PyTorch [43], nnScaler demonstrates great power and flexibility, facilitating the discovery of new parallelization plans (§4.2, §8) that achieve up to 3.5× speedup over existing parallel training systems, such as DeepSpeed [47], Megatron-LM [39], and Alpa [65], for popular deep learning models in computer vision (Swin-Transformer [34]), language translation (T5 [45]), and biological analysis (AlphaFold2 [27]). nnScaler has been used to develop, train, and finetune next generation deep learning models across Microsoft. The code is available in [5].

2 Background and Motivation

Search space for parallelization plans. A parallelization plan refers to a training execution plan that specifies the model partitioning and corresponding spatial-temporal scheduling scheme on a given set of GPUs. Training a large model with hundreds of billions of parameters requires thousands of GPUs [9]. A large model may consist of approximately 100 layers, each representing a sub-neural architecture (e.g., attention [58]) with tens of operators handling tensors with tens of thousands of dimension size (e.g., the hidden dimension). The vast partitioning choices (for a large model) and the enormous spatial-temporal scheduling choices (on a large number of GPUs) combine to create a prohibitively large, combinatorial search space for parallelization plans.

Existing approaches rely on well-studied, handcrafted parallelization plans or search space to address this problem. For example, *data parallelism*, a special parallelization plan, partitions an operator along the batch dimension of its associated tensors. These partitioned operators are then replicated across multiple devices (GPUs) and shared with the same model parameters (weights) to enable concurrent model training.

Tensor parallelism is a class of more general plans that permit the partitioning on dimensions not limited to the batch dimension [26, 50, 59]. This approach allows the partitioned operators to be distributed across different devices, accommodating models too large to fit into a single device.

As a large DNN model typically consists of multiple layers, it is also possible to partition a model into multiple stages, with each stage containing one or several layers. The stages are placed on different devices and executed in a pipeline, hence the name *pipeline parallelism*. To improve pipeline efficiency, a batch of training samples is further divided into micro-batches, and are then executed following a carefully designed temporal order [18, 24, 30].

The aforementioned parallelism schemes can be combined into a new scheme, known as *3D parallelism*, to further improve training efficiency. Megatron-LM [39] incorporates 3D parallelism, which integrates data, tensor, and pipeline parallelism in a parameterized manner to support GPT-like models. Given N devices, Megatron-LM partitions a model into K stages, with each stage divided into M partitions. The model is executed using K-stage pipeline parallelism and M-way tensor parallelism. For a sufficiently large N, Megatron-LM can also employ $(\frac{N}{M*K})$-way data parallelism to achieve further improvement in training performance. 3D parallelism represents a few well-studied classes of parallelization plans within the large search space.

Alpa [65] further generalizes these parallelism schemes to handcraft a two-level hierarchical search space. This hierarchy enables the use of efficient searching techniques like dynamic programming. Alpa has been shown to produce superior parallelization plans due to its larger search space, i.e., a combination of SPMD [61] (a generalized tensor-parallelism space) and pipeline parallelism.

Limitations of existing search space. Although existing handcrafted parallelization search space is shown effective for mainstream models with similar model architectures, it relies on assumptions that simplify the search and construction of parallelization plans. These simplifications, however, may exclude promising plans from considerations (§4.2).

In tensor parallelism, it is assumed that partitioned operators and their corresponding split tensors are distributed across *disjoint* devices. For example, to train a vision model with high fidelity images (e.g., [34]), tensor parallelism splits the large tensors associated with the large image and distributes the divided tensors among disjoint devices. This excludes cases where the split operators are placed on fewer devices, meaning multiple operators share one device and compute in a streamlined manner to reduce memory consumption and inter-device communication costs simultaneously [11] (detailed in §4.2 and §8).

Pipeline parallelism assumes that the training involves one forward pass and one backward pass. However, models like AlphaFold2 [17, 27] require three forward passes coupled with a single backward pass. This unconventional training approach renders existing pipeline parallelism [24, 38, 39] inapplicable.

Pipeline parallelism also assumes that different pipeline stages are spread across *disjoint* devices and prohibit any two stages from sharing the same set of devices through temporal multiplexing. For example, multi-lingual LLMs [45, 62, 64] often employ a large embedding table in the early computational stage in the model. This results in significant GPU memory consumption (>40%) but small computation utilization (<5%). Given the disjoint device assignments in pipeline parallelism (and also in tensor parallelism), the imbalance in hardware utilization is unavoidable.

The later handcrafted search spaces (e.g., [61, 65]) that combine tensor and pipeline parallelism (and others), inherit these assumptions and, therefore, suffer from the same limitations. This motivated us to design a more *flexible* method for space construction that can enable domain experts to find more effective training plans for their models (§3, §4, §5).

New challenges due to flexibility. Introducing a more flexible way to construct parallelization plan space brings new challenges. While existing frameworks like Megatron-LM [50], Alpa [65], and DeepSpeed [47] only implement a few well-studied partitioning, scheduling, and communications schemes that support parallelization plans in well-understood parallelization spaces, the new space could uncover new ways of operator partitioning, new operator scheduling with unconventional communication patterns. Furthermore, more flexible parallelization plans are less studied and hence could be error-prone. To address the above challenges, we designed a compiling process to detect and prevent potential errors in parallelization plans (e.g., cycles in a transformed DFG), and to generate the runtime code with efficient communication operations for the discovered parallelization plan (§6).

Primitives	**Usage**
`op-trans(op,algo,n)`	`algo` $\in$ `op.algos()` `n` $\in \mathbb{N}$, natural numbers
`op-assign(op,d)`	`d` $\in \mathbf{D}$, a set of devices
`op-order(op1,op2)`	`op1` executes before `op2`

Table 1: Primitives for parallelization space construction.

3 Parallelization Search Space Construction

A parallelization plan can be naturally expressed by the *model partitioning* and the *spatial-temporal* scheduling of the partitioned model. Correspondingly, nnScaler proposes three primitives, `op-trans`, `op-assign`, and `op-order` (summarized in Table 1), to capture the three aspects of a parallelization plan. Combined, the primitives can be used to compose any search space for a parallelization plan given arbitrary models and accelerator devices.

`op-trans`. `op-trans(op,algo,n)` transforms an operator `op` into `n` sub-operators according to a transformation algorithm `algo`, selected from the algorithm set corresponding to the type of `op`. For example, `matmul`$(A_{i,k}, B_{k,j})$, the matrix multiplication operator, can be partitioned into two `matmul` operators along dimension i of tensor A while replicating tensor B. In fact, most operators can be partitioned along a certain dimension (e.g., i or k in A or B) of the associated tensors and the computation of partitioned (sub) operators would remain the same as that of the original operators. Based on this observation, nnScaler implements the partitioning algorithms for the major operators in most DNN models. Domain experts can then reuse the desired algorithm via the `algos()` interface. nnScaler can also integrate custom transformation algorithms, such as those developed by domain experts, for any given operator. Note that the transformation algorithm can be more than just operator partitioning. For instance, an operator can be augmented by an additional recomputing operator or a memory-swapping operator to save memory [11, 23, 28, 41, 53]. In this paper, we use the term "transformation" and "partitioning" interchangeably.

`op-assign`. Given a set of devices $\mathbf{D}$ and an operator `op`, `op-assign(op,d)` denotes that `op` will be executed on the `d`-th device in $\mathbf{D}$.

op-order. When non-dependent operators, e.g., op1 and op2, are assigned to the same device, op-order(op1,op2) ensures that op1 must execute before op2. Execution order for non-dependent operators can play a vital role in training performance. For example, in pipeline parallelism, an operator in a pipeline stage can be partitioned into multiple micro-batches along the batch dimension. We denote these (sub)operators as op.mb$_1$, op.mb$_2$, etc., where mb$_i$ designates the corresponding microbatch ID. The operators op.mb$_i$ can be executed in an arbitrary order with regard to op.mb$_j$ ($i \neq j$). Nonetheless, various research shows that once these operators are being orchestrated carefully in temporal dimension, it is possible to minimize the pipeline "bubble" [24, 54] to significantly improve training efficiency.

With the three primitives mentioned above, domain experts can write Python codes to compose arbitrary search spaces for parallelization plans given any DNN model. These codes are not necessarily tied to specific DNN models. Consequently, nnScaler separates the model codes from the codes related to search space and search policy. Note that to ease programming efforts, op in the primitives can represent a sub-graph, where the primitive applies to each of the operators in the sub-graph.

Due to the flexibility of the primitives and the scale of large DNN models, the constructed parallelization search space often contains hundreds and thousands of operators with combinatorial search complexity. To address this issue, nnScaler allows domain experts to impose *constraints* when applying those primitives. These constraints can significantly reduce the search space (§4), thereby enabling effective search methods (§5).

4 Applying Constraints in the Search Space

In nnScaler, constraints are expressed as parameterized arguments to the primitives in Table 1. When all arguments become specific values, the whole space is reduced to a concrete parallelization plan. Below, we illustrate how well-studied parallelization plans like data, tensor, and pipeline parallelism can be expressed by using the three primitives and constraints (§4.1). Several new constraints that lead to novel parallelization plans are discussed in §4.2.

4.1 Constraints for Existing Search Spaces

Constraints for data and tensor parallelism. Table 2 shows the primitives and the associated constraints for data and tensor parallelism. Both data parallelism and tensor parallelism partition an operator evenly into n partitions. The partition is performed along a certain dimension, depicted by algo, where each partitioned sub-operator is assigned to a distinct device for concurrent execution, i.e., constraints ② and ③ in Table 2. Note that data parallelism always partitions along the batch dimension, hence the selection of algo is more restricted compared to tensor parallelism.

Primitives	Constraints
① sub-ops = op-trans(op,algo,n)	$n = \lvert \mathbf{D} \rvert$
② op-assign(sub-op$_i$,d$_i$) ③ op-assign(sub-op$_j$,d$_j$)	$d_i, d_j \in \mathbf{D}$, $d_i \neq d_j$

Table 2: Constraints for data and tensor parallelisms.

Constraints for pipeline parallelism. Given a device set $\mathbf{D}$, pipeline parallelism divides a model G in to sub-graphs G_i ($0 \leq i < \lvert \mathbf{D} \rvert$), where i denotes the i-th pipeline stage. And those sub-graphs will be assigned in disjoint devices, shown in Table 3.

To minimize the bubble, pipeline parallelism divides a batch of samples into micro-batches. A sub-graph, denoted as (G$_i$,n), operates on the the n-th micro-batch. We further denote a forward pass subgraph as fG$_i$ and a backward pass subgraph as bG$_i$, the constraints to schedule the well-known 1F1B [24] pipeline parallelism can be summarized in Table 4.

Primitives	Constraints
① op-assign(G$_i$, d$_i$) ② op-assign(G$_j$, d$_j$)	$d_i, d_j \in \mathbf{D}$, $d_i \neq d_j$

Table 3: Constraints for dividing a model G into $\lvert \mathbf{D} \rvert$ stages.

Primitives	Constraints
① op-order((fG$_i$,m),(fG$_i$,n)) ② op-order((bG$_i$,m),(bG$_i$,n))	$m < n$
③ op-order((fG$_i$,m+$ofst$),(bG$_i$,m)) ④ op-order((bG$_i$,m),(fG$_i$,m+$ofst$+1))	$ofst = \lvert \mathbf{D} \rvert - i$, $m \geq 0$

Table 4: Constraints for 1F1B schedule.

As illustrated in Figure 1, Constraints ① and ② in Table 4 ensure that: in stage i, the execution order of micro-batches must be the same for both forward and backward passes. That is, given any two micro-batches m and n, where $m < n$, fG_m should be executed before fG_n (①). The same applies to bG_m and bG_n in the backward pass (②).

Constraints ③ and ④ in Table 4 specify the subtle scheduling order of 1F1B. They define $ofst$, the offset with respect to the current stage. The earlier the stage in the pipeline, the larger the offset. Therefore, given G_i, the backward pass of the earlier microbatch should be executed later w.r.t. the forward pass (③). And the forward pass of the later micro-batch should be executed in adjacency to the backward pass of the earlier micro-batch (④).

The hierarchical combination of tensor parallelism and pipeline parallelism forms the space of Alpa [65], where tensor parallelism is nested within each stage of pipeline parallelism. This can be constructed by replacing d_i in the pipeline constraints in Table 3 with a set of devices D_i for each stage.

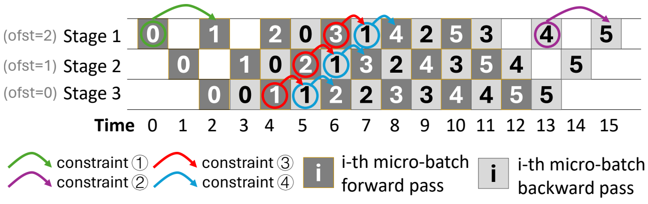


Figure 1: The constraints from Table 4 for 1F1B pipeline.

The stage is then applied with the constraints of tensor parallelism within D_i. For ease of exposition, the construction of this handcrafted parallelization space for certain sub-operators of a model, along with the constraints, is collectively defined in a general interface named `staged_spmd(ops, devices)`, to be used later.

4.2 New Constraints

In addition to existing search spaces, domain experts can apply new constraints to construct customized search spaces to search for new, more performant parallelization plans for various models, as we will elaborate next.

Constraints for Swin-Transformer. To enhance capability in vision tasks, there has been a growing trend to adopt higher resolution images to train large vision models such as Swin Transformer [34]. The use of larger images results in larger intermediate tensors during training, especially in the attention (Attn) and feedforward (FF) operators (for transformer-based models). It requires larger memory that a single GPU cannot accommodate.

Tensor parallelism is the standard practice used to address this issue. Given a pipeline, operators in Attn and FF are split and assigned to $|M_i|$ devices, where M_i denotes the set of devices accommodating operators in the i-th stage. Operators split by tensor parallelism are placed disjointedly, and so each device holds only one split operator. However, we observe that sometimes multiple split operators can share a single device and compute in a streamlined manner, resulting in fewer devices required for each pipeline stage and less memory consumption. Although the streamlined computing of multiple split operators may slow down the computing process, the reduced communications across fewer devices can lower cost and speed up the overall process.

Given any operator `op` from Attn and FF in stage i, let `sub_op` to be any transformed sub-operator of `op`. Suppose we allow C of such `sub_ops` to share one device, leading to a set of devices D_i assigned to stage i operators, where $|D_i| < |M_i|$. The constraints are as specified in Table 5. The rest operators can be described by the existing search space, namely the one defined in [65]. Note that C is a hyper-parameter where the value can be searched by the policy in §5.

Operators	Primitives	Constraints

`op` $\in$
 $\{\text{Attn} \cup \text{FF}\}$
	`sub_ops =`
	 `op-trans(op,algo,n)`
		$n = C \cdot \mid \mathbf{D}_i \mid$

	`op-assign(`$\texttt{sub\_op}^{j}_{i}$`,`$\texttt{d}_\texttt{i}$`)`	$0 \leq j < \lvert C \rvert$ $d_i \in \mathbf{D}_i$

Table 5: Constraints for Swin-Transformer.

Constraints for T5. Multi-lingual models such as T5 [45] often employ a large embedding table, say **E**, which contains vocabulary embeddings from multiple languages [64]. The table **E**, required only in the first and last layers of an LLM, incurs significant memory consumption but requires little computation cost. Pipeline parallelism would prioritize the device assignment to accommodate **E**, leaving the remaining devices for the other operators. This arrangement results in imbalanced hardware utilization, with devices containing **E** exhibiting low GPU cycle usage but high memory usage.

Thanks to nnScaler's three primitives and constraints, we can split **E** across the entire device set **D**. All other operators across all pipeline stages can then share the remaining resource left in **D** by constructing a search space following the conventional search space. These constraints, highlighted in Table 6, breaks the conventional assumption that operators in different pipeline stages cannot share the same set of devices. Similar solutions are also applicable to the training of graph neural networks [19].

Operators	Primitives	Constraints

`op` $\in \mathbf{E}$	`sub_ops =`
	 `op-trans(op,algo,n)`
	 `op-assign(`$\texttt{sub\_op}_\texttt{i}$`,`$\texttt{d}_\texttt{i}$`)`
		$n = \lvert\mathbf{D}\rvert$
		 $d_i \in \mathbf{D}$
		 $0 \leq i < \lvert D \rvert$

`ops` $\notin \mathbf{E}$	`staged_spmd(ops,` $\mathbf{D}$`)`	

Table 6: Constraints for T5.

Constraints for AlphaFold2. In AlphaFold2 [27], training each micro-batch requires three forward passes and one backward pass, i.e., 3F1B. Traditional 1F1B pipeline parallelism cannot support this type of pattern. As shown on the left side of Figure 2, a naive approach of training one micro-batch after another is inefficient due to pipeline bubbles and the accumulation of many unnecessary intermediate results. Therefore, we decided to interleave the forward and backward passes across different micro-batches while maintaining constraints on temporal orders. Let f_pG_i denote the forward sub-graph fG_i at the i-th pipeline stage in the p-th forward pass, and let *ofst* be $S - i$, where S denotes the total number of pipeline stages. Table 7 highlights the constraints for 3F1B.

Constraints ① and ② in Table 7 interleave the three forward passes of consecutive micro-batches in decreasing order. Constraint ③ specifies that the smallest micro-batch in the last executed forward pass should be executed before the corresponding backward pass sub-graph on a micro-batch ID with an offset (*ofst*) relative to the current stage, where *ofst* is defined similarly to that in Figure 1 of §4.1.

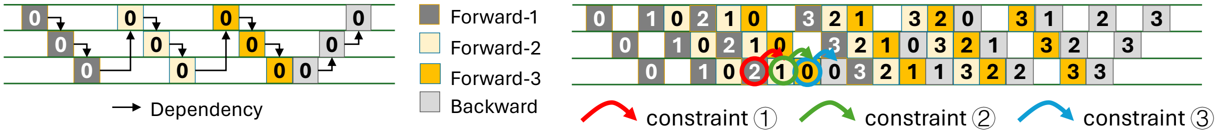


Figure 2: 3F1B schedule for AlphaFold2.

Primitives	Constraints
① op-order((f$_1$G$_i$,m+2),(f$_2$G$_i$,m+1))	$m \geq 0$
② op-order((f$_2$G$_i$,m+1),(f$_3$G$_i$,m))	$m > 0$
③ op-order((f$_3$G$_i$,m),(bG$_i$,m-ofst))	$m > ofst$

Table 7: Constraints for AlphaFold2.

In addition to Table 7, the search space for 3F1B also reuses the primitives and constraints in Tables 2 and 3. As shown on the right side of Figure 2, these constraints together form a space comprising unconventional parallelization plans that exhibit improved training performance (§8).

4.3 Discussion

Constraints are a powerful abstraction for customizing various parallelization plans and defining the search space for the plans. To design effective constraints, nnScaler assumes its users, usually domain experts, are knowledgeable on model architecture and parallel training. With such knowledge, it becomes intuitive to construct a search space using the three primitives. Based on our own experiences, effective constraints can be derived by identifying performance bottlenecks in the training, e.g., excessive GPU memory usage, computation/communication imbalance. The constraints can then be defined to alleviate the bottlenecks. And constraints can be refined iteratively along with the changing bottlenecks after the adjustment in constraints [33]. Through the refinement of constraints, nnScaler makes the generation of parallelization plans significantly easier than previous approaches.

5 Plan Search Policy

With the new user-defined search space, nnScaler incorporates a general policy framework to search for an efficient parallelization plan. As illustrated in Algorithm 1, the policy takes model graph G and a user-specified search space as inputs. We denote a space as C_{trans}, C_{assign}, C_{order}, corresponding to the three primitives op-trans, op-assign and op-order, along with augments associated with the constraints. The policy gradually shrinks the space with increasingly stringent constraints, ultimately reducing the space to a unique parallelization plan, denoted as C^{final}_{trans}, C^{final}_{assign}, C^{final}_{order}. A key feature of this policy framework is that it allows developers to "carve out" a sub-space from the new search space, where existing search polices are applicable. Specifically, the search process consists of two phases: operator partitioning and placement search, and temporal ordering search.

Algorithm 1: The policy framework of plan search.

```
Input: $G$, Model graph; $C_{trans}$, $C_{assign}$, $C_{order}$, the space
       defined by the primitives with constraints.
Output: $C^{final}_{trans}$, $C^{final}_{assign}$, $C^{final}_{order}$, that determine a concrete
        parallelization plan.
  /* Operator partitioning & placement search   */
  /* Subgraph search with existing search algo */
1 $G^{sub}$, $C^{sub}_{trans}$, $C^{sub}_{assign}$ ← GetSubSpace($G$, $C_{trans}$, $C_{assign}$);
2 $C^{new}_{trans}$, $C^{new}_{assign}$ ← Alpa($G^{sub}$, $C^{sub}_{trans}$, $C^{sub}_{assign}$);
  /* Search in the rest option space            */
3 $C_{trans}$, $C_{assign}$ ← ShrinkSpace($C_{trans}$, $C^{new}_{trans}$, $C_{assign}$, $C^{new}_{assign}$);
4 $C^{final}_{trans}$ $C^{final}_{assign}$ ← ILP($G$, $C_{trans}$, $C_{assign}$, objective=$eq.1$);
  /* Temporal ordering search                   */
5 $C^{final}_{order}$ ← Tessel($G$, $C^{final}_{trans}$, $C^{final}_{assign}$, $C_{order}$);
6 return $C^{final}_{trans}$, $C^{final}_{assign}$, $C^{final}_{order}$; /* a concrete plan */
```

Operator Partitioning and Placement Search. The goal of this phase is to evenly distribute computations across devices while minimizing communication costs. Various partitioning options for an operator yield different communication costs. For instance, partitioning the batch dimension involves an allreduce on parameters, while partitioning parameters leads to replicating input activation tensors across devices. Different placement options for operators also result in varying execution times for each device. Therefore, the execution time on a device d is the sum of its assigned operators' computation time $Comp_d$ and the associated communication time $Comm_d$. Overall runtime is dictated by the slowest device [54, 65], which is formulated as:

$$\text{minimize} \quad \max_{d \in \mathbb{D}} \{Comp_d + Comm_d\}. \tag{1}$$

By representing partitioning and placement options as integers, this optimization problem can be viewed as an integer linear programming problem, which is NP-hard.

With the application of constraints, the space in Equation 1 can be greatly reduced, thus enabling a faster search process (§8). nnScaler searches within a gradually reduced space by leveraging multiple policies. It firstly inspects the constructed search space and extracts a subspace (e.g., staged_spmd) that can leverage existing search policies like Alpa [65] through GetSubSpace (line 1 in Algorithm 1), an interface that reduces the search space through the input constraints. The extracted subspace may only consist of a subset of operators, leaving the rest operators undetermined. Once the transformation and placement decisions are made for the operator subset, the search space can be further reduced. Then, nnScaler fetches the reduced search space through `ShrinkSpace` (line 3 in Algorithm 1) and proceeds to use other policies (e.g., ILP solvers) within it, until finding the transformation and placement decisions for every operator.

For example, Table 5 reduces the operator assignment space to C operators per device; and Table 7 mainly specifies temporal order. The remaining subspace of these two cases can be organized like the space defined by `staged_spmd`. Additionally, Table 6 evenly pre-allocates the embedding table **E** to all devices evenly, with the remaining space corresponding exactly to `staged_spmd`. Consequently, the framework can apply the search policy in Alpa [65] to these sub-spaces to find a specific partitioning and assignment scheme for the involved operators (possibly a subset), i.e., C_{trans}^{new} and C_{assign}^{new} (line 2). These two new constraints, combined with their original versions, produce a smaller search space (line 3), where the framework can apply an ILP solver to find the final partitioning and assignment solution for the entire model, denoted as C_{trans}^{final} and C_{assign}^{final} (line 4). Note that as a general framework, users can replace the policies in Algorithm 1 by other search policies such as FlexFlow [26] or Tofu [59].

Temporal Ordering Search. After operator transformation and assignment, the temporal order of some operators is already specified by the data dependency in the transformed graph. However, it is possible for two operators on the same device to have no direct dependency, which means they can be executed in arbitrary orders. Moreover, for pipeline parallelism, the order of the same operator computed on different micro-batches within one batch is unspecified. nnScaler leverages Tessel [32], a state-of-the-art search policy, to determine the execution orders for these operators. Tessel groups operators within a micro-batch on each device into sub-graphs and formulates their execution order as an ILP problem. The optimization goal is to minimize the end-to-end execution latency of a mini-batch. Each sub-graph is assigned to an integer time slot, and the search, powered by Z3 Solver, enumerates possible order options without violating data dependencies. User-specified constraints on `op-order`, acting as Z3 constraints, play a crucial role in effectively reducing the search cost (line 5 in Algorithm 1).

Note that nnScaler does not claim contributions on individual search policies discussed in this section. It is the abstraction of primitives and constraints that makes the efficient search of parallelization plans possible.

6 Parallelization Plan Compilation

nnScaler compiles a model and the generated parallelization plan into executable codes, following the end-to-end process illustrated in Figure 3. The system first converts a deep learning model into a data flow graph, known as Graph IR. With the search space defined by the primitives and the associated constraints, nnScaler leverages a search policy to generate a parallelization plan. The plan compilation then applies the primitives and the constraints defined in the plan to the Graph IR. Data dependency tracking is performed during this step with the vTensor-pTensor abstraction. The resulting Graph IR, describing the new data dependency and the additional communication operations incurred due to operator distribution across devices, will be further materialized into parallel executable code.

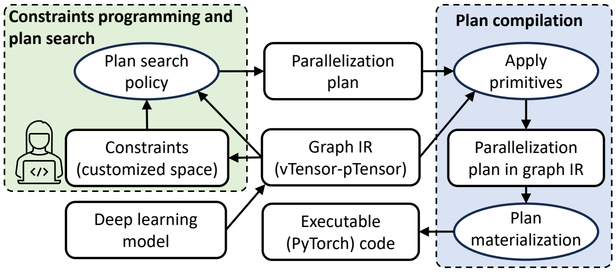

Constraints programming and plan search
Plan search policy
Constraints (customized space)
Parallelization plan
Graph IR (vTensor-pTensor)
Deep learning model
Executable (PyTorch) code
Plan compilation
Apply primitives
Parallelization plan in graph IR
Plan materialization

Figure 3: System overview of nnScaler.

Tensor Abstraction vTensor and pTensor are introduced to track changing data dependencies during the application of the three primitives. As depicted in Figure 4, a pTensor represents a tensor in the original logical model; vTensors are the resulting tensors after applying the three primitives to the pTensor. A vTensor links to a pTensor and maintains a mask indicating the accessed portion of the pTensor that this vTensor represents. A pTensor can be associated with multiple operators. At the top of Figure 4, the output of operator `A` serves as the input for operator `B`. Both operators are linked to the same pTensor through their respective vTensors.

With vTensor, each operator can be transformed, assigned, and ordered independently. When applying an `op-trans`, nnScaler partitions vTensors through the "mask", leaving pTensors unchanged. For instance, in Figure 4, operator `A` only splits itself and its output vTensor, while the vTensor of operator `B` remains unaffected. For other type of primitives, vTensor's mask remains unchanged. Therefore, given a producer vTensor (e.g., in `A`) and a consumer vTensor (e.g., in `B`) that are linked to the same pTensor, nnScaler can detect whether they have data dependency by intersecting their masks. With a dataflow graph, each operator in the graph consumes and produces vTensors according to underlying pTensors, thus facilitating the fine-grained data dependency tracking. During runtime execution, only vTensors will be instantiated to real GPU tensor instances.

With the data dependency tracking enabled by vTensor-pTensor abstraction during data flow graph transformation, nnScaler can detect cycles in the new graph that leads to

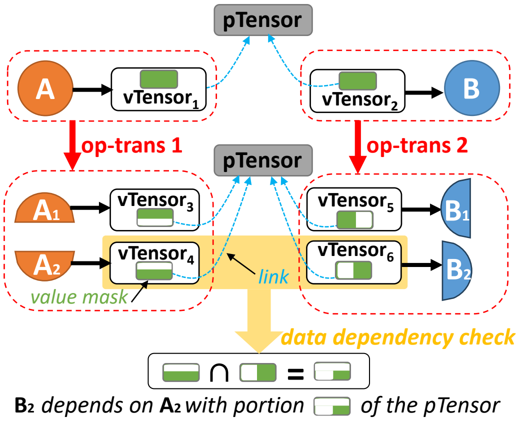


Figure 4: Tracks data dependencies with pTensor-vTensor.

deadlock, thus excluding invalid parallelization plans.

Data Dependency Materialization After applying primitives and constraints, nnScaler materializes the new data dependencies described by vTensor-pTensor into concrete data operations and communications. For a consumer vTensor (e.g., B_1 in Figure 4), nnScaler identifies dependent producer vTensors (e.g., A_1 and A_2) and inserts tensor manipulation operations, such as `torch.split` or `torch.chunk`, to extract the corresponding tensor fragments. When producers and consumers reside on different devices (due to `op-assign`), peer-to-peer send-recv communication operators [44] will be inserted during materialization.

To improve communication efficiency, certain communication patterns across vTensors within the same pTensor can be implemented using collective communication primitives, such as allgather, allreduce, or alltoall [44]. For instance, in Figure 4, communications between vTensors 3, 4 of A and vTensors 5, 6 of B can be materialized using the more efficient alltoall primitive. nnScaler employs simple pattern matching to identify appropriate collective primitives for each pTensor and its associated vTensors.

7 Implementation and Experiences

We implemented nnScaler based on PyTorch [43] with 24K lines of Python code. nnScaler takes a PyTorch model developed for a single device, and converts it into an intermediate graph representation (IR). After the transformation, the spatial-temporal scheduling, and the insertion of communications and tensor manipulation operations specified in a parallelization plan, each device will receive a sub-graph represented by the IR. nnScaler then converts the sub-graph back to a PyTorch code file. And PyTorch runs the code files (i.e., using `torchrun`) in parallel for distributed training.

To support a wide range of PyTorch models, nnScaler implements an augmented graph converter based on TorchFX [55], comprising 2,243 lines of Python code. This converter combines TorchFX's symbolic execution with value tracing of `torch.jit.trace` to handle control flow, which is a typical barrier when converting PyTorch models to TorchFX. By default, PyTorch models usually contain only the forward pass. nnScaler automatically completes the backward pass using autograd functionality with the chain rule [40]. So far, nnScaler has successfully converted 26319 out of 31301 (84.1%) PyTorch models from HuggingFace [25] Natural Language Processing tasks. The conversion failures are mainly due to unsupported operators, e.g., the custom operators designed for specific models. We are actively exploring way to support more operators, along with their corresponding transformation algorithms.

7.1 Experiences

nnScaler has been used by multiple projects across different teams in Microsoft to support the pretraining and fine-tuning of next generation DNN models on several generations of NVIDIA and AMD GPUs. This includes RetNet [51], YOCO [52], LongRoPE [16], Phi-3 series [7][1], and a large science foundation model consisted of a transform-based model combined with a graph neural network. The model size ranges from 3 billion to 92 billion parameters.

The decision to use nnScaler is based on two key factors. First, incorporating new models into existing distributed training frameworks presents intricate engineering challenges. This involves tasks such as the parallelization of the new modules, identifying suitable partition options, and ensuring the end-to-end training correctness, which includes tasks like data loading, gradient normalization (*gnorm*) [12], and optimizer. This process typically takes two experienced engineers about two months to complete. Compounding to the problem, existing parallelization plans often do not works well on new models, resulting in unsatisfactory Model FLOPs Utilization (MFU). Second, the research on new models often requires changes to model architectures, configurations, and training settings. This, in turn, may necessitate further adjustments to be made to the parallelization plan for efficient training, a daunting task for machine learning researchers. nnScaler precisely targets these pain points. Since nnScaler separates codes for the logical model from codes for the parallelization plan, it enables a separation of concerns: model developers can focus on model architecture innovations while system developers can study better parallelization plans. Moreover, our collaboration with these teams have yielded a number of insights, which will be discussed next.

Debugging nnScaler. nnScaler offers great flexibility in model training, but the new primitives and constraints also contribute to increased system complexity, rendering certain parallelization plans error-prone. nnScaler enables a modular approach to debugging system problems, where a new,

[1] nnScaler is used in some post training steps for the long context version of Phi-3, not for the model pretraining.

less-studied sub-graph generated by a new constraint can be replaced with a well-tested constraint. For instance, nnScaler can selectively apply data parallelism, which is less likely to have bugs, to a portion of the model while maintaining the existing parallelization plan for the rest of the model unchanged. This adjustment does not require model code modification; it simply configures the pre-build parallelization plan. By iteratively changing the suspected modules in the plan, it facilitates the identification of the problematic module.

Model accuracy. Achieving high model accuracy is the ultimate goal of model training. However, oftentimes, even a small bug in the training framework or model code can result in a degraded accuracy. Further complicating matters is that while the situation may appear normal in the early training stage, the loss curve tends to deteriorate over an extended training period (e.g., thousands of steps for a 7B LLM) and may eventually diverge. Directly comparing the loss and gradient values with well-tested training plans like data parallelism is impractical. For example, as the reduce operations (e.g., `matmul` or `allreduce`) in more complicated plans introduce drifts in floating-point values due to different orders of summation [20], which is an expected behavior. This makes it difficult to discern if it is an expected numeric deviation or a semantic bug. With respect to this issue, nnScaler firstly evaluates the correctness of a large-scale parallelization plan for a model by reducing the model's hidden dimension to fit the training in a single device. This makes debugging the correctness a much easier task. The model change is easily achievable by slight changes in the model code, thanks to the clean separation between the model code and training code. Subsequently we applied the searched parallelization plan to the reduced model and then assessed the overlap of the loss and gnorm curves with their counterparts in the well-tested data parallelism training. We observed that the gnorm curve is a good indicator, amplifying divergence at earlier stages and signaling potential bugs in the system.

In-place operators. To improve training performance, in-place operators like `Tensor.add_` update tensors in-place. However, partitioning in-place operators could become problematic. For example, if the partitioning of the in-place operator leads to the cloning of a tensor that originally implements in-place updates, the resulting non-inplace sub-operator would not preserve the original effect of the in-place operator's effect. This is due to a violation of the Static Single Assignment (SSA) form [14] when mixing in-place and non-inplace operators. To avoid this problem, nnScaler follows SSA during graph transformation, then replaces some of the non-inplace operators with their original in-place versions in the later optimization phase.

8 Evaluation

The evaluation of nnScaler covers the expressiveness of parallelization primitives and the search efficiency of parallelization plan with constraints. More importantly, we evaluated the performance of the newly searched parallelization plans on real-word models to demonstrate the effectiveness of the entire system in achieving efficient parallelization of new models and settings. In summary, the evaluation results show that:

- The parallelization primitives in nnScaler can construct various parallelization plans, including both existing handcrafted ones (§8.1) and newly innovated ones, as introduced in this paper(§8.2).
- End-to-end evaluation of the three novel parallelization plans on SwinTransformer, T5, and AlphaFold2 shows up to 3.5×, 2.5×, 1.4× speedup, respectively, compared to the baselines of Megatron-LM [39], Alpa [65], DeepSpeed [47], and DAP [13]. (§8.3)
- Parallelization space with constraints helps nnScaler quickly discover efficient plans, resulting in an 11.7× search speedup compared to the searches without constraints.

8.1 Expressiveness of Plan Construction

We evaluated the expressiveness of the three primitives for plan construction by implementing popular handcrafted parallelization plans listed in Table 8. These plans can be decomposed into operator transformation, placement and ordering, which is well aligned with the three primitives in Table 1. 14 out of 17 parallelization plans can be successfully supported by nnScaler. The parallelization plans under SPMD are implemented through `op-trans`. Data and flexible tensor parallelism can be easily supported. Transformer Parallelism and DAP are handcrafted tensor parallelisms for Transformer and AlphaFold2, respectively. Sequence Parallelism and ZeRO stage-3 are special tensor parallelisms, that decouple the partitioning of the operator and its input tensor to optimize memory usage. nnScaler supports them by inserting an `identity` operator between the input tensor and its operator through `op-trans`, facilitating easy decoupling.

The parallelization plans under MPMD are different types of handcrafted pipeline parallelism. They can be supported using `op-order` without implementing a new execution engine. Notably, nnScaler does not support PipeDream due to its asynchronous training method, as nnScaler respects the original training semantics of a model. For TeraPipe, nnScaler currently lacks access to concrete values in tensors, preventing it from determining data dependency at the token level (*i.e.,* tensor masks), a requirement for TeraPipe. In the future, nnScaler can implement TeraPipe through instrumentation tools for deep learning models like [21].

Beyond parallelisms, nnScaler also accommodates memory optimization techniques (e.g., recompute, swap) and the overlapping of computations and communications. Its support of recompute relies on a customized `algo` of `op-trans`

Categories	Mechanisms	Support
SPMD Parallelism	Data Parallelism [1]	✓
	Flexible Tensor Parallel [26, 59, 61]	✓
	Transformer Parallelism [50]	✓
	DAP [13]	✓
	Sequence Parallelism [29]	✓
	ZeRO [47]	✓
MPMD Parallelism	1F1B [18, 50]	✓
	GPipe [24]	✓
	Chimera [30]	✓
	PipeDream (Async) [38]	×
	TeraPipe [31]	×
Memory Optimizations	Gradient Accumulation [60]	✓
	Recompute [11]	✓
	Chain-recompute [28]	✓
	Swap [23]	✓
Overlapping	ByteScheduler [42]	×
	All-reduce Overlap [49]	✓*

Table 8: Supported parallelization plans. '*' requires additional co-scheduling of computation and communication at runtime.

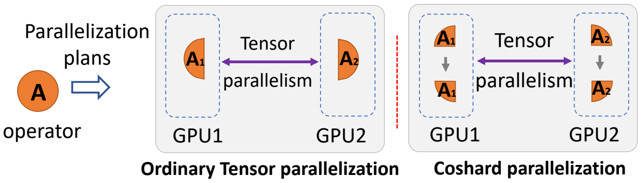


Figure 5: Coshard plan found by nnScaler.

to transform an operator to its recompute version, similar to `torch.utils.checkpoint` in PyTorch [6], while deferring the materialization to the compilation phase. nnScaler does not support ByteScheduler [42], which overlaps two consecutive mini-batches. This is because the boundary of transformation and scheduling in nnScaler is a single mini-batch, though it could potentially be extended to multiple mini-batches.

8.2 Plan Search Results

With the new constraints described in §4.2, nnScaler searches within each constructed space and discovers three novel parallelization plans that show superior training performance.

Coshard. Figure 5 illustrates Coshard, which is used for models with large tensors like SwinTransformer. It can co-exist with tensor parallelism to reduce peak memory of activation tensors. For example, A_1 is partitioned into two, placed on the same device, and executed sequentially. After applying recompute of A_1, the peak activation size of A_1 is halved. Due to the reduced peak memory, tensor parallelism can now span fewer devices (e.g., from 8-way to 4-way), reducing communication cost.

Interlaced pipeline. Figure 6 shows the pipeline schedule searched under the constraints specified in Table 6. The embedding layer is partitioned across four devices using tensor parallelism. The remaining components (i.e., non-embedding layers) are separated to distinct device groups following `staged_spmd`. During the ordering search, all the layers compose into a schedule that resembles executing embedding layers and an 1F1B-like schedule following a time-sharing pattern. There are two columns with 0-th embedding because the embedding layer is used twice, one at the beginning of the model and the other at the end. Thanks to the scheduling search, the pipeline can reach a stable phase with zero bubbles as shown on the right of the figure.

3F1B pipeline. Figure 2 displays the timeline for the 3F1B pipeline which has been described in §4.2. The constraints outlined in Table 7 define how forward and backward passes interleave in the stable state of the pipeline. The schedule for the warm-up and cool-down phases remains unspecified. These phases are tailored through the search process.

8.3 End-to-End Performance

We evaluate the three new parallelization plans on Swin-Transformer, T5, and AlphaFold2, respectively, with different model configurations and on varying number of GPUs.

8.3.1 Experimental Setup

Machine configurations. Our evaluation is performed on DGX-2 clusters with 32 NVIDIA Tesla V100 (32GB) GPUs. Each server is equipped with 16 GPUs that are connected via NVLink [4]. Servers are interconnected with 8 InfiniBand 100 Gbps network adapters. All the servers are installed with NCCL 2.14 [3] and PyTorch v2.0.1 [43]. As 8 × 100 Gbps InfiniBand is a high-end hardware configuration, we also demonstrate the training performance on commodity hardware that is prevalent in many organizations [8]. Specifically, we conducted experiments on DGX-1 clusters with 32 NVIDIA Tesla V100 (32GB) GPUs, each equipped with 1 InfiniBand 100 Gbps network adapter in §8.3.5.

Model configurations. Table 9 summarizes the configurations of SwinTransformer, T5, and AlphaFold2, each of which has four different model configurations ranging from small models to large ones. For each configuration, we list its number of parameters, number of layers, hidden dimensions, and number of heads. For example, <1.8B, 32 layers, hidden size 512, 16 heads> is a configuration for SwinTransformer. The four small to large configurations for each model run on 4, 8, 16, and 32 GPUs respectively.

Baseline systems. We compared nnScaler with three popular distributed training systems: 1) Megatron-LM [39] is designed to train transformer-based models, which hierarchically combines pipeline parallelism with data and tensor parallelism. For pipeline parallelism, it evenly partitions model layers into

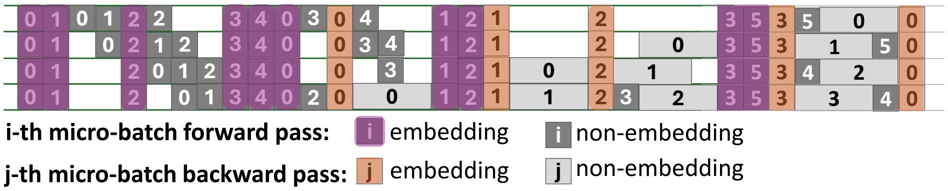


Figure 6: Interlaced pipeline plan found by nnScaler.

Model	SwinTransformer [34]			
Param# (B)	1.8	5.0	10	27
Layer#	32	40	48	56
Hidden	512	768	1K	1.5K
Head#	16	24	32	32
Model	T5 [45, 62]			
Param# (B)	3.9	11	21	47
Layer#	48	64	64	64
Hidden	2K	3K	4K	6K
Head#	32	48	64	96
Model	AlphaFold2 [27]			
Param# (B)	0.087	0.93	2.4	3.2
Layer#	48	64	96	128
Hidden	256	512	1K	1K
Head#	8	16	32	32

Table 9: Model architecture with the increasing number of GPUs. K: thousand. B: billion.

pipeline stages, and each stage can be further applied with data and tensor parallelism. 2) Alpa [65] is an automatic parallelization system for deep learning models under the 3D parallelization space. Its search algorithm and training system are currently based on TensorFlow. To conduct a side-by-side comparison, we implemented the Alpa's search algorithm as a policy in nnScaler. 3) DeepSpeed [47] is a distributed training system similar to Megatron-LM. It supports pipeline, data, and tensor parallelism, Additionally, it incorporates techniques including ZeRO [46] and ZeRO-Offload [48] to optimize GPU memory usage. ZeRO mainly optimizes memory usage of optimizer states by keeping a single copy in data parallelism. ZeRO-Offload offloads weights to CPU memory to reduce the memory pressure of GPU and retrieves them after they are used. It does not support offloading activation tensors.

Neither Megatron-LM nor DeepSpeed features automatic search for parallelization plans in their supported parallelization space. Therefore, we manually found the best-performing plans for them by separately traversing the degrees of pipeline, tensor, and data parallelism respectively. In all the following experiments, we applied layer-wise recompute [11] to reduce the memory consumption of activation tensors. Following the common practice [29, 65], we used the aggregated effective TFLOPS as our performance metric.

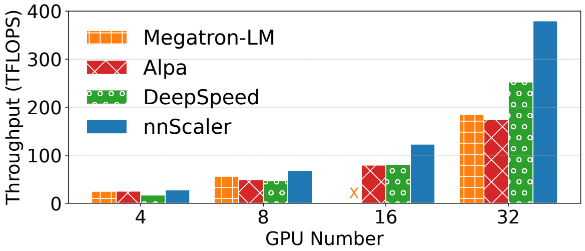


Figure 7: End-to-end training throughput of SwinTransformer. "×" denotes failure due to out of memory.

8.3.2 Results of SwinTransformer

Figure 7 illustrates the end-to-end training throughput of SwinTransformer on four systems. Both Megatron-LM and Alpa use pure tensor parallelism for all model configurations due to the substantial size of activation tensors is huge (e.g. 21GB for the first transformer layer for a 5.0B model), even with recompute applied. DeepSpeed employs ZeRO-Offload and ZeRO stage3 to optimize memory usage. Therefore, DeepSpeed is able to apply 2-way tensor parallelism for the 4 GPUs setting and 4-way tensor parallelism for the remaining three settings. Data parallelism is further applied to scale out across all the available GPUs. nnScaler applies Coshard on the first four layers (Attention+MLP) of SwinTransformer, because these layers occupy a large proportion of memory due to activation tensors. nnScaler applies 2-way, 2-way, 4-way, and 8-way tensor parallelism to the four configurations, respectively, combined with 2-way, 4-way, 4-way, and 4-way pipeline parallelism, respectively. Coshard has 6 partitions sequentially executed on each GPU for the 8 GPUs setting and 4 partitions for the remaining three settings. As shown in Figure 7, nnScaler is 1.2×, 1.5×, and 1.5× faster than DeepSpeed on 8, 16, and 32 GPUs, respectively. Although with ZeRO stage3 to reduce the degree of tensor parallelism to control the communication overheads, ZeRO stage3 still introduces heavy communication costs for weights on the critical path of forward and backward passes, especially when it is applied on 32 GPUs, which involves cross-node communication. In contrast, nnScaler applies Coshard to reduce peak memory, making it possible to use a less degree of tensor parallelism, which reduces communication costs.

Coshard is also used in the long-context post training of the Phi-3 series models to reduce the excessive memory usage due to the long context window [16].

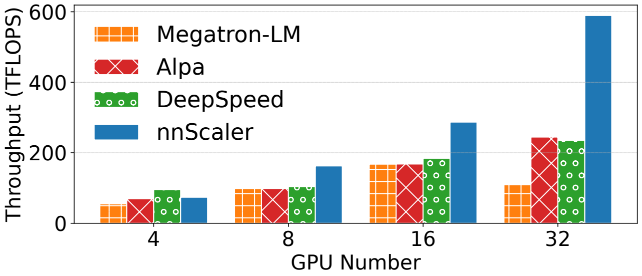


Figure 8: End-to-end training throughput of T5.

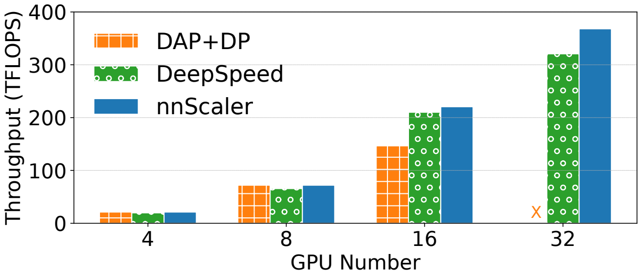


Figure 9: End-to-end training throughput of AlphaFold2. "×" denotes failure due to out of memory.

8.3.3 Results of T5

Figure 8 illustrates the end-to-end training throughput of T5. Megatron-LM uses 2-way tensor parallelism with 2-way pipeline parallelism for 4 GPUs, and uses pure tensor parallelism for 8, 16, 32 GPUs. For 4 GPUs, Alpa uses 3-way pipeline parallelism with the middle stage applying 2-way tensor parallelism. It *must* use pure tensor parallelism for 8, 16, and 32 GPUs due to large memory consumption. As T5 of 3.9B parameters is relatively small, DeepSpeed can use data parallelism with ZeRO stage3 for 4 GPUs. It applies 4-way tensor parallelism for 8, 16, and 32 GPUs, with ZeRO-Offload and ZeRO stage3 applied. Additionally, data parallelism is further applied to scale out to all the available GPUs. nnScaler applies the interlaced pipeline. The large embedding layer uses tensor parallelism on all the available GPUs. The remaining layers apply 4-way pipeline parallelism, with each stage applied 1-way, 2-way, 4-way, and 8-way tensor parallelism for 4, 8, 16, and 32 GPUs, respectively.

nnScaler performs 1.5×, 1.6×, and 2.5× better than DeepSpeed for 8, 16, and 32 GPUs respectively. Megatron-LM and Alpa have a low performance because the high degrees (e.g., 32) of tensor parallelism introduces high communication overheads, especially when the tensor parallelism spans more than one node. This is why Megatron-LM performs much worse with 32 GPUs. As Alpa searches for suitable partition options for tensor parallelism, many operators (e.g., dropout or layernorm) are replicated across nodes to reduce communication costs, and so it performs better than Megatron-LM. Although DeepSpeed has a lower degree of tensor parallelism, its performance is only comparable to Alpa because DeepSpeed uses ZeRO-Offload and ZeRO stage3 to make lower degrees of tensor parallelism feasible. ZeRO-Offload introduces high overheads due to the offloading of large embedding weight (e.g., 12GB in the 21B model). ZeRO stage3 also introduces high communication costs, such as the online gathering of (embedding) weights on the critical path. This shows the effectiveness of the proposed interlaced pipeline on models like T5, compared to conventional approaches like tensor parallelism, ZeRO-Offload, and ZeRO stage3. Note that to highlight the advantage of interlaced pipeline, nnScaler tentatively disables ZeRO in the experiments in Figure 8. And nnScaler still outperforms others in most cases except for the 4-GPU case when T5 is small enough to fit-in memory after applying data parallelism with ZeRO (i.e., DeepSpeed's plan).

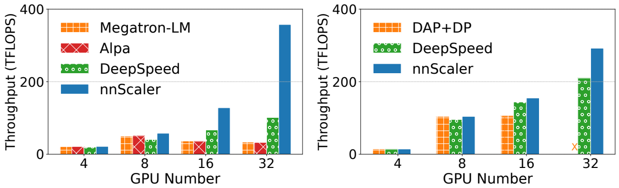
8.3.4 Results of AlphaFold2

Figure 9 shows the end-to-end training throughput of AlphaFold2. In this experiment, we compared nnScaler with two baselines. One is DAP [13], which is a handcrafted tensor parallelism specifically designed for AlphaFold2. We also applied data parallelism to scale out DAP, referred to as DAP+DP. For 4 and 8 GPUs, DAP+DP uses pure data parallelism since the models are small. It uses 4-way tensor parallelism with 4-way data parallelism for 16 GPUs. The other baseline is DeepSpeed. As the model sizes are much smaller than those of SwinTransformer and T5, the application of ZeRO-Offload is not necessary. DeepSpeed uses pure data parallelism for 4, 8, and 16 GPUs with ZeRO stage3, and uses 2-way tensor parallelism with 16-way data parallelism for 32 GPUs. nnScaler also uses pure data parallelism for 4 and 8 GPUs. It applies the 3F1B pipeline for 16 and 32 GPUs. For 16 GPUs, nnScaler uses 4-way pipeline parallelism with 4-way data parallelism, while for 32 GPUs it uses 2-way tensor parallelism with 2-way pipeline parallelism and 8-way data parallelism.

nnScaler performs 1.5× better than DAP+DP on 16 GPUs and 1.1× better than DeepSpeed on 32 GPUs. DeepSpeed performs better than DAP+DP on 16 GPUs, because the activation tensors in AlphaFold2 are large, and the communication of activation tensors using 2-way tensor parallelism is more efficient than that using 4-way tensor parallelism. nnScaler performs better than DeepSpeed because the customized 3F1B pipeline reduces communication costs. The training conducted on multiple nodes, which is common for large model training, amplifies the advantage of pipeline parallelism.

8.3.5 Experiments on Less Powerful Hardware

To demonstrate the effectiveness of the new parallelization plans and understand how different hardware affects training performance, we evaluate SwinTransformer and AlphaFold2 in the DGX-1 cluster. As shown in Figure 10a, nnScaler is 1.9× and 3.5× faster than DeepSpeed on 16 and 32 GPUs, respectively. Compared with data shown in Figure 7, for 32 GPUs, the performance of nnScaler is degraded by 6%, while that of DeepSpeed, Alpa, and Megatron-LM is degraded by 60%, 82% and 82%, respectively. The degradation of nnScaler is smaller because the parallelization plan (i.e., Coshard) used by nnScaler optimizes the communication cost, and thereby it tolerates the changes in communication bandwidth. Figure 10b shows the results of AlphaFold2 on DGX-1. The relative performance gain of nnScaler is also improved to 1.1× and 1.4× over DeepSpeed on 16 and 32 GPUs, respectively. The lower bandwidth cross nodes in DGX-1 further amplifies the advantage of pipeline parallelism, rendering the 3F1B pipeline much faster than tensor parallelism in DAP+DP and DeepSpeed. These experiments indicate that with the flexible customization of parallelization plans and automatic plan search, nnScaler can adapt more flexibly to changes in hardware.



(a) SwinTransformer

(b) AlphaFold2

Figure 10: End-to-end training throughput on DGX-1. "×" denotes the failure of training due to out-of-memory.

8.4 Search Efficiency with Constraints

Algorithm 1 suggests that the parallelization plan search cost in nnScaler consists of: (1) operator transformation and placement cost (i.e., line 1-4 in Algorithm 1), and (2) operator temporal ordering cost (i.e., line 5 in Algorithm 1). Figure 11 illustrates the end-to-end search cost, as well as the breakdown time of the three customized spaces defined in §4.2 for different model configurations using the policy illustrated in §5. The search on SwinTransformer's space takes less than 150s. The search time increases with the increase of model size as the number of operators increases. The ordering search for T5 takes around 150s due to an absence of constraints on the ordering in T5's space. There is almost no search cost of the ordering in SwinTransformer and AlphaFold2. For SwinTransformer, the order is largely determined by data dependencies, and for AlphaFold2, the ordering constraints greatly reduces the space.

Figure 12 further shows the temporal ordering search time of the 3F1B schedule with and without constraints. The left figure shows that the search time increases exponentially with the increase of stage number. However, with constraints applied, the search time is kept within 60s, resulting in 11.7× speedup in finding the efficient temporal ordering for 4 stages. This is attributed to the temporal ordering constraints in Table 7, where the ordering constraints of independent forward and backward operators from different micro-batches are explicitly specified, leading to a significantly reduced search space exposed to the search algorithm (i.e., Tessel). For the case of 4 stages, the right figure further shows the search time as each ordering constraints from Table 7 is applied one by one. The first constraint reduces the search time by 100s. The second constraint further reduces it by 50% of search time. This demonstrates the importance of constraints.

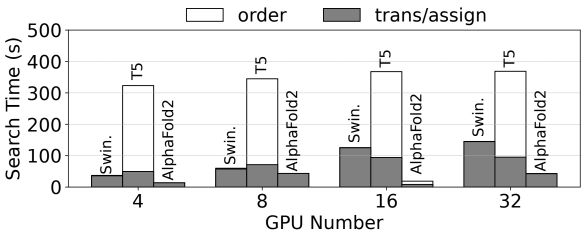


Figure 11: End-to-end search cost of each model.

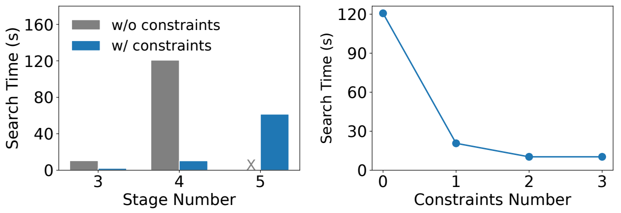


Figure 12: Search time of the 3F1B schedule with and without constraints. "×" denotes a search time exceeding one hour.

9 Related Work

Existing parallelization search spaces. Recently, data, tensor, and pipeline parallelisms [1, 18, 24, 30, 46] have been widely used in distributed DNN training. Various memory optimizations [11,23,28] have also been adopted to exploit large-scale model training under GPU memory constraints. Systems such as Megatron-LM [29,39,50], DeepSpeed [47], Piper [54], Unity [57], and Alpa [65], combine multiple parallelisms and memory optimizations to accelerate distributed DNN training. However, these solutions fall short in because they rely on empirical parallelism configurations and have limited execution scheduling choices. Thus, despite their successful applications on existing training workloads, they still fail to fully utilize hardware capabilities. In contrast, nnScaler provides a different approach to parallelization, supporting the expression of parallelization sub-spaces with fine-grained transformation and scheduling primitives. Consequently, nnScaler is compatible with them as all these solutions can be achieved using particular constraints. In additional, nnScaler is able to support more flexible and efficient parallelization plans that extend beyond the aforementioned parallelization sub-spaces, which is considerably crucial for continuously evolving DNN models.

Explorations on specific parallelization plans. Parallelization strategies tailored for specific scenarios play a crucial role in optimizing the performance of parallel computing frameworks. For instance, Transformer Parallelism [50], DAP [13], and Sequence Parallelism [29] are designed for specific model architectures, showcasing a nuanced approach to parallelization. To address the need for optimized pipeline orchestration, innovative scheduling strategies have been proposed by GPipe [24], 1F1B [18, 50], and Chimera [30]. Furthermore, optimizations such as Gradient Accumulation [60], Recompute [11], Chain-recompute [28], Swap [23], and All-reduce Overlap [49] specifically target improvements in memory or communication efficiency. These strategies can be seamlessly incorporated into nnScaler's plan with appropriate constraints, eliminating the need for a comprehensive system overhaul and demonstrating the platform's adaptability.

Parallelization plan search and others. To improve training performance with combined parallelisms, DNN systems [22, 26, 37, 54, 59, 63, 65] use different searching techniques to find efficient parallelism configurations. Most recently, Alpa [65] leverages both integer programming and dynamic programming solvers, and Tessel [32] enables the exploration of schedule search in pipeline parallelism, significantly harnessing performance potential beyond manually crafted pipeline schedules. nnScaler, as a parallelization plan engine that emphasizes customizing the parallelization space through constraints, is complementary to the above algorithms and can leverage them to speedup the search within a customized space.

Kernel fusion and tuning optimizations. Besides efficient parallelization plans, kernel fusion and tuning [10, 15, 36, 66] can also improve execution efficiency on a device by fusing multiple consecutive operators into a single more performant GPU kernel. For instance, Flash-Attention [15] fuses multiple operations within the attention layer into a single kernel to improve performance with reduced I/O. These techniques are complementary to nnScaler as they can be applied after nnScaler partitions computation across devices, to further enhance the local computation efficiency on each device.

10 Conclusions

nnScaler is a framework that enables domain experts to leverages three primitives, `op-trans`, `op-assign`, and `op-order`, along with constraints to construct arbitrary search spaces for parallelization plans given any DNN model. This approach represents a more general abstraction to describe both existing parallelization search spaces and new spaces. Experiments show that nnScaler is able to construct new spaces that lead to the discovery of new parallelization plans for deep learning training on emerging DNN models as well as mainstream models, significantly outperforming existing plans.

11 Acknowledgements

We sincerely thank our shepherd and all the anonymous reviewers for their valuable feedback. We also thank Madan Musuvathi, Guodong Liu, Xiaoxiang Shi and Jun Huang for their suggestions and help.

References

[1] Distributed Data Parallelism. https://pytorch.org/docs/stable/notes/ddp.html. [Online; accessed Sep. 2022].

[2] Introducing ChatGPT. https://openai.com/blog/chatgpt. [Online; accessed Nov. 2023].

[3] NVIDIA collective communications library. https://developer.nvidia.com/nccl. [Online; accessed Aug. 2021].

[4] NVIDIA NVLINK. http://www.nvidia.com/object/nvlink.html.

[5] Project nnScaler. https://github.com/microsoft/nnscaler.

[6] PyTorch Team. https://pytorch.org/docs/stable/checkpoint.html. [Online; accessed Aprl. 2022].

[7] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[8] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[12] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning (ICML)*, pages 794–803. PMLR, 2018.

[13] Shenggan Cheng, Ruidong Wu, Zhongming Yu, Binrui Li, Xiwen Zhang, Jian Peng, and Yang You. Fastfold: Reducing alphafold training time from 11 days to 67 hours. *arXiv preprint arXiv:2203.00854*, 2022.

[14] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:16344–16359, 2022.

[16] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[17] Richard Evans, Michael O'Neill, Alexander Pritzel, Natasha Antropova, Andrew Senior, Tim Green, Augustin Žídek, Russ Bates, Sam Blackwell, Jason Yim, Olaf Ronneberger, Sebastian Bodenstein, Michal Zielinski, Alex Bridgland, Anna Potapenko, Andrew Cowie, Kathryn Tunyasuvunakool, Rishub Jain, Ellen Clancy, Pushmeet Kohli, John Jumper, and Demis Hassabis. Protein complex prediction with alphafold-multimer. *bioRxiv*, 2021.

[18] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 431–445, 2021.

[19] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–568, 2021.

[20] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[21] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, Chen Zhang, Chao Li, and Minyi Guo. Amanda: Unified instrumentation framework for deep neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[22] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. Towards optimal placement and scheduling of dnn operations with pesto. In *Proceedings of the 22nd International Middleware Conference*, pages 39–51, 2021.

[23] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1341–1355, 2020.

[24] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 103–112, 2019.

[25] Hugging Face Community. Model hub – natural language processing. `https://huggingface.co/models`. [Online; accessed Jun. 2023].

[26] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML*, 2019.

[27] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[28] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *9th International Conference on Learning Representations, (ICLR), Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[29] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[30] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–14, 2021.

[31] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning (ICML)*, pages 6543–6552. PMLR, 2021.

[32] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. Tessel: Boosting distributed execution of large dnn models via flexible schedule search. *arXiv preprint arXiv:2311.15269*, 2023.

[33] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, pages 163–181, 2024.

[34] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin transformer v2: Scaling up capacity and resolution. *arXiv preprint arXiv:2111.09883*, 2021.

[35] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[36] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 881–897, 2020.

[37] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15, 2021.

[40] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[41] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 891–905, 2020.

[42] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–29, 2019.

[43] PyTorch Team. PyTorch. https://pytorch.org//. [Online; accessed Mar. 2022].

[44] PyTorch Team. PyTorch Distributed Communication Package. https://pytorch.org/docs/stable/distributed.html. [Online; accessed Nov. 2023].

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 2020.

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 3505–3506, 2020.

[48] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC)*, pages 551–564, 2021.

[49] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using GPU model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[51] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023.

[52] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. *arXiv preprint arXiv:2405.05254*, 2024.

[53] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[54] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[55] PyTorch Team. TorchFX. https://pytorch.org/docs/stable/fx.html.

[56] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[57] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–284, 2022.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[59] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*, pages 1–17, 2019.

[60] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, pages 84–97, 2016.

[61] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[62] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*, 2020.

[63] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:906–917, 2020.

[64] Bo Zheng, Li Dong, Shaohan Huang, Saksham Singhal, Wanxiang Che, Ting Liu, Xia Song, and Furu Wei. Allocating large vocabulary capacity for cross-lingual language model pre-training. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 3203––3215, 2021.

[65] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 559–578, 2022.

[66] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. Roller: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–248, 2022.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

ChameleonAPI: Automatic and Efficient Customization of Neural Networks for ML Applications

Yuhan Liu[‡], Chengcheng Wan[*], Kuntai Du[‡], Henry Hoffmann[‡], Junchen Jiang[‡], Shan Lu[‡†], Michael Maire[‡]
[‡]*University of Chicago* [*]*East China Normal University* [†]*Microsoft Research*

Abstract

ML APIs have greatly relieved application developers of the burden to design and train their own neural network models—classifying objects in an image can now be as simple as one line of Python code to call an API. However, these APIs offer the same pre-trained models *regardless* of how their output is used by different applications. This can be suboptimal as not all ML inference errors can cause application failures, and the distinction between inference errors that can or cannot cause failures varies greatly across applications.

To tackle this problem, we first study 77 real-world applications, which collectively use six ML APIs from two providers, to reveal common patterns of *how ML API output affects applications' decision processes*. Inspired by the findings, we propose *ChameleonAPI*, an optimization framework for ML APIs, which takes effect without changing the application source code. ChameleonAPI provides application developers with a parser that automatically analyzes the application to produce an abstract of its decision process, which is then used to devise an application-specific loss function that only penalizes API output errors critical to the application. ChameleonAPI uses the loss function to efficiently train a neural network model customized for each application and deploys it to serve API invocations from the respective application via existing interface. Compared to a baseline that selects the best-of-all commercial ML API, we show that ChameleonAPI reduces incorrect application decisions by 43%.

1 Introduction

The landscape of ML applications has greatly changed, with the rise of ML APIs significantly lowering the barrier of ML application developers. Instead of designing and managing neural network models by themselves via frameworks like TensorFlow and PyTorch, application developers can now simply invoke ML APIs, provided by open-source libraries or commercial cloud service providers, to accomplish common ML tasks like object detection, facial emotion analysis, etc. This convenience thus gives rise to a variety of ML applications on smartphones, tablets, sensors, and personal assistants [9, 29, 50, 65].

Although ML APIs have eased the integration of ML tasks with applications, they are suboptimal by serving different applications with the same neural network models. This issue is particularly striking when applications use the ML API results to make control-flow decisions (also referred to as *application decisions* in this paper). Different applications may check the result of the same ML API using different control-flow code structures and different condition predicates, a process that we refer to as the application's *decision process* (see §2 for the formal definition). Due to the heterogeneity across applications' decision processes, we make two observations.

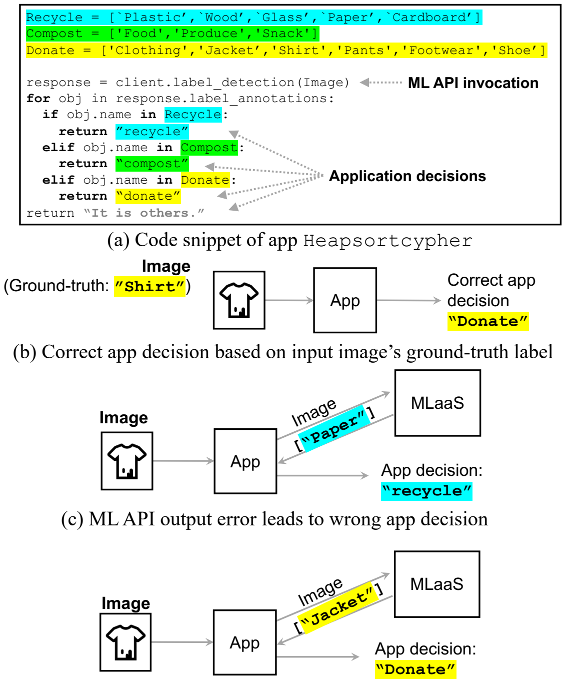
- First, some incorrect ML API outputs may still lead to correct application decisions, with only certain *critical errors* of API output affecting the application's decision.
- Second, among all possible output errors of an ML API, which ones are critical *vary* significantly across applications that use this API. That is, the same API output error may have a much *greater* effect on one application than on another.

Figure 1 illustrates the decision process of a garbage-classification application `Heapsortcypher` [49]. It first invokes Google's classification API upon a garbage image. Then, based on the returned labels, a simple logic is used to make the *application decision* about which one of the pre-defined categories (`Recycle`, `Compost`, and `Donate`) or `others` the image belongs to. For example, for an input image whose ground-truth label is `"Shirt"`, the correct application decision is `Donate`, as shown in Figure 1 (b).

For this application, when the classification API fails to return `"Shirt"`, the application decision may or may not be wrong. For example, Figure 1 (c) and (d) show two possible wrong API output: if the output is `"Paper"`, the application will make a wrong decision of `Recycle`; however, if the output is `"Jacket"`, the application will make the correct decision of `Donate` despite not matching the ground-truth label. More subtly, if the API returns a list of two labels, `"Shirt"` and `"Paper"`, the application would make a correct decision if `"Shirt"` is ordered before `"Paper"` by the API, but would make a wrong decision if `"Paper"` is ordered before `"Shirt"`. The reason is that the application logic, the `for` loop in Figure 1 (a), checks one API-output label at a time. As we will see later, there are also other ways that applications check the API-output list, which will affect application decision differently.

As we can see, for a specific application, some errors of an ML API may be critical, like mis-classifying the shirt to `"Paper"` in the example above, and yet some errors may be non-critical, like mis-classifying the shirt image as `"Jacket"` or classifying the shirt image as both `"Shirt"` and `"Paper"` in the examples above. Which errors are critical varies, depending on the application's decision process.

These observations regarding the critical errors specific to



```
Recycle = [`Plastic',`Wood',`Glass',`Paper',`Cardboard']
Compost = ['Food','Produce','Snack']
Donate = ['Clothing','Jacket','Shirt','Pants','Footwear','Shoe']

response = client.label_detection(Image)
for obj in response.label_annotations:
  if obj.name in Recycle:
    return "recycle"
  elif obj.name in Compost:
    return "compost"
  elif obj.name in Donate:
    return "donate"
return "It is others."
```

(a) Code snippet of app `Heapsortcypher`

(b) Correct app decision based on input image's ground-truth label

(c) ML API output error leads to wrong app decision

(d) ML API output error still leads to correct app decision

Figure 1: *An example ML application whose decision depends on the output of ML API (multi-label classification), but not all errors of ML API output have the same effect.*

each application suggest substantial room for improvement by customizing the ML API, essentially the neural network model underneath the API, for individual application's decision process. In particular, for a given application, the customized model can afford having more errors less critical to the application for the benefit of having fewer critical errors that cause wrong application decisions.

Thus, our goal is to allow ML APIs and their underlying neural network models to be *automatically* customized for a given application, so as to *minimize* incorrect application decisions *without* changing the application's source code or interface between ML API and software exposed to developers. This way, application developers who do not have the expertise to design and train customized ML models can still enjoy the accessibility of generic ML APIs while getting closer to the accuracy of ML models customized for the application.

No prior work shares the same goal as us. The closest line of prior work specializes DNN models for given queries [7, 8, 36, 37, 43], but they require application developers to use a domain specific language (*e.g.,* in SQL [36]) instead of general programming languages, like Java and Python, and mostly focus on reducing the DNN's size. In contrast, we keep both the ML API interface and the application source code intact while avoiding incorrect decisions for ML applications.

With the aforementioned goal, this paper makes two contributions. *First,* we run an empirical study over 77 real-world applications that collectively use six ML APIs to reveal several common patterns of how the outputs of ML APIs affect the application decisions (§2).

Our study identifies two types of ML API output that are used by applications to make control-flow decisions (categorical labels and sentiment scores), and three types of decision types (True-False, Multi-Choice, and Multi-Selection) with different implications regarding which ML API output errors are critical to the application.

Our study also quantitatively reveals opportunities of model customization. (1) Although popular image-classification models are trained to recognize as many as 19.8K different labels, the largest number used by any one application for decision making is only 54. Consequently, mis-classification among the remaining tens of thousands of labels are completely irrelevant to an application. (2) More importantly, applications tend to treat multiple labels (4.7 on average) as one equivalence class in their decision making, such as labels `Plastic`, `Wood`, `Glass`, `Paper`, and `Cardboard` in Figure 1(a). Mis-classification among those labels inside one equivalence class does not matter. (3) Which labels are relevant to an application's decision making vary greatly across applications, with only 12% of application pairs share any labels used for their decision making.

Second, inspired by the empirical study, we propose ChameleonAPI, which customizes and serves ML models behind the ML API for each given application's decision process, without any change to the existing ML API or the application source code (§3). ChameleonAPI works in three steps. First, it provides a parser that analyzes application source code to extract information about how ML inference results are used in the application's decision process. Based on the analysis result, ChameleonAPI then constructs the loss function to reflect which ML model output is more relevant to the given application as well as the different severity of ML inference errors on the application decisions. The ML model will be retrained accordingly using the new loss function. Finally, when the ML API is invoked by the application at runtime, a customized ML model will be used to serve this query.

We evaluate ChameleonAPI on 57 real-world open-source applications that use Google and Amazon's vision and language APIs. We show that ChameleonAPI's re-trained models reduce 48% of incorrect decisions compared to the off-the-shelf ML models and 50% compared to the commercial ML APIs. Even compared with a baseline that selects the best-of-all commercial ML API, ChameleonAPI reduces 43% of incorrect decisions. ChameleonAPI only takes up to 24 minutes on a GeForce RTX 3080 GPU to re-train the ML model.

Our code is publicly available at `https://github.com/UChi-JCL/chameleonAPI`.

2 Understanding Application Decision Process

We conduct an empirical study to understand how applications make decisions based on ML APIs (§2.3), and how this

ML API name	ML task	Provider	# of apps
label_detection	Vision::Image classification	Google	29
detect_labels	Vision::Image classification	Amazon	11
object_localization	Vision::Object detection	Google	8
analyze_sentiment	Language::Sentiment analysis	Google	14
analyze_entities	Language::Entity recognition	Google	6
classify_text	Language::Text classification	Google	9

Table 1: *Summary of applications used in our empirical study.*

decision making logic implies the different severity of ML inference errors (§2.4). This study will reveal why and how to customize the ML API backend for each application. As a representative sample of ML APIs, this study focuses on cloud AI services due to their popularity.

2.1 Definitions

Preliminaries: We begin with basic definitions.

- *Application decision*: the collective control-flow decisions (i.e., which branch(es) are taken) made by the application under the influence of a particular ML API output.
- *Incorrect ML API output:* a situation when the API output differs from the API input's human-labeled ground truth. We refer to such ML API outputs as *API output errors*.
- *Correct decision*: the application decision if the API output is the same as the human-labeled ground-truth of the input.
- *Application decision failure*: a situation when the application decision is different from the correct decision, also referred to as *application failure* for short in this paper.

Software decision process: Given these definitions, an application's *software decision process* (or decision process for short) is the logic that maps an ML API output to an application decision. The code snippet in Figure 1 shows an example decision process, which maps the output of a classification ML API on an image to the image's recycling categorization specific to this application.

Critical and non-critical errors: For a given decision process, some API output errors will still lead to a correct decision, whereas some API output errors will lead to an incorrect decision and hence an application failure. We refer to the former as *non-critical errors*, and the latter as *critical errors*.

2.2 Methodology

Our work focuses on applications that use ML API output to make control-flow decisions. To this end, we look at 77 open-source applications which collectively use six widely used vision and language APIs [10,65] offered by two popular cloud AI service providers, as summarized in Table 1.

These applications come from two sources. First, we study all 50 applications that use vision and language APIs from a recently published benchmark suite of open-source ML applications [66]. Second, given the popularity of image classification APIs [11, 12], we additionally sample 27 applications from GitHub that use Google and Amazon image classification APIs (16 for the former and 11 for the latter). We obtain these 27 by checking close to 100 applications that use image classification APIs and filtering out those that directly print out or store the API output. Every application in our benchmark suite uses exactly one ML API for decision making.

Threats to validity: While many applications use the APIs listed in Table 1, there are a few other APIs not covered in our study. A few vision and language-related ML tasks are not as popular and hence are not covered in our study (*e.g.,* face recognition and syntax analysis). Speech APIs are not covered, because their outputs are rarely used to affect application control flow based on our checking of open-source applications. Finally, our study does not cover applications that use ML APIs offered by other cloud or local providers.

2.3 Understanding the decision mechanism

***Q1**: What types of ML API outputs are typically used by applications to make decisions?*

ML APIs produce output of a variety of types. The sentiment analysis API outputs a list of floating-point value pairs (`score` and `magnitude`), describing the sentiment of the whole document and every individual sentence; the other five APIs in Table 1 each produces a list of categorical labels ranked in descending order of their confidence scores, which is also part of the output. Some APIs' output also contains other information, like coordinates of bounding boxes, entity names, links to Wikipedia URLs, and so on. Among all these, only two types have been used in application decision processes of our studied application: the floating-point pair (`score` and `magnitude`) and the categorical labels.

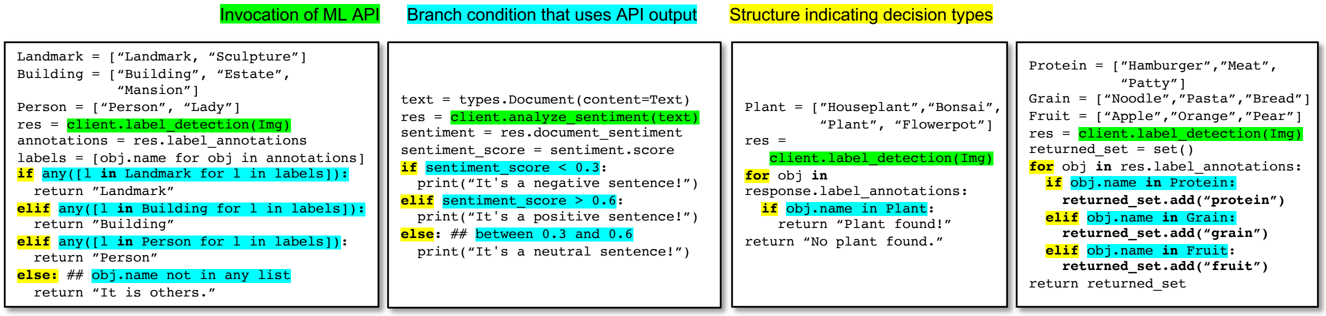
For the 63 applications that use categorical-label output from the five APIs (all except `analyze_sentiment` in Table 1), they each define one or more *label lists* and check which label list(s) an API output label belongs to. The code snippet of a landmark classification application in Figure 2(a) is an example of this. It calls the `label_detection` API with a sight-seeing image and checks the output labels to see if the image might contain `Landmark`, or just ordinary `Building`, or `Person`.

For the 14 applications that use the `analyze_sentiment` API, they each define several *value ranges* and check which range the sentiment `score` and/or `magnitude` falls in. The code snippet of `FoodDelivery` [48] in Figure 2(b) is an example. This application calls `analyze_sentiment` with a restaurant review text, and then checks the returned sentiment `score` to judge if the review is negative, positive, or neutral.

***Q2**: What type of decisions do applications make?*

We observe three categories of ML-based decision making, which we name following common question types in exams:

(1) True-False decision, where a single label list or value range is defined and one selection is allowed: either the ML API output belongs to this list/range or not. This type occurs


Invocation of ML API
Branch condition that uses API output
Structure indicating decision types

```
Landmark = ["Landmark, "Sculpture"]
Building = ["Building", "Estate",
            "Mansion"]
Person = ["Person", "Lady"]
res = client.label_detection(Img)
annotations = res.label_annotations
labels = [obj.name for obj in annotations]
if any([l in Landmark for l in labels]):
  return "Landmark"
elif any([l in Building for l in labels]):
  return "Building"
elif any([l in Person for l in labels]):
  return "Person"
else: ## obj.name not in any list
  return "It is others."
```

(a) `Aander-ETL`
(Label output, Multi-Choice, App-Order)

```
text = types.Document(content=Text)
res = client.analyze_sentiment(text)
sentiment = res.document_sentiment
sentiment_score = sentiment.score
if sentiment_score < 0.3:
  print("It's a negative sentence!")
elif sentiment_score > 0.6:
  print("It's a positive sentence!")
else: ## between 0.3 and 0.6
  print("It's a neutral sentence!")
```

(b) `FoodDelivery`
(Float-point value output, Multi-Choice)

```
Plant = ["Houseplant","Bonsai",
          "Plant", "Flowerpot"]
res =
   client.label_detection(Img)
for obj in
response.label_annotations:
  if obj.name in Plant:
    return "Plant found!"
return "No plant found."
```

(c) `Plant-watcher`
(Label output, True-False)

```
Protein = ["Hamburger","Meat",
           "Patty"]
Grain = ["Noodle","Pasta","Bread"]
Fruit = ["Apple","Orange","Pear"]
res = client.label_detection(Img)
returned_set = set()
for obj in res.label_annotations:
  if obj.name in Protein:
    returned_set.add("protein")
  elif obj.name in Grain:
    returned_set.add("grain")
  elif obj.name in Fruit:
    returned_set.add("fruit")
return returned_set
```

(d) `The-Coding-Kid`
(Label output, Multi-Select)

Figure 2: *Code snippets from five example applications where ML API output affects control flow decisions in different ways.*

in about one third of the applications in our study. For example, the plant management application `Plant-watcher` [57] (Figure 2(c)) checks to see if the image contains plants or not.

(2) Multi-Choice decision, where multiple lists of labels or value ranges are defined, and one selection is allowed. The ML API output will be assigned to *at most one* list or range; the application's decision making logic determines which of these lists/ranges the output belongs to, or determines that the output belongs to none of them. This type of decision is the most common, occurring in about 45% of benchmark applications. The garbage classification application discussed in §1 makes such a Multi-Choice decision. It decides which one of the following classes the input image belongs to: `Recycle`, `Compost`, `Donate`, or none of them.

(3) Multi-Select decision, where multiple label lists or value ranges are defined, and *multiple* selections are allowed about which label lists or value ranges the ML API output belongs to. This type of decisions occur in close to a quarter of the applications. Figure 2(d) illustrates such an example from the nutrition advisor application `The-Coding-Kid` [62]. This application defines three label lists to represent nutrition types: `Protein`, `Grain`, and `Fruit`, and it checks to find all the nutrition types present in the input image.

In the remainder of the paper, we will use **target class** to refer to a label list (or a value range) that is used to match against a categorical label (or a value). For instance, the code snippet in Figure 2(a) has three label lists as its target classes (`[Landmark, Sculpture]`, `[Building, Estate, Mansion]`, and `[Person, Lady]`), and the code snippet in Figure 2(b) has three value ranges as its target classes (`<0.3`, `>0.6`, and in between).

Q3: How do applications reach Multi-Choice decisions?

When the ML API outputs multiple labels, the outcome of a Multi-Choice decision varies depending on which *matching order* is used. First, the matching order can be determined by the API output. For example, the garbage classification application (Figure 1) first checks whether the first label in the API output matches any target class. If so, later API output labels will be skipped, even if they might match with a different class. If there is no match for the first label, the second output label is checked, and so on. These labels are ranked by the API in the descending order of their associated confidence scores, so we refer to such a matching order as API-order. It is used by 80% of applications that make Multi-Choice decisions.

The matching order can also be specified by the application, referred to as App-order. For instance, regardless the API output, application `Aander-ETL` [1] (Figure 2(a)) always first checks if the `Landmark` class matches with *any* output label. If there is a match, the decision is made. Only when it fails to match `Landmark`, will it move on to check the next choice, `Building`, and so on. This matching order is used by 20% of applications that make Multi-Choice decisions.

2.4 Understanding the decision implication

Q4: Does an application need ML APIs that can accurately identify thousands of labels?

ML models behind popular ML APIs are well trained to support a wide range of applications. For example, Google and Microsoft's image-classification APIs are capable of identifying more than 10000 labels [44], while Amazon's image-classification API can identify 2580 labels [3]. However, for each individual application, its decision making only requires classifying the input image into a handful of target classes: 7 at most in our benchmark applications. The largest number of image-classification labels checked by an application is 54, a tiny portion of all the labels an image-classification API could output.

Clearly, for any application, a customized ML model that focuses on those target classes used by the application's decision process has the potentially to out-perform the big and generic ML model behind ML APIs. How to accomplish the customization without damaging the accessibility of ML APIs will be the goal of ChameleonAPI.

Q5: Are there equivalence classes among ML API outputs in the context of application decision making?

For the 63 applications that make decisions based on API output of categorical labels, they present 121 target classes in total, each containing 4.7 labels on average (3 being the median). Only 35 target classes in 22 applications contain a single label. For the 14 applications that make decisions based on floating-point sentiment `score` and `magnitude`, their target classes *all* contain an infinite number of `score` or `magnitude` values. In other word, no class contains just a single value.

Clearly, the wide presence of multi-value target classes creates equivalence classes among output returned by the API—errors within one equivalence class are *not* critical to the corresponding application. This offers another opportunity for ML customization.

***Q6**: How much difference is there between different applications' target classes?*

Overall, the difference is significant. We have conducted pair-wise comparison between any two applications in our benchmark suit, and found that 88% of application pairs share *no* common labels in any of their target classes. Similarly, among the 381 labels that appear in at least one application's target classes, 88% of them appear in only one application (*i.e.,* 335 out of 381 labels).

Clearly, there is *little overlap* among the target classes of different applications, again making a case for *per-application* customization of the ML models used by the ML APIs.

***Q7**: Do different decision mechanisms imply different sensitivity to output errors of ML APIs?*

Even for two applications that have the same target classes, if they try to make different types of decisions, they will have different sensitivity to ML API output errors—some API errors might be critical to one application, but not to the other. For example, errors that affect the selection of different target classes are equally critical to Multi-Select decisions. However, this is not true for Multi-Choice, where only the first matched target class matters. Furthermore, the matching order of a Multi-Choice decision affects which errors are critical. When the API-output order is used (*e.g.,*`HeapsortCypher` in Figure 1), an error on the first label in the API output is more likely to be critical than an error on other labels in the output. However, when App-order order is used (*e.g.,*`Aander-ETL` in Figure 2(a)), errors related to labels in the first target class (*e.g.,*`Landmark`) are more likely to be critical than those related to labels in later target classes (*e.g.,*`Person`).

Clearly, to customize ML models for each application, we need to take into account what is the decision type and what is the matching order (for Multi-Choice decisions).

3 Design of ChameleonAPI

Inspired by the study of §2, we now present ChameleonAPI which automatically customizes ML models for applications.

3.1 Problem formulation

Goal: For an application that uses ML APIs, our goal is to **minimize critical errors** in the API outputs for this application by efficiently re-training the original generic neural network models underneath these APIs into customized models; our approach stands in contrast to typical approaches that minimize *all* inference errors. In other words, the new ML

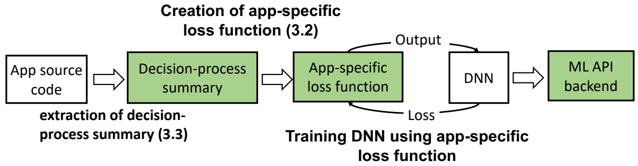

Creation of app-specific loss function (3.2)
App source code
Decision-process summary
App-specific loss function
Output
DNN
ML API backend
Loss
extraction of decision-process summary (3.3)
Training DNN using app-specific loss function

Figure 3: *The logical steps of how ChameleonAPI customizes d for individual applications.*

model should return outputs that lead the application process to the same decision as if the ground-truth of the input is returned by the ML API.

To formally state this objective, we denote how an application makes a decision by $App(API(\mathbf{x}))$, where $\mathbf{x}$ is the input to the ML API and $API(\mathbf{x})$ is the API output. Then for a given application decision process of $App(\cdot)$ and an input set $\mathbf{X}$[1], our goal is to train an ML model $DNN(\cdot)$ such that

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | App(API(\mathbf{x}_i)) \neq App(\widehat{API}(\mathbf{x}_i))\} \right|,$$
$$\text{where } API(\mathbf{x}_i) = F(DNN(\mathbf{x}_i)) \qquad (1)$$

Here, $\widehat{API}(\mathbf{x}_i)$ is a hypothetical API function that always returns the ground truth of input $\mathbf{x}_i$, and $F(\cdot)$ represents the postprocessing used by the API to translate a DNN output to an API output. For instance, an image classification model's output is a vector of confidence scores between 0 and 1 (each for a label), but the ML API will use a threshold θ to filter and return only labels with scores higher than θ, or the top k labels with the highest confidence scores.

Our goal in Eq 1 differs from the traditional goal of an ML model, which minimizes any errors in the API output, *i.e.,*

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | API(\mathbf{x}_i) \neq \widehat{API}(\mathbf{x}_i)\} \right|. \qquad (2)$$

Given that it is hard to obtain a DNN with 100% accuracy, the difference between the two formulations is crucial, since not all API output errors in Eq. 2 will cause incorrect application decisions in Eq. 1. Thus, compared to optimizing Eq. 2, optimizing Eq. 1 is more likely to focus the DNN training on reducing the critical errors for the application.

To train a DNN that optimizes Eq. 1, we need to decide if a DNN inference output $DNN(\mathbf{x})$ is a critical error or not (*i.e.,* $App(DNN(\mathbf{x})) \neq App(\widehat{API}(\mathbf{x}))$) at the end of *every* training iteration. This decision needs to be made automatically and efficiently. For example, repeatedly running the entire ML application after every training iteration would not work, as it may significantly slow down the training procedure.

[1] A careful reader might notice that the formulation in Eq. 1 also depends on the input set. Though the input set should ideally follow the same distribution of real user inputs of the application, this distribution is hard to obtain in advance and may also vary over time and across users. Instead, we focus our discussion on training the ML model to minimize Eq. 1 with an assumed input distribution. Our evaluation (§5) will test the resulting model's performance over different input distributions.

Logical steps of ChameleonAPI: To customize and deploy the DNN for an application, ChameleonAPI takes three logical steps (Figure 3). First, ChameleonAPI extracts from an application's source code a *decision-process summary* (explained shortly), a succinct representation of the application's decision process, which will be used to determine if a DNN inference error is critical (details in §3.3). Second, ChameleonAPI converts a decision-process summary to a *loss function*, which can be directly used to train a DNN (details in §3.2). This loss function only penalizes DNN outputs that lead to critical errors with respect to a given application. Finally, the loss function will be used to train a customized DNN for this particular application's ML API invocations (§3.4).

A decision-process summary is a succinct abstraction of the application that contains enough information to determine if a DNN inference output causes a critical error or not. Specifically, it includes three pieces of information (defined in §2.3):

- *Composition of target classes:* the label list or value range of each target class;
- *Decision type:* True-False, Multi-Choice, or Multi-Select;
- *Matching order:* over the target classes, API-order or App-order, if the application makes a Multi-Choice decision.

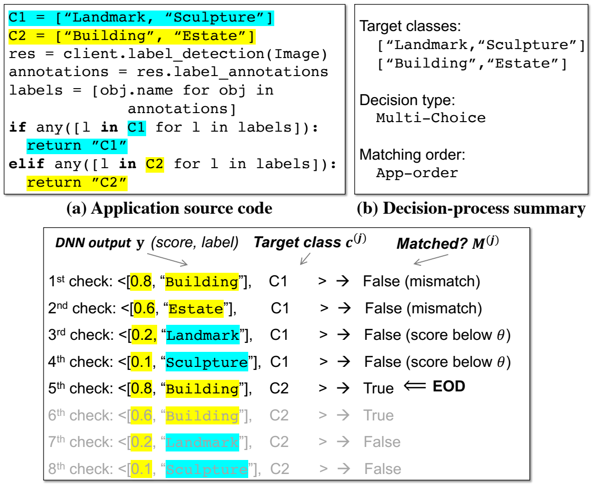
For a concrete example, the decision-process summary of the garbage classification application in Figure 2(a) contains (1) three label lists representing three target classes: `Recycle`, `Compost`, and `Donate`; (2) the Multi-Choice type of decision; and (3) the matching order of API-order.

What is changed, what is not: ChameleonAPI does *not* change the ML API or the application source code. Unlike recent work that aims to shrink the size of DNNs or speed them up [36, 37, 54], we do not change the DNN architecture (shape and input/output interface); instead, we train the DNN to minimize critical errors. That said, deploying ChameleonAPI has two requirements. First, the application developers need to run ChameleonAPI's parser script to automatically extract the decision-process summary. Second, an ML model needs to be retrained for each application, instead of serving the same model to all applications.

The remainder of this section will begin with the design of the application-specific loss function based on decision-process summary, followed by how to extract the decision-process summary from the application, and finally, how the customized ML models are used to serve ML API queries.

3.2 Application-specific loss function

Given Eq 1, ChameleonAPI trains a DNN model with a *new loss function*, which only penalizes critical errors of an application, rather than all DNN inference errors. Since decision processes vary greatly across applications (§2.4), we first explain how to conceptually capture different decision processes in a generic description, which allows us to derive the mathematical form of ChameleonAPI's loss function later.



```
C1 = ["Landmark, "Sculpture"]
C2 = ["Building", "Estate"]
res = client.label_detection(Image)
annotations = res.label_annotations
labels = [obj.name for obj in
                annotations]
if any([l in C1 for l in labels]):
  return "C1"
elif any([l in C2 for l in labels]):
  return "C2"
```

(a) Application source code

Target classes:
`["Landmark,"Sculpture"]`
`["Building","Estate"]`

Decision type:
`Multi-Choice`

Matching order:
`App-order`

(b) Decision-process summary

1st check: <[0.8, "Building"], C1 > → False (mismatch)
2nd check: <[0.6, "Estate"], C1 > → False (mismatch)
3rd check: <[0.2, "Landmark"], C1 > → False (score below θ)
4th check: <[0.1, "Sculpture"], C1 > → False (score below θ)
5th check: <[0.8, "Building"], C2 > → True ⟸ EOD
6th check: <[0.6, "Building"], C2 > → True
7th check: <[0.2, "Landmark"], C2 > → False
8th check: <[0.1, "Sculpture"], C2 > → False

(c) Generic description of the decision process
on DNN output $y = [0.1, 0.2, 0.8, 0.6]$ & score threshold $\theta = 0.3$

Figure 4: *The generic description (shown in (c)) of an application (whose source code is shown in (a) and decision-process summary in (b)) on a DNN inference output* **y**.

Generalization of decision processes: For each application in our study (§2.2), our insight is that its decision process can always be viewed as traversing a sequence of conditional *checks* until an *end-of-decision* (*EOD*) occurs:

$$1^{\text{st}} \text{ check: } < \mathbf{y}, c^{(1)} > \rightarrow M^{(1)}$$

$$\ldots$$

$$j^{\text{th}} \text{ check: } < \mathbf{y}, c^{(j)} > \rightarrow M^{(j)} \qquad \Leftarrow \textbf{EOD}$$

$$\ldots$$

where the j-th check takes as input the DNN output $\mathbf{y}$ and one of target classes $c^{(j)}$, and returns a binary $M^{(j)}$ indicating whether $\mathbf{y}^{(j)}$ *matches* the condition of $c^{(j)}$ and a binary decision whether this check happens before the EOD. The set of target classes successfully matched before the EOD will be those selected by the application.

Figure 4 shows (a) an example application, (b) the decision-process summary, and (c) the generic description for this application's decision process and a DNN output.

This generic description (*e.g.,* the traversal order of the target classes, how a match is determined in a check, and when the EOD occurs) will depend on the information in the decision-process summary and the DNN output **y**. We stress that this generic description may *not* apply to all applications, but it does apply to all applications in our study (§2.2).

Categorization of critical errors: Importantly, this generic description helps to categorize critical errors:

- *Type-1 Critical Errors:* A correct target class c is not matched before EOD, but will be so if EOD occurs later.
- *Type-2 Critical Errors:* A correct target class c is never matched, before or after the EOD.
- *Type-3 Critical Errors:* An incorrect target class c is matched before EOD.

A useful property of this categorization is that any wrong decision (a correct target class not being picked, or an incorrect target class being picked) falls in a unique category, and non-critical errors do not belong to any category. In other words, as long as the loss function penalizes the occurrences of each category, it will only capture critical errors.

ChameleonAPI's first attempt of a new loss function: To understand why it is difficult to penalize critical errors and critical errors *only*, we first consider the common practice of assigning a higher weight to the loss of a DNN output if the ground-truth of the input will lead to a selection of some target classes (*e.g.,* [26, 35, 64]). Henceforth, we refer to this basic design of loss function as ChameleonAPI$_{basic}$.

At best, ChameleonAPI$_{basic}$ might improve the DNN's *label-wise* accuracy on inputs whose ground-truth decision selects some target classes. However, as elaborated in §2.3, we also need to consider which labels belong to the same target class, the decision type, and the matching order of an application decision process in order to capture the three types of critical errors. For instance, in the garbage-classification application (Figure 1), without knowing the label lists of each target class, ChameleonAPI$_{basic}$ will give an equal penalty to a critical error of mis-classifying a `Paper` image to `Wood` and a non-critical error of mis-classifying a `Paper` image to `Shirt`. Similarly, without knowing the matching order, ChameleonAPI$_{basic}$ will equally penalize the output of [`Plastic`, `Jacket`] and [`Jacket`, `Plastic`], but only the latter leads to correct output because `Jacket` is matched first.

ChameleonAPI's loss function: ChameleonAPI leverages the categorization of critical errors to systematically derive a loss function that penalizes each type of critical error. To make it concrete, we explain ChameleonAPI's loss function of "label-based API, Multi-Choice type of decision, and App-order" (*e.g.,* Figure 4). Appendix§B will detail the loss functions of other decision processes. The loss function of such applications has three terms, each penalizing one type of critical error:

$$L(\mathbf{y}) = \overbrace{\text{Sigmoid}\left(\min\left(\max_{l\in\cup_{c<\hat{c}}G_c}\mathbf{y}[l], \max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)-\theta\right)}^{\textbf{Type-1}\text{ Critical Errors}} \qquad (3)$$

$$+\overbrace{\text{Sigmoid}\left(\theta-\max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)}^{\textbf{Type-2}\text{ Critical Errors}}+\overbrace{\sum_{c<\hat{c}}\text{Sigmoid}\left(\max_{l\in G_c}\mathbf{y}[l]-\theta\right)}^{\textbf{Type-3}\text{ Critical Errors}}$$

Here, $\mathbf{y}[l]$ denotes the score of the label l, G_c denotes the set of labels of target class c, $\hat{c}$ denotes the correct (*i.e.,* ground-truth) target class, and the sigmoid function $\text{Sigmoid}(x) = \frac{1}{1+e^x}$ will incur a higher penalty on a greater positive value.

Why does it capture the critical errors? Given this application is Multi-Choice, the EOD will occur right after the first match of a target class, *i.e.,* the first check with a c such that $\max_{l\in G_c}\mathbf{y}[l] \geq \theta$.

- A Type-1 critical error occurs, if (1) the correct target class $\hat{c}$ is matched *and* (2) it is matched after the EOD. First, the correct target class $\hat{c}$ is matched, if and only if at least one of its labels has a score above the confidence threshold, so $\max_{l\in G_{\hat{c}}}\mathbf{y}[l] \geq \theta$). Second, this match happens after the break, if and only if some target class c before $\hat{c}$ (*i.e., $c < \hat{c}$*) is matched, so $\max_{l\in G_c}\mathbf{y}[l] \geq \theta$). Put together, the first term of Eq 3 penalizes any occurrence of these conditions.
- A Type-2 critical error occurs, if no label in the correct target class $\hat{c}$ has a score high enough for $\hat{c}$ to be matched, *i.e.,* $\max_{l\in G_{\hat{c}}}\mathbf{y}[l] < \theta$, so the second term of Eq 3 penalizes any occurrence of this condition.
- A Type-3 critical error occurs, if any incorrect target class c before $\hat{c}$ (*i.e., $c < \hat{c}$*) has a label with a score high enough for c to be matched, *i.e.,* $\max_{l\in G_c}\mathbf{y}[l] - \theta$, so the third term of Eq 3 penalizes any occurrence of this condition.

To train a DNN, the loss function must be differentiable with respect to the DNN ouput $\mathbf{y}$. Eq 3 uses the max function several times. Though max is not naturally differentiable, it can be closely approximated in well-known differentiable forms provided by PyTorch's differentiable operators [56]).

3.3 Extracting applications' decision process

The current prototype of ChameleonAPI program analysis supports Python applications that make decisions based on categorical label output or floating point output of ML APIs. We first discuss how it works for ML APIs with categorical label output, like all the APIs in Table 1 except for `analyze_sentiment`. We will then discuss a variant of it that works for most use cases of `analyze_sentiment`.

Given application source code, ChameleonAPI first identifies all the invocations of ML APIs. For every invocation I in a function f, ChameleonAPI then identifies all the branches whose conditions have a data dependency upon the ML API's label output. We will refer to these branches as I-branches. If there is no such branch in f, ChameleonAPI then checks the call graph, and analyzes up to 2 levels of callers and up to 5 levels of callees of f until such a branch is identified. If no such branch is identified after this, ChameleonAPI considers the ML API invocation I to not affect application decisions and hence does not consider any optimization for it. If some I-branches are identified, ChameleonAPI records the top-level function analyzed, F, and moves on to extract the decision-process summary in following steps.

What are the target classes? ChameleonAPI figures out all the target classes and their composition in two steps.

The first step leverages symbolic execution and constraint solving to identify all the labels that belong to *any* target classes. Specifically, ChameleonAPI applies symbolic execution to function F, treating the parameters of F and the label output of I as symbolic (*i.e.,* the symbolic execution skips the ML API invocation I and directly uses I's symbolic output in the remaining execution of F)[2]. Since applications typically match only one label in API output at a time (as observed in §2.3), we set the label array returned by I to contain one element (label) and use a symbolic string to represent it. Through symbolic execution, ChameleonAPI obtains constraints for every path that involves an I-branch, solving which tells ChameleonAPI which labels need to be in the output of the ML API in order to execute each unique path, essentially all the labels that belong to any target class.

One potential concern is that a solver may only output one instead of all values that satisfy a constraint. Fortunately, the symbolic execution engine used by ChameleonAPI, NICE [31], turns Python code into an intermediate representation where each branch is in a simplest form. Take Figure 2(d) as an example, the source-code branch `if obj.name in Protein` is transformed into three branches where `obj.name` is compared with "`Hamburger`", "`Meat`", and "`Patty`" separately, allowing us to capture all three labels by solving three separate path constraints.

The second step groups these labels into target classes by comparing their respective paths: if two API output, each with one label, lead the program to follow the same execution path at the source-code level, these two labels belong to the same target class. For example, in Figure 2(d), the execution path is exactly the same when the `label_detection` API returns `["Hamburger"]`, comparing with when it returns `["Meat"]`, with all function parameters and other API output fields being the same. Consequently, we can know that label `Hamburger` and label `Meat` belong to the same target class. To figure out the path, ChameleonAPI simply executes function F using each input produced by the constraint solver and traces the source-code execution path using the Python trace module.

One final challenge is that ChameleonAPI needs to identify and exclude the path where none of the target classes are matched (*e.g.,* the "`It is others.`" path in Figure 2(a)). We achieve this by carefully setting the default solution in the constraint solver to be an empty string, which is impossible to output for any ML APIs in this paper. This way, whenever this default solution is output, ChameleonAPI knows that the corresponding path matches no target class.

What is the type of decision? When only one target class is identified, ChameleonAPI reports a True-False decision type. Otherwise, ChameleonAPI decides whether the decision type is Multi-Choice or Multi-Select by checking the source-code execution path associated with every target class label obtained above. If any execution evaluates an I-branch *after* another I-branch is already evaluated to be true, ChameleonAPI reports a Multi-Select decision type; otherwise, ChameleonAPI reports a Multi-Choice decision type.

What is the matching order over the target classes? To tell whether a Multi-Choice decision is made through API-Order like in Figure 1 or App-Order like in Figure 2(a), ChameleonAPI first identifies all the `for` loops that iterate through the label array output by the ML API and have control-dependency with I-branches, *e.g.,* the `for l in labels` in Figure 2(a) and the `for obj in response.label_annotations` in Figure 1.

ChameleonAPI then checks how many such output-iterating loops there are. If there is only one and this loop is not inside another loop, like that in Figure 1, ChameleonAPI considers the matching order to be API-Order, as the application only iterates through each output label once, with the matching order determined by the output array arranged by the ML API. Otherwise, ChameleonAPI considers the matching order to be App-Order. This is the case for the example shown in Figure 2(a), where three output-iterating loops are identified, each of which matches with one target class in an order determined by the application: the `Landmark` target class, followed by the `Building`, and finally the `Person`.

How to handle floating-point output of ML APIs? Recall in §2.3 that some ML APIs, *e.g.,*`analyze_sentiment`, have floating-point output and the application defines several value ranges to put each floating-point output into one category. To handle this type of API, ChameleonAPI needs to identify the value range of each target class, which is not supported by NICE and other popular constraint solvers. Fortunately, many applications directly compare API output with constant values in I-branches, giving ChameleonAPI a chance to infer the value range. For these applications, ChameleonAPI first extracts those constant values that are compared with API output in I-branches, *e.g.,* 0.3 and 0.6 in Figure 2(b). ChameleonAPI then forms tentative value ranges using these numbers, like -1 – 0.3, 0.3 – 0.6, and 0.6 – 1 for Figure 2(b) (-1 and 1 are the smallest and biggest possible `score` output of `analyze_sentiment` based on the API manual). To confirm these value ranges and figure out the boundary situation, ChameleonAPI then executes function F with all the boundary values, as well as some values in the middle of each range. By comparing which values lead to the same execution path, ChameleonAPI finalizes the value ranges. For the example in Figure 2(b), after executing with `score` set to -0.35, 0.3, 0.45, 0.6, and 0.8, ChameleonAPI settles down on the final value ranges to be: (-1,0.3), [0.3,0.6), and [0.6,1).

Limitation The static analysis in ChameleonAPI does not handle the iterated object of while loops, unfolded loops, and recursive functions. For complexity concerns, ChameleonAPI only checks caller and callee functions with limited levels, and hence may miss some I-branches far away from the API invocation. ChameleonAPI's ability of identifying target classes is limited by the constraint solver. ChameleonAPI assumes different source-code paths correspond to different target classes, which in theory could be wrong if the application behaves exactly the same under different execution paths.

[2] Recall that an API output contains several fields not used to influence control flow in any applications. We set them with pre-defined dummy values.

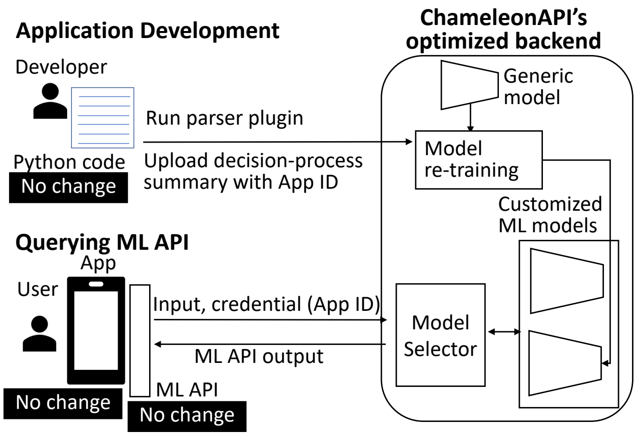


Figure 5: *Workflow of ChameleonAPI.*

3.4 Putting them together

We put these components together into a ML-as-a-Service workflow shown in Figure 5.

First, when an application (A) is developed or updated, the developers run a parser (described in §3.3) provided by ChameleonAPI on A's source code to extract the decision-process summary for A. The developers can then upload the decision-process summary to ChameleonAPI's backend together with a unique application ID[3] (which will later be used to identify queries from the same application).

ChameleonAPI's backend then uses the received decision-process summary to construct a new application-specific loss function (described in §3.2). When a DNN is trained using the new loss function, its inference results will lead to fewer critical errors (*i.e.,* incorrect application decisions) for application A. In our prototype, ChameleonAPI uses the new loss function to re-train an off-the-shelf pre-trained DNN, a common practice to save training time (see §5 for quantification). The DNN re-training uses an application-specific dataset sampled from the dataset used by the pre-trained generic DNN (see Table 2 and §4), so that each target/non-target class is selected by ground-truth decisions of the same number of inputs.

Finally, ChameleonAPI backend maintains a set of DNN models, each customized for an application and keyed by the application ID. When application A invokes an ML API at run time, the ChameleonAPI backend will use the application ID associated with the API query to identify the DNN model customized for A, run the DNN on the input, and return the inference result of the selected model to the application.

Note that, ChameleonAPI can also be used to customize ML models that run locally behind the ML APIs, instead of those in the cloud through ML service providers. In this case, developers run the ChameleonAPI parser on their application and save the parser's result into a local file. This local file will then be consumed to help re-train an off-the-shelf DNN into a customized DNN to serve the application.

	Dataset	Generic model
Image Classification	OpenImages [44]	TResNet-L [6]
Object Detection	COCO [14]	Faster-RCNN [58]
Sentiment Analysis	Amazon review [39]	BERT [18]
Text Classification	Yahoo [30]	BERT [18]
Entity Recognition	conll2003 [63]	BERT [18]

Table 2: *The ML APIs and datasets in evaluation.*

4 Implementation

Extractor of decision-process summary: The current prototype of ChameleonAPI is implemented for Python applications that use Google or Amazon ML APIs. It takes as input the application source code and returns as output the decision-process summary in the JSON format. It uses NICE symbolic execution engine [31] and CVC5 constraint solver [5] to identify target classes, and uses Python static analysis framework Pyan [47] and Jedi [24] to identify the decision type and the matching order. Particularly, it identifies the object that is iterated through by a `for`-loop through the `iter` expression in each for-loop header, which is used to distinguish Multi-Choice and Multi-Select decisions and the matching order.

ML re-training: The re-training module is implemented in PyTorch v1.10 and CUDA 11.1. It uses a decision-process summary to construct a new loss function (see §3.2), and then replaces the builtin loss function in Pytorch with the new loss function, and uses the common forward and backward propagation procedure to re-train an off-the-shelf pre-trained DNN model (explained next).

Generic models: Without access to the models and the training data used by commercial ML services, we use open-sourced pre-trained DNNs and their training datasets as a proxy, which are summarized in Table 2. These DNNs are trained on the "training" portion of their respective datasets. They are trained to achieve good accuracy over a wide range of labels, and we have confirmed that their accuracies in terms of application decisions are similar to the real ML APIs (§5.2).

Training data: We make sure that the labels included in these datasets cover the labels used in the decision processes of the applications in our study. An exception is text classification: to our best knowledge, there is no open-source dataset that covers the classes in Google's text classification API. Instead, we use the Yahoo Question topic classification dataset [30], whose classes are similar to those used in the applications.

Instead of training DNNs on all training data, most of which do not match any target classes of an application, we create a downsampled training set for ChameleonAPI and ChameleonAPI$_{basic}$. For each application, we randomly sample (without replacement) its training data such that each target class and the non-target class (not matching any target

[3] In many MLaaS offerings [2, 20], a connection between the application and the MLaaS backend is commonly created before the application issues any queries. Existing MLaaS already allows applications to specify the application ID via the connection between the application and backend.

class) is the correct decision for the same number of training inputs, which depending on applications, ranges from 12K to 40K. With such training set, ChameleonAPI$_{basic}$ will be equivalently implemented by training on the downsampled training set using the conventional loss function (*i.e.,* cross-entropy loss for classification tasks). Moreover, the downsampled training set significantly speedups DNN re-training (§5.2).

5 Evaluation

Our evaluation aims to answer following questions: How much can ChameleonAPI reduce incorrect application decisions? How long does it take ChameleonAPI to customize DNN models for applications? and Why does ChameleonAPI reduce incorrect application decisions where ChameleonAPI$_{basic}$ falls short?

5.1 Setup

Applications: We have applied ChameleonAPI on all the 77 applications summarized in Table 1. Due to space constraints, our discussion below focuses on the 57 applications that involve three popular ML tasks, image-classification, object-detection, and text-classification, and omits the remaining 20 applications that involve sentiment analysis and entity recognition. The results of the latter show similar trends of advantage from ChameleonAPI and are available in Appendix §D.

Metrics: For each scheme (explained shortly) and each application, we calculate the *incorrect decision rate* (*IDR*): the fraction of testing inputs whose application decisions do not match the correct application decisions (*i.e.,* decisions based on human-annotated ground truth).

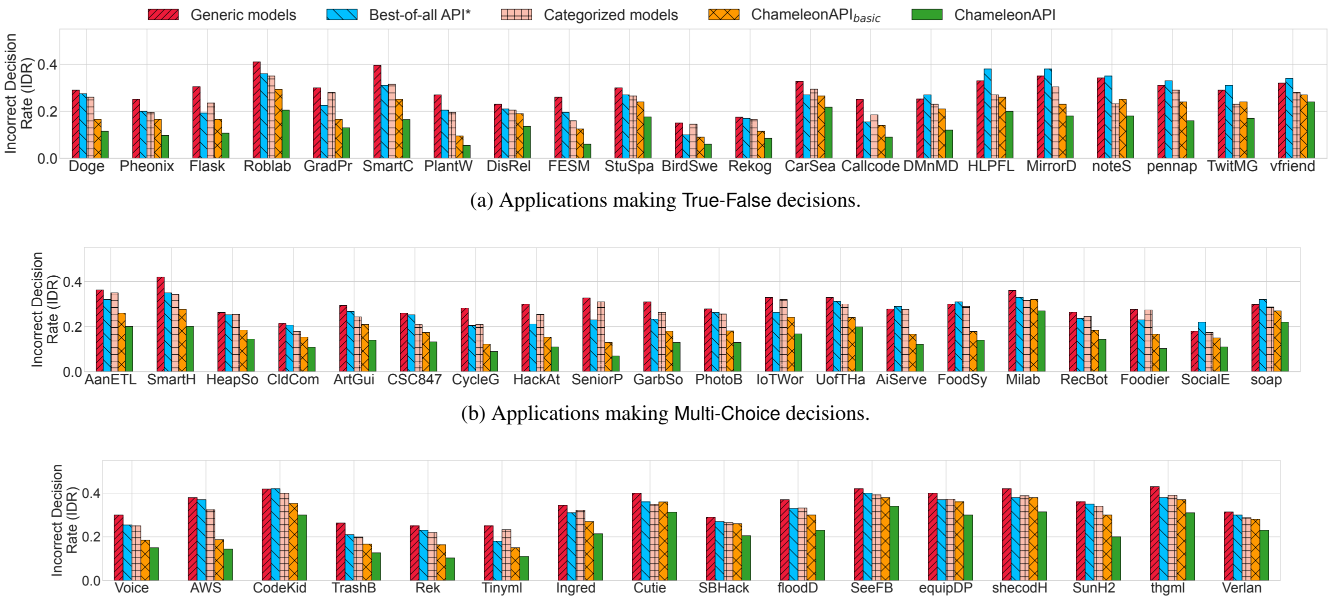
Schemes: We compare the results of these schemes:

- *Various commercial ML APIs*: the results returned by ML APIs of three service providers (Google [20], Amazon [2] and Microsoft [51]).
- *Best-of-all API**: a hypothetical method that queries ML APIs from those three service providers on each input and picks the best output based on the classic definitions of accuracy: label-wise recall for classification tasks and mean-square-error of floating-point output for sentiment analysis. This serves as an idealized reference of recent work [11,12], which tries to select the best API output with high label-wise accuracy.
- *Generic models*: the open-sourced generic model based on which the next three schemes are re-trained. They serve as a reference without customization and achieve similar accuracy as commercial APIs. Their details are explained in Section 4.
- *Categorized models*: This scheme pre-trains a number of specialized models. Each specialized model replaces the last layer of the generic model so that it outputs the confidence scores for a smaller number of labels representing a common category (e.g., "dog", "animal", "person" and a few other labels represent the "natural object" category), and is fine tuned from the generic model accordingly. A simple parser checks which labels are used by an application. If all the labels belong to one category, the corresponding model specialized for this category is used to serve API calls from this application. If the labels belong to multiple categories, multiple specialized models will be used, which we will explain more later. We set up 35 categories for image classification and 7 categories for object detection based on the Wikidata knowledge graph [68], as well as 15 categories for text classification based on the inherent hierarchy in Google text-classification output. More details of how we have designed these categories are available in the Appendix §C.

  Note that, we have designed this scheme to represent a middle-point in the design space between the generic model and the ChameleonAPI approach: on one hand, this scheme offers some application customization, but not as much as ChameleonAPI (e.g., which labels belong to the same target class, what is the decision process, and what is the matching order used by the application are all ignored); on the other hand, this scheme requires a simpler parser compared to ChameleonAPI.
- *ChameleonAPI$_{basic}$*: the model is re-trained with ChameleonAPI's training data, which concentrates on labels used by the application, but with the conventional loss function. Like Categorized models, this scheme only needs a simple parser that extracts which labels are used by the application, and does not make use of other application information that ChameleonAPI uses. Unlike Categorized models, this scheme prepares a customized model for each application, instead of relying on a small number of categorized models.
- *ChameleonAPI* (our solution): the model re-trained with our training data and loss function.

Testing data: For the same application, all schemes are tested against the same testing input set. The testing set of an application is randomly sampled from the "testing" portion of the dataset associated with the application's generic model (Table 2). We make sure that *no* testing input appears in the training data. Like the creation of training data of ChameleonAPI (§4), by default, we randomly sample the testing data such that each target class and the non-target class (not matching any target class) appear as the correct decision for the same number of testing inputs, which ranges from 1.2K to 4K. This is similar to the testing sets used in related work on ML API (*e.g.,* [11,36,37,66]). Such data downsampling is commonly used in ML [19,46]. Other than Figure 9, we will use this as the default testing dataset.

Hardware setting: We evaluate ChameleonAPI and other approaches on a GeForce RTX 3080 GPU, and an Intel(R) Xeon(R) E5-2667 v4 CPU, with 62GB memory.



(a) Applications making True-False decisions.

(b) Applications making Multi-Choice decisions.

(c) Applications making Multi-Select decisions.

Figure 6: *ChameleonAPI reduces the incorrect decision rate (IDR) on the 57 applications that use Google's or Amazon's image-classification, text-classification, and object-detection APIs.*

	True-False	Multi-Choice	Multi-Select
Google API	0.29	0.32	0.35
Microsoft API	0.30	0.33	0.32
Amazon API	0.31	0.33	0.36
Best-of-all API*	0.26	0.27	0.31
Generic models	0.29	0.30	0.34
Categorized models	0.24	0.27	0.31
ChameleonAPI$_{basic}$	0.19	0.22	0.27
ChameleonAPI	**0.13**	**0.16**	**0.21**

Table 3: *Average incorrect decision rate (IDR) among apps that make different types of decisions. The lower the better. The top half represents commercial APIs and their idealistic combinations; the bottom half represents open-source models.*

	Single-category	Multi-category
Generic models	0.32	0.28
Categorized models	0.28	0.27
ChameleonAPI$_{basic}$	0.24	0.18
ChameleonAPI	0.17	0.14

Table 4: *Average IDR among single-category and multi-category applications. The lower the better.*

5.2 Results

Overall gains: Measured by the average incorrect decision rate (IDR) across all applications, the most accurate scheme is ChameleonAPI, with an IDR of 0.16, and the least accurate scheme is Generic models, with an IDR of 0.31. In other words, ChameleonAPI successfully reduces the number of incorrect decisions of its baseline model by almost 50%. ChameleonAPI$_{basic}$ (0.22), Categorized models (0.28), and Best-of-all API* (0.28) have IDR rates in between.

The advantage of ChameleonAPI, and even ChameleonAPI$_{basic}$, over the other schemes is consistent across all three types of applications that make different types of decisions, as shown in Table 3. In fact, ChameleonAPI offers the highest accuracy by a clear margin for every single application in our evaluation, as shown in Figure 6.

To better compare the ChameleonAPI approach with Categorized models, we divide the 57 applications into two types: (1) 39 single-category applications — each application uses labels that belong to one category and hence can benefit from one specialized model in the Categorized models scheme; (2) 18 multi-category applications — each application uses labels that belong to multiple categories. For these applications, the Categorized models scheme feeds the API input to multiple specialized models and combines these models' output to form the API output. As shown in Table 4, the Categorized models scheme does offer improvement from Generic models by considering which labels belong to an application's target classes, particularly for single-category applications. However, both ChameleonAPI and ChameleonAPI$_{basic}$ perform better than Categorized models for both single-category and

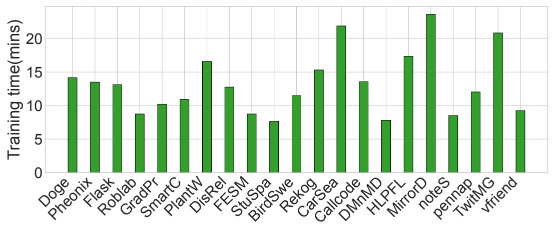


Figure 7: *Re-training time for applications in Figure 6(a).*

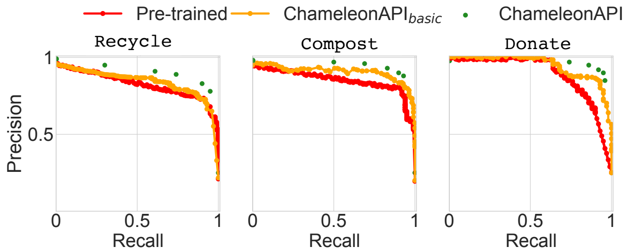


Figure 8: *Precision-recall trade-off for* `HeapsortCypher`*.*

multi-category applications—the per-application customization in ChameleonAPI and ChameleonAPI$_{basic}$ has paid off.

The above advantage of ChameleonAPI over ChameleonAPI$_{basic}$ and Categorized models shows that the static analysis used in ChameleonAPI to extract not only what labels are used by the application, but also which labels belong to the same target class, the decision type, and the matching order, as described in Section 3.3, is worthwhile.

Cost of obtaining customized models: The customization effort of ChameleonAPI includes two parts (1) extracting the decision-process summary from application source code, and (2) re-training the ML model. The first part takes a few seconds: on an Intel(R) Xeon(R) E5-2667 v4 CPU machine, our parser extracts the decision-process summary from every benchmark application within 10 seconds.

The second part takes a few minutes, much faster than training a neural network from scratch. As shown in Figure 7, re-training DNNs for the 21 applications in Figure 6(a) on a single RTX 3080 GPU takes 8 to 24 minutes. Focusing on a small portion of all possible labels (§2.4), ChameleonAPI fine-tunes pre-trained models using much less training data than the generic models and thus needs fewer iterations to converge.

Considering that a V100 GPU with similar processing GFLOPS as our RTX 3080 GPU only costs $2.38 per hour on Google Cloud [21], re-training an ML model for one application costs less than $1.

Cost of hosting customized models: For cloud providers, ChameleonAPI would incur a higher hosting cost than traditional ML APIs by serving a customized DNN for every application instead of a generic DNN for all applications.

The extra cost includes more disk space to store customized neural network models. For example, each image-classification model in ChameleonAPI uses 115 MB of disk space. So, for n applications, $115 \cdot n$ MB of disk space may be needed to store ChameleonAPI customized models.

The extra cost also involves more GPU resources. A naive design of using one GPU to exclusively serve requests to one customized neural network model will likely lead to under-utilization of GPU resources. To serve different applications' customized models on one GPU, we need to pay attention to memory working set and performance isolation issues. In our experiments on an RTX 3080 GPU, loading an image-classification model from CPU to GPU RAM takes 18 to 40 ms (inference itself takes 10 to 35 ms with a batch size of 1). Fortunately, modern GPU has sufficiently large RAMs to host several requests to different customized models simultaneously: in our experiments, the peak memory consumption of one inference request is less than 2GB. Furthermore, the majority of the model inference memory consumption comes from intermediate states, instead of the model itself. Consequently, the memory consumption of multiple inference requests on different models is similar to that on the same model.

Of course, ChameleonAPI can take advantage of recent proposals to improve GPU sharing [15, 55, 71, 73] as well as to reduce the footprint of serving multiple DNNs [33]. These techniques could be advantageously employed by ChameleonAPI to determine the optimal degree of sharing among customized DNNs, and we leave them to future work.

Finally, there is also the extra cost of needing more complex software to manage the DNN serving. For instance, ChameleonAPI needs to dynamically route each request to a GPU that serves the DNN of the application (see §3.4).

Precision-recall tradeoffs: Traditionally, for a trained ML model, it is common to vary the confidence-score thresholds in order to find the best precision-recall tradeoff of a trained model. Thus, it is important that ChameleonAPI also achieves better precision-recall tradeoffs. Figure 8 shows the precision-recall results in each target class of a particular application, by varying the detection threshold θ (defined in §3.2) of two baselines (real APIs are excluded, because we cannot change their thresholds and their IDR is not as low as ChameleonAPI$_{basic}$). ChameleonAPI's tradeoffs are better than both baselines (and we observe similar results in other applications). Note that since ChameleonAPI's loss function uses an assumed θ, we do not vary the θ when testing it; instead, we re-train five DNNs of ChameleonAPI, each with a different θ and test them with their respective thresholds.

Understanding the improvement: ChameleonAPI's unique advantage is that it factors in the decision process of an application, including not only the target classes but also the decision type and the matching order. Next, we use two case studies to further reveal the underlying tradeoffs made by ChameleonAPI to achieve its improvement on application-decision accuracy.

First, ChameleonAPI reduces errors related to different

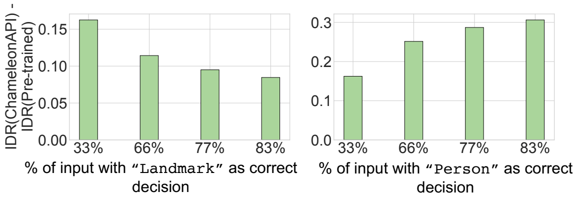


Figure 9: *How the accuracy advantage of ChameleonAPI changes with different input distribution.*

target classes differently depending on their different roles in the application decision process. This effect is particularly striking in Multi-Choice applications with the matching order of App-Order, where the first target class is always matched against API output. Thus, when the correct target class is not the first one, falsely including a label that belongs to the first target class will more likely be a critical error than other mis-classifications, because it will block the match of other target classes. To illustrate this, we consider the Multi-Choice application of `Aander-ETL`. We increase the percentage of testing inputs whose correct action is the first target class or the last target class. Figure 9 shows that increasing the portion of inputs of the last target class (`Person`) generally leads to bigger gains of ChameleonAPI, whereas increasing the portion of the first target class (`Landmark`) does the opposite. This shows the application itself already has good tolerance to mis-classification on inputs that belong to the first target class, but not to mis-classification on the inputs that belong to later target classes, which is exactly where ChameleonAPI can help.

Second, recall from §3.2 that our loss function helps to minimize critical errors, even at the cost of missing labels that do not affect application decisions (*i.e.,* non-critical errors). To show this, we define *label error rate* on an image as the fraction of the image's ground-truth labels that are missed by the DNN output (a label list). We consider `IoTWor` (explained in Table 5), which similar to `Anander-ETL` makes Multi-Choice decisions with App-Order matching order. The average label missing rate of ChameleonAPI on our testing images is 0.21, which is slightly higher than ChameleonAPI$_{basic}$'s 0.18. This means ChameleonAPI makes more label-level mistakes than ChameleonAPI$_{basic}$. However, our IDR (0.17) is 44% lower than ChameleonAPI$_{basic}$, which means ChameleonAPI makes far fewer critical errors.

6 Related Work

Due to space constraints, we discuss related papers that have not been discussed earlier in the paper.

Optimizing storage and throughput of DNN serving: Various techniques have been proposed to optimize the delay, throughput, and storage of ML models via model distillation [40, 54, 61], pruning [26] or cascading [4, 7]. This line of work explores a different design space than ChameleonAPI: they design ML models with higher inference speed or smaller model size with minimum loss in accuracy. ChameleonAPI focuses on re-training existing ML models such that the rate of incorrect decisions of a given application is reduced.

Application-side optimization: Recent work also proposes to change the applications to better leverage existing ML APIs. One line of work [11, 12, 69] invokes ML APIs from different service providers to achieve high accuracy within a query cost budget. Another line of work aims to eliminate misuse of ML APIs in applications [65, 66]. They require changes to the application source code (*e.g.,* changing the API input preparation, switching from image-classification API to object-detectin API, etc.). They are complementary to our work, because we customize the ML-API backend DNN and do not require changes on the application's source code.

Measurement work on MLaaS: For their rising popularity, ML-as-a-Service platforms have also attracted many measurement studies to understand accuracy [10], performance [70], robustness [28], and fairness [41]. However, they have so far not taken in account the ML applications that use ML APIs, and is thus different from our empirical study of ML applications in §2. Previous work that studies ML applications [65] did not look into the decision making process and how ML API errors might affect different applications differently.

Finally, a myriad of techniques have been studied to better manage and schedule GPU resources in ML training/serving systems (*e.g.,* [13, 16, 17, 22, 25, 27, 32, 42, 45, 53, 59, 60, 67, 72, 74]). They aim for different goals than ChameleonAPI, but these techniques can be used to help ChameleonAPI train and serve the application-specific ML models.

7 Conclusion

ML APIs are popular for its accessibility to application developers who do not have the expertise to design and train their own ML models. In this paper, we study how the generic ML models behind ML APIs might affect different applications' control-flow decisions in different ways, and how some ML API output errors may or may not be critical due to the application decision making logic. Guided by this study, we have designed ChameleonAPI that offers both the accuracy advantage of a custom ML model and the accessibility of the traditional ML API.

8 Acknowledgement

We thank all the reviewers for their insightful feedback and suggestions. The project is funded by NSF CNS-2146496, CNS-2131826, CNS-2313190, CNS-1901466, CNS-1956180, CCF-2119184, UChicago CERES Center, and Marian and Stuart Rice Research Award. The project is also supported by Chameleon Projects [38].

References

[1] Aander-ETL. A smart album application. https://github.com/Grusinator/Aander-ETL.

[2] Amazon. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services, 2022.

[3] Amazon. Amazon rekognition: detecting labels. Online document https://docs.aws.amazon.com/rekognition/latest/dg/labels.html, 2022.

[4] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1466–1477. IEEE, 2019.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Emanuel Ben-Baruch, Tal Ridnik, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. Asymmetric loss for multi-label classification. *arXiv preprint arXiv:2009.14119*, 2020.

[7] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Thia: Accelerating video analytics using early inference and fine-grained query planning. *arXiv preprint arXiv:2102.08481*, 2021.

[8] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 559–572, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 8(18):13849–13875, 2021.

[10] Lingjiao Chen, Tracy Cai, Matei Zaharia, and James Zou. Did the model change? efficiently assessing machine learning api shifts. *arXiv preprint arXiv:2107.14203*, 2021.

[11] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalMCT: Efficient online ML API selection for multi-label classification tasks, 2022.

[12] Lingjiao Chen, Matei Zaharia, and James Y Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. *Advances in neural information processing systems*, 33:10685–10696, 2020.

[13] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

[14] COCO. Coco dataset. https://cocodataset.org/#home, 2017.

[15] NVIDIA Corporation. Nvidia multi-process service (mps) documentation, 2023. Accessed: 2023-07-12.

[16] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Amr ElRafey and Janusz Wojtusiak. Recent advances in scaling-down sampling methods in machine learning. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(6):e1414, 2017.

[20] Google. Google cloud ai. Online document https://cloud.google.com/products/ai, 2022.

[21] Google. Compute engine pricing. Online document https://cloud.google.com/compute/all-pricing, 2023.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[23] Armin Haller, Axel Polleres, Daniil Dobriy, Nicolas Ferranti, and Sergio J Rodríguez Méndez. An analysis of links in wikidata. In *European Semantic Web Conference*, pages 21–38. Springer, 2022.

[24] Dave Halter. Jedi: an awesome auto-completion, static analysis and refactoring library for python. Online document https://jedi.readthedocs.io.

[25] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[26] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Hossein Hosseini, Baicen Xiao, and Radha Poovendran. Google's cloud vision api is not robust to noise. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 101–105. IEEE, 2017.

[29] Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li, and Junwei Cao. Edge computing with artificial intelligence: A machine learning perspective. *ACM Comput. Surv.*, aug 2022. Just Accepted.

[30] Zhang Huangzhao. Yahoo_question_answer. https://github.com/LC-John/Yahoo-Answers-Topic-Classification-Dataset.git, 2018.

[31] M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

[32] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 250–263, 2019.

[33] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association.

[34] Philipp Jund, Andreas Eitel, Nichola Abdo, and Wolfram Burgard. Optimization beyond the convolution: Generalizing spatial relations with end-to-end metric learning. *CoRR*, abs/1707.00893, 2017.

[35] Heechul Jung, Sihaeng Lee, Junho Yim, Sunjeong Park, and Junmo Kim. Joint fine-tuning in deep neural networks for facial expression recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 2983–2991, 2015.

[36] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018.

[37] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529*, 2017.

[38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233. USENIX Association, July 2020.

[39] Phillip Keung, Yichao Lu, György Szarvas, and Noah A. Smith. The multilingual amazon reviews corpus. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020.

[40] Hakbin Kim and Dong-Wan Choi. Pool of experts: Real-time querying specialized knowledge in massive neural networks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2244–2252, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Allison Koenecke, Andrew Nam, Emily Lake, Joe Nudell, Minnie Quartey, Zion Mengesha, Connor Toups, John R Rickford, Dan Jurafsky, and Sharad Goel. Racial disparities in automated speech recognition. *Proceedings of the National Academy of Sciences*, 117(14):7684–7689, 2020.

[42] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: Erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 30–46, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Nick Koudas, Raymond Li, and Ioannis Xarchakos. Video monitoring queries. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[44] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

[45] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018.

[46] Shigang Liu, Jun Zhang, Yang Xiang, Wanlei Zhou, and Dongxi Xiang. A study of data pre-processing techniques for imbalanced biomedical data classification. *International Journal of Bioinformatics Research and Applications*, 16(3):290–318, 2020.

[47] David Marby and Nijiko Yonskai. Pyan3: Offline call graph generator for python 3. https://github.com/davidfraser/pyan.

[48] Petru Martincu. Cloud-computing. https://github.com/Martincu-Petru/Cloud-Computing, 2020.

[49] Chu Matthew. heapsortcypher. https://github.com/matthew-chu/heapsortcypher.git, 2019.

[50] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges. *IEEE Internet of Things Journal*, 9(17):15435–15459, 2022.

[51] Microsoft. Microsoft azure cognitive services. Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2022.

[52] Microsoft. Microsoft azure image tagging. Online document https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-tagging-images, 2022.

[53] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[54] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3573–3582, 2019.

[55] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for Memory-Efficient, Real-Time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, Boston, MA, April 2023. USENIX Association.

[56] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[57] Plant-Watcher. A plant management application. https://github.com/siwasu17/plant-watcher.

[58] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[59] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[60] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[61] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3646–3654, 2017.

[62] The-Coding-Kid. A smart album application. https://github.com/The-Coding-Kid/888hacks-flask.

[63] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003.

[64] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

[65] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Are machine learning cloud apis used correctly? In *43th International Conference on Software Engineering (ICSE'21)*, 2021.

[66] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *44th International Conference on Software Engineering (ICSE'22)*, 2022.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[68] Wikidata. A free and open knowledge base. Online document https://www.wikidata.org/, 2022.

[69] Shuzhao Xie, Yuan Xue, Yifei Zhu, and Zhi Wang. Cost effective mlaas federation: A combinatorial reinforcement learning approach. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2022.

[70] Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference*, pages 384–397, 2017.

[71] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[72] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[73] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[74] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. {DeepCPU}: Serving {RNN-based} deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, 2018.

Appendix A Applications

Table 5: The statistics of 77 applications in empirical study. (Multi-Choice-* refer to Multi-Choice (*-Order).)

Application name (Link to Github repo)	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
Image Multi-Label Classification (Google `label_detection`, AWS `detect_labels`)			
2019-iot-ai-workshop	Multi-Choice-App	**2** (7, 2)	[Capuchin monkey, ...]; [Wildlife biologist, ...]
Aander-ETL	Multi-Choice-App	**3** (9, 6, 5)	[Landmark, Sculpture, ...]; [Building, Estate, ...]; [Human, ...]
ArtGuide	Multi-Choice-API	**2** (6, 3)	[Painting, Picture frame, ...]; [Building, Architecture, ...]
AWS_CloudComputing	Multi-Select	**2** (1, 1)	[Hot dog]; [Food]
DoorWatch	True-False	**1** (6)	[Clothing, Person, Human, Furniture, Child, Man]
AWSRekognition	Multi-Select	**2** (3, 3)	[Person, People, Human]; [Art, Drawing, Sketch]
GraduateProject	True-False	**1** (5)	[Orator, Professor, Projection Screen, ...]
Voice-Assistant	Multi-Select	**3** (5, 3, 1)	[Highway, Lane, ...]; [Car, ...]; [Classroom]
callforcode	True-False	**1** (5)	[Water, Waste, Bottle, Plastic, Pollution]
Car-Image-search	True-False	**1** (6)	[Sedan, Mini SUV, Coupe utility, Truck, Van, Convertible]
cloudComputing_project2	Multi-Choice-API	**3** (1, 3, 1)	[Person]; [Dog, Cat, Mammal]; [Flower]
CSC847_GAE_Proj2	Multi-Choice-API	**3** (2, 2, 1)	[Mammal, Livestock]; [Human, People]; [Flower]
cutiehack	Multi-Select	**2** (1, 3)	[Banana]; [Lemon, Citrus fruit, Apple]
CycleGAN-tensorflow_pixie	Multi-Choice-API	**3** (1, 6, 4)	[Food]; [Girl, Boy, Man, ...]; [Room, Living room, House, ...]
DisasterRelief	True-False	**1** (8)	[Hurricane, Flood, Tornado, Landslide, Earthquake, Volcano, ...]
Dogecoin_musk	True-False	**1** (4)	[Dog, Mammal, Carnivore, Wolf]
flaskAPI	True-False	**1** (3)	[Food, Recipe, Ingredient]
food-assessment-system	Multi-Choice-API	**5** (35, 22, 54, 4, 6)	[Dessert, ...]; [Grilling, ...]; [Strawberries, ...]; [Cigarette, ...]; ...
Foodier	Multi-Select	**2** (13, 1)	[Building, Logo, Menu, Person, Vehicle, People, ...]; [Food]
Hack-At-Home-II	Multi-Choice-API	**2** (3, 3)	[Food, Junk food, Plastic]; [Drinkware, Wood, Metal]
HeapSortCypher	Multi-Choice-API	**3** (8, 5, 11)	[Food, Food grain, ...]; [Clothing, Shirt, ...]; [Paper bag, ...]
IngredientPrediction	Multi-Select	**3** (1, 1, 1)	[Spaghetti]; [Bean]; [Naan]
FESMKMITL	True-False	**1** (1)	[Face]
milab	Multi-Choice-App	**3** (1, 1, 1)	[Sign]; [Nature]; [Car]
BirdSwe	Multi-Choice-API	**1** (5)	[Smoke, Bird, ...]
ai-server-proto	Multi-Choice-API	**3** (3, 14)	[Eye, Eyeball, Eyes]; [Landmark, Sculpture, Monument, ...]
Pheonix	True-False	**1** (1)	[Fire]
photo_book	Multi-Choice-API	**3** (10, 10, 2)	[Mammal, Bird, Insect, ...]; [Skin, Lip, ...]; [Flower, Plant]
Plant-Watcher	True-False	**1** (5)	[Plant, Flowerpot, Houseplant, Bonsai, Wood]
RecBot	Multi-Choice-App	**2** (11, 8)	[Tin, Paper, Magazine, Carton, ...]; [Food, Bread, Pizza, ...]
roblab-hslu	True-False	**1** (7)	[Raincoat, Coat, Jacket, T-shirt, Trousers, Jeans, Shorts]
senior-project	Multi-Choice-API	**3** (2, 3, 1)	[Landscape, Landmark]; [Self-portrait, Portrait, ...]; [Flower]
smart-can	True-False	**1** (9)	[Paper, Bottle, Plastic, Container, Tin can, Glass, ...]
smart-trash-bin	Multi-Choice-API	**2** (14, 5)	[Aviator sunglass, Beer glass, ...]; [Plastic arts, ...]
smartHamper	Multi-Choice-API	**3** (7, 4, 3)	[Shirt, T-shirt, ...]; [Trousers, Denim, ...]; [Brand, Text, ...]
StudySpaceAvailability	True-False	**1** (4)	[Hardware, Power Drill, Drill, Electronics]
The-Coding-Kid	Multi-Select	**6** (9, 6, 9, 3, 6, 3)	[Noodle, ...]; [Meat, ...]; [Produce, ...]; [Fruit, ...]; [Milk, ...]; ...
Tinyml	Multi-Select	**3** (4, 6, 5)	[Car, Truck, ...]; [Gun, Weapon Violence, ...]; [Cat, Dog, ...]
UofTHacksBackend	Multi-Choice-API	**4** (3, 3, 7, 4)	[T-shirt, ...]; [Outerwear, ...]; [Pants, ...]; [Footwear, ...]
garbage-sort	Multi-Choice-API	**2** (1, 20)	[Food];[Metal, ...];
Image Object Detection (Google `object_localization`)			
equipment-detection-poc	Multi-Select	**1** (1)	[Shoe]
flood_depths	Multi-Select	**1** (5)	[Car, Van, Truck, Boat, Toy vehicle]
SBHacks2021	Multi-Select	**1** (1)	[Person]
SeeFarBeyond	Multi-Select	**1** (2)	[Spoon, Coin]

(To be continued)

Table 5: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
shecodes-hack	Multi-Select	**1** (2)	[Dress, Top]
SunHacks2019	Multi-Select	**1** (3)	[Person, Chair, Table]
thgml	Multi-Select	**1** (7)	[Pizza, Food, Sushi, Baked goods, Snack, Cake, Dessert]
Verlan	Multi-Select	**1** (2)	[Dog, Animal]
Text Sentiment Classification (Google `sentiment_detection`)			
animal-analysis	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.5, 1]; [0, 0.5]; [-0.5, 0]; [-1, -0.5]
calhacksv2	Multi-Choice-API	**6** (1, 1, 1, 1, 1, 1)	[0.5, 1]; [0.5, 1]; [0.1, 0.5]; [-0.1, 0.1]; [-0.5, -0.1]; [-1, -0.5]
carbon_hack_sentiment	Multi-Choice-API	**3** (1, 1, 1)	[0.3333, 1]; [-0.3333, 0.3333]; [-1, -0.3333]
FoodDelivery	Multi-Choice-API	**3** (1, 1, 1)	[0.6, 1]; [0.3, 0.6]; [-1, 0.3]
devfest	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.6, 1]; [0.4, 0.6]; [0.2, 0.4]; [-1, 0.2]
EC601_twitter_keyword	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [0.25, 1]
ElectionSentimentAnalysis	Multi-Choice-API	**3** (1, 1, 1)	[0.05, 1]; [0, 0.05]; [-1, 0]
Hapi	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
JournalBot	Multi-Choice-API	**3** (1, 1, 1)	[0.5, 1]; [0, 0.5]; [-1, 0]
Mind_Reading_Journal	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.15, 1]; [0.1, 0.15]; [-0.15, 0.1]; [-1, -0.15]
Sarcatchtic-MakeSPP19	Multi-Choice-API	**2** (1, 1)	[-0.5, 1]; [-1, -0.5]
stockmine	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
Tone	Multi-Choice-API	**3** (1, 1, 1)	[-1, -0.5]; [-0.5, 0.5]; [0.5, 1]
UOttaHack_2019	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [-1, -0.25]
Text Entity Detection (Google `entity_detection`)			
GeoScholar	True-False	**1** (1)	[LOC]
HackThe6ix	Multi-Choice-API	**7** (1, 1, 1, 1, 1, 1, 1)	[PERSON]; [LOC]; [ADD]; [NUM]; [DATE]; [PRICE]; [ORG]
Klassroom	Multi-Choice-API	**2** (2, 2)	[PERSON, PROPER]; [LOC, ORG]
newsChronicle	True-False	**1** (1)	[OTHER]
ocr-contratos	True-False	**1** (1)	[NUM]
uofthacks6	True-False	**1** (1)	[OTHER]
Text Topic Classification (Google `text_classify`)			
DMnMD	True-False	**1** (1)	[Health]
HLPFL	True-False	**1** (8)	[Public Safety, Law & Government, Emergency Services, News, ...]
MirrorDashboard	True-False	**1** (7)	[Jobs & Education, Law & Government, News, ...]
noteScript	True-False	**1** (1)	[Food]
pennapps_2019f	True-False	**1** (2)	[News/Politics, Investing]
soap	Multi-Choice-API	**2** (2, 2)	[Sensitive Subjects, ...]; [Discrimination & Identity Relations, ...]
SocialEyes	Multi-Choice-API	**2** (2, 1)	[people & society, sensitive subjects]; [adult]
Twitter_Mining_GAE	True-False	**1** (1)	[Sentitive]
vfriendo	True-False	**1** (1)	[Restaurants]

Appendix B Loss function for other decision-process summaries

True-False:

$$L(y) = \overbrace{\text{Sigmoid}(\max_{l\in G_{\hat{c}}}(\mathbf{y}) - \theta)}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\text{Sigmoid}(\theta - \max_{l\in G_c}(\mathbf{y}))}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} \tag{4}$$

Multi-Select:

$$L(y) = \overbrace{\sum_{c\in\hat{T}} \text{Sigmoid}(\theta - \max_{l\in Gc} \mathbf{y}[l])}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\sum_{c\in\cup_c G_c\setminus\hat{T}} \text{Sigmoid}(\max_{l\in Gc} \mathbf{y}[l] - \theta)}^{\text{Penalize } \textbf{Type-3} \text{ Critical Errors}} \tag{5}$$

Multi-Choice API-order: Here we explain why this loss function captures the critical errors:

- A Type-1 error occurs, if (1) the correct target class is matched, thus at least one of its labels has a score above the confidence threshold ($\max_{l\in G_{\hat{c}}} \mathbf{y}[l] \geq \theta$), and (2) it is matched after the EOD because all of the labels belonging to the correct target class have scores below the maximum score of labels in the incorrect target classes.
- A Type-2 error occurs if the maximum score for labels in a correct target class falls below threshold θ, thus it is never matched (before or after EOD).
- A Type-3 error occurs if any labels belonging ($\max_{l\notin G_{\hat{c}}} \mathbf{y}[l]$) to incorrect target classes appears before labels in the correct target class.

$$L(\mathbf{y}) = \overbrace{\text{Sigmoid}\left(\max_{l\in\cup_{c\neq\hat{c}}G_c} \mathbf{y}[l] - \max_{l\in G_{\hat{c}}} \mathbf{y}[l]\right)}^{\textbf{Type-1} \text{ Critical Errors}} + \overbrace{\text{Sigmoid}\left(\max_{l\in\cup_{c\neq\hat{c}}G_c} \mathbf{y}[l] - \theta\right)}^{\textbf{Type-2} \text{ Critical Errors}} + \overbrace{\sum_{c\neq\hat{c}} \text{Sigmoid}\left(\max_{l\in G_c} \mathbf{y}[l] - \max_{l\in G_{\hat{c}}} \mathbf{y}[l]\right)}^{\textbf{Type-3} \text{ Critical Errors}} \tag{6}$$

Value ranges: As for APIs that output a score $\mathbf{y}$ to describe the input, applications typically define several value ranges as target classes to make decisions, where the lower bound of the c^{th} target class is denoted as l_c and the upper bound of the c^{th} target class is denoted as u_c.

$$L(y_i) = \overbrace{\text{Sigmoid}(\mathbf{y} - u_{\hat{c}}) + \text{Sigmoid}(l_{\hat{c}} - \mathbf{y})}^{\textbf{Type-1} \text{ Critical Errors}} + \overbrace{\sum_{c\neq\hat{c}} \text{Sigmoid}(u_c - \mathbf{y}) + \text{Sigmoid}(\mathbf{y} - l_c)}^{\textbf{Type-3} \text{ Critical Errors}} \tag{7}$$

where $\hat{c}$ is the index of the correct target class. A Type-1 error occurs (*i.e.,* a correct target class is matched after EOD) when the output score $\mathbf{y}$ exceeds the upper bound of the ground-truth value range (u_c), or falls below the lower bound of the ground-truth value range (l_c). A Type-3 error occurs when the upper bound of an incorrect value range exceeds $\mathbf{y}$ *and* its lower bound falls below $\mathbf{y}$, leading it to be selected. Type-2 errors are absent in this application because all the target classes span the whole output range, thus a target class must be matched.

Appendix C Setup of Categorized models

Here, we describe in detail how we construct the label categories to support the scheme of Categorized models, which is one of the schemes in comparison with ChameleonAPI (Section 5).

Image-classification Image-classification APIs typically contains many thousands of labels without providing their categorization or hierarchy. Therefore, we create categories leveraging the Wikidata knowledge graph [68], a widely used knowledge graph database that has been referred to during the creation of many popular ML training datasets [23, 34, 44]. In this knowledge graph, each node is a named entity, covering *all* the labels used in popular image-classification APIs [2, 20, 52], and each edge represents a relationship between two entities (e.g., "subclass of", "different from", "said to be the same as").

Extracting "subclass of" edges in Wikidata knowledge graph, we get a directed acyclic graph (DAG) of label hierarchy. We believe it offers a principled foundation to create label categories based on two observations: (1) If an entity/node e is reachable from another entity/node e' through several subclass-of edges, e' is also covered by the *category* of e (e.g., entity "motor vehicle" is directly connected to "land vehicle", entity "land vehicle" is directly connected to "vehicle", so "motor vehicle" is also covered by the "vehicle" category); (2) The distance, measured in the number of subclass-of edges, between an entity/node and the DAG root indicates the specificity of the concept behind this node, with shorter distance representing coarser-grained categories. We will refer to a node that is k edges away from the root as a Level-k node.

Based on these observations, we formally define a set of categories C_k for all the image-classification labels L at a specificity level k as follows: C_k is the minimum set of Level-k nodes such that every label $l \in L$ is covered by at least one category node $c_k \in C_k$. We could categorize all the applications into single-category and multi-category applications using any level of specificity settings. In this paper, we adopt Level-2 specificity setting, since the number of single-category applications drops a lot when moving from Level-2 to Level-3, indicating Level-3 categories may be too fine-grained.

Under Level-2, we set up 35 categorized models that cover all the image-classification labels. Six of them are used by applications in our benchmark, including *natural object*, *temporal entity*, *artificial entity*, *system*, *phenomenon*, and *continuant*. With this categorization, 27 of the 40 image-classification applications are single-category, and the rest 13 applications are multi-category.

Object-detection Similar as image-classification API, every object-detection API label also corresponds to an entity node in the Wikidata knowledge graph. Therefore, we use the same methodology and the same specificity Level-2 to define categories for object detection labels.

Seven categorized models are set up to cover all object-detection labels. The object-detection labels used by 8 object-detection benchmark applications belong to 3 categories: *natural object*, *artificial entity*, and *system*. Under this setting, there will be 7 single-category applications, and 1 multi-category applications.

Text-classification The Google text-classification API [20] offers the hierarchy tree of all its labels. We simply follow their categorization and get 15 categories to covering all text-classification labels. Nine categories are used by applications in our benchmark, including *business & industrial*, *people & society*, *health*, *food & drink*, *jobs & education*, *news*, *sensitive subjects*, *adult*, and *law & government*. Under this categorization, there are 5 single-category text-classification applications, and 4 multi-category text-classification applications.

Other types of applications There are two other types of applications in our benchmark that are not suitable for designing pre-specialized models: sentiment analysis and named entity recognition.

For sentiment analysis API, the corresponding applications typically define several value-ranges and determine which range the API output (a sentiment score) falls into. Since the API output is a floating point number, there are infinite ways of defining value-ranges. Therefore, it is impracticable to create pre-categorized models.

For named entity recognition API, it only has 6 labels: *person, location, organization, number, date*, and *misc* [20]. They are already high-level categories. There is no need to create pre-categorized models for each category.

Appendix D Results of other applications

As mentioned in §5.1, the results of the 20 applications that involve sentiment analysis and entity recognition were not included in the evaluation section. Their results are shown in Figure 10 and 11. As we can see, the advantage of ChameleonAPI is consistent across these applications, similar to what we presented in§5. Note that, the scheme of Categorized models does not apply to applications that involve these two types of ML tasks, and hence is not included in Figure 10 and 11.

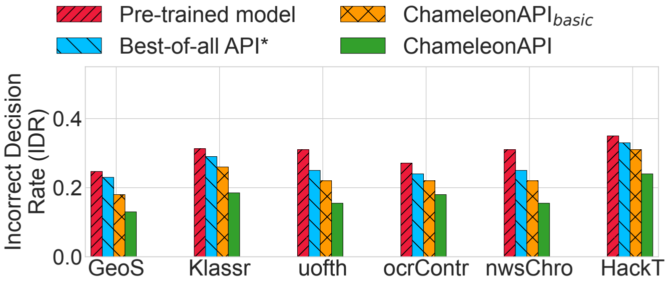


Figure 10: Results on entity-recognition applications.

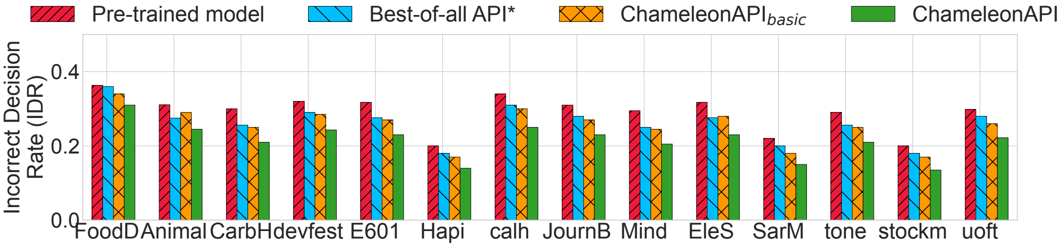


Figure 11: Results on sentiment-analysis applications.

SquirrelFS: using the Rust compiler to check file-system crash consistency

Hayley LeBlanc
University of Texas at Austin

Nathan Taylor
University of Texas at Austin

James Bornholt
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin

Abstract

This work introduces a new approach to building crash-safe file systems for persistent memory. We exploit the fact that Rust's typestate pattern allows compile-time enforcement of a specific order of operations. We introduce a novel crash-consistency mechanism, *Synchronous Soft Updates*, that boils down crash safety to enforcing ordering among updates to file-system metadata. We employ this approach to build SquirrelFS, a new file system with crash-consistency guarantees that are *checked at compile time*. SquirrelFS avoids the need for separate proofs, instead incorporating correctness guarantees into the typestate itself. Compiling SquirrelFS only takes tens of seconds; successful compilation indicates crash consistency, while an error provides a starting point for fixing the bug. We evaluate SquirrelFS against state of the art file systems such as NOVA and WineFS, and find that SquirrelFS achieves similar or better performance on a wide range of benchmarks and applications.

1 Introduction

One of the most important properties for file systems is to preserve their integrity and user data in the face of a crash or a power loss [16, 20, 28, 31, 42, 43, 51]. Unfortunately, building crash-consistent file systems is challenging; checking or ensuring crash consistency is even more so [17, 40].

There are two main approaches to building file systems today, as summarized in Table 1. First, we build file systems using low-level languages like C, and we use runtime testing to gain some confidence in the correctness of the systems [36, 37, 41, 46, 47, 59]. Note that this approach is necessarily incomplete; testing can only reveal bugs, not prove their absence. However, this approach allows rapid development, and entire testing ecosystems have sprung up around this basic approach, like the widely-used xfstests [10] and Linux Test Project [5].

A different approach to building file systems is to verify them: we write a high-level specification of correct behavior (including crash behavior) and then prove that the implementation matches the specification [17–19, 29, 53]. This approach can prove that the implementation does not have certain classes of bugs; however, it comes at a high cost. For each line of code in the implementation, we may need to write 7–13 lines of proof. Writing and maintaining proofs is time-consuming and requires specialized expertise, constraining rapid development.

Approach	Complete	Dev effort	Time to check
Testing	No	Low	Medium
Verification	Yes	High	High
This work	Yes	Medium	Low

Table 1: Comparison of different approaches to ensuring crash consistency in file systems.

In this work, we seek to find a middle ground between these two approaches. We would like to verify some aspects of file systems, but without the burden of having to write and maintain proofs. In particular, we are interested in *crash consistency*, a correctness property that is especially difficult to test for. In order to be crash consistent, systems must ensure that updates become persistent on storage media *in the correct order*; however, hardware or caching layers may reorder updates to improve performance in unanticipated ways. Exposing crash-consistency bugs thus requires one to find and reproduce these low-level orderings, which requires specialized testing software [36, 37, 41, 46, 47, 59]. Our goal is to develop lightweight approaches to statically check for crash-consistency bugs without the overhead of full verification.

We exploit two recent developments to achieve this goal (§2). First, the Rust programming language has a strong type system that supports powerful compile-time safety checks. Our work takes inspiration from Corundum [32], a Rust crate (library) that uses Rust's type system to check low-level PM safety properties. In this work, we observe that Rust's type system can also statically enforce that certain operations are carried out in a given order [27, 38]. Since the root of crash consistency is ordering updates to storage, if we can encode those ordering-based invariants in the type system, the com-

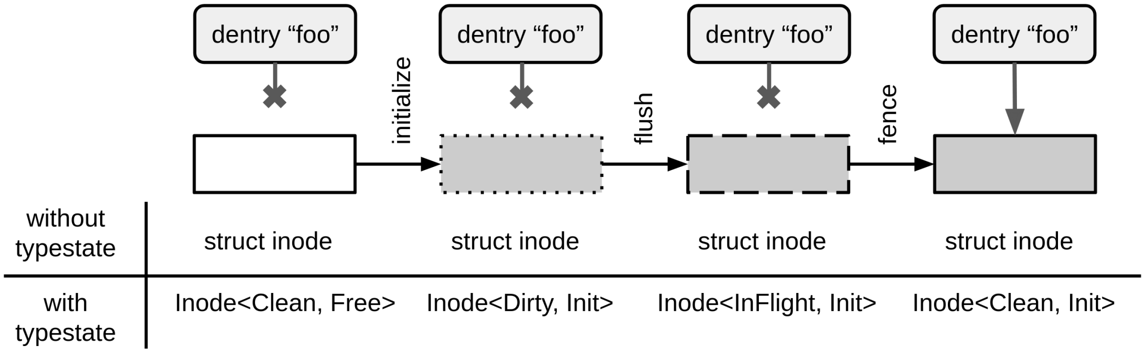


Figure 1: For a soft updates file system to be crash-consistent, directory entries should only point to fully-initialized, durable inodes. In existing file systems, all persistent inodes have the same type, regardless of whether they are durable or have been initialized. With typestate, durability and the inode’s contents are reflected in its type.

piler can ensure the invariants hold at compile time.

However, to do so, crash consistency must be derived purely from ordering-based invariants; some mechanisms such as journaling use writes to a log to obtain atomicity, which is harder to encode in the type system. Soft updates achieves crash consistency purely via ordering [43], but the traditional soft updates scheme is complex and hard to implement [13, 25].

We observe that the low latency of persistent memory [56, 59] allows file-system operations to be *synchronous*; all updates to storage media are durable by the time each operation returns [24, 34, 35, 39, 57]. We take advantage of persistent memory’s synchronous updates and byte addressability to develop a new mechanism for crash consistency we term *Synchronous Soft Updates*.

Synchronous Soft Updates builds on the classical soft updates mechanism [43], but avoids most of the complexity that prevented the widespread adoption of soft updates [13]. Two of the most complicated aspects of soft updates, dependency structures and cyclic dependency management, arise due to the need to track ordering requirements between block-sized updates across asynchronous operations. Synchronous Soft Updates eliminates these challenges entirely by using fast, fine-grained storage to back synchronous operations.

We ensure that the ordering invariants of Synchronous Soft Updates hold by using the Rust compiler. We take advantage of Rust’s support for the *typestate pattern*, an API design pattern where an object’s type reflects the operations that have been performed on it [54]. The legal order of operations is encoded in function signatures and enforced by Rust’s typechecker. For example, an uninitialized inode has a different type than an initialized one; attempting to use one where the other is expected will result in a compile-time error. Figure 1 illustrates the approach.

We implement Synchronous Soft Updates in a new file system for PM called SquirrelFS and use the typestate pattern in Rust to check that update orderings are implemented correctly. SquirrelFS provides crash-atomic metadata system calls, including `rename`; on the original soft updates, a crash during rename could result in both the source and destination existing. SquirrelFS compiles and typechecks in seconds, whereas running verification on existing storage systems takes minutes or hours. Building SquirrelFS required no modifications to the Rust language.

We evaluate SquirrelFS by comparing to a number of file systems meant for persistent memory, such as NOVA [57] and WineFS [34] (§5). We use Intel’s Optane DC Persistent Memory Module for our comparison, and find that SquirrelFS offers comparable or better performance to other PM file systems across a broad range of workloads. The current SquirrelFS prototype prioritizes simplicity of update ordering rules over performance in some areas, leading to relatively high mount times and memory utilization; however, these are not fundamental limitations of the design. We also model the design of SquirrelFS using the Alloy model-checking language [33] to gain confidence in the correctness of its Synchronous Soft Updates mechanism.

We note that SquirrelFS is not fully verified, and thus does not obtain the strong correctness guarantees of verified storage systems like FSCQ [19]. Crash-consistency bugs may still occur in SquirrelFS if their root causes are unrelated to ordering, if the ordering rules enforced by the compiler are incorrect, or if trusted code in SquirrelFS’s implementation or the Rust compiler are buggy. For example, SquirrelFS’s ordering rules guarantee that inodes are always initialized before they are linked into the file system tree, but they do not guarantee that the contents of the inode are correct. SquirrelFS’s static checks are also limited by the capabilities of the Rust compiler. For instance, the Rust compiler cannot check properties about variable-sized sets of data structures, as checking such properties is undecidable in general.

SquirrelFS offers a useful new point in the spectrum of approaches to building robust storage systems; it provides weaker guarantees than verified systems, but comes at a lower cost. As such, we hope that it proves useful for developers of storage systems that require strong guarantees, good performance, and rapid development.

In summary, this work makes the following contributions:

- Statically-checked crash consistency, an approach where high-level properties are encoded into the type system and checked at compile time (§3)
- The Synchronous Soft Updates crash-consistency mechanism for persistent-memory file systems (§3.1)
- The SQUIRRELFS prototype, along with a discussion of lessons learned during its development (§4).

SQUIRRELFS and its Alloy model are publicly available at https://github.com/utsaslab/squirrelfs.

2 Background and Motivation

2.1 Crash Consistency

A file system is *crash consistent* if it can recover to a consistent state after a power loss or a crash [16,20,51]. A consistent file system is one where all the metadata is in sync; for example, two files cannot (mistakenly) claim the same data block. Files present before the crash must exist post-crash, and the data in files must remain valid.

Crash-consistency mechanisms. Crash consistency is generally achieved using mechanisms such as journaling [28,48], copy-on-write [1,31,42], or soft updates [43]. The root of crash consistency is correctly *ordering* writes to storage [20]; for example, a data block must be initialized before a file points to it. Soft updates achieves crash consistency by carefully ordering in-place updates to storage such that all possible crash states are consistent [43]. To enforce ordering, soft updates must track updates across asynchronous operations and resolve cyclic dependencies when they arise. Though soft updates is used in FreeBSD [44], it has not been widely adopted due to its high complexity.

Ensuring crash consistency. Ensuring that a given file system achieves crash consistency is challenging. There are two main approaches. The first approach is testing, in which possible crash states of a file system are explored and checked for consistency. Obtaining these crash states requires support from tools like eXplode [59], CrashMonkey [46], Hydra [37], Chipmunk [41], or Vinter [36]. While such testing tools can find many bugs, they cannot prove overall correctness or the absence of crash-consistency bugs.

The second approach is to build verified file systems. A developer writes a high-level specification of correctness and a lower-level implementation, and proves that the implementation satisfies the specification. This approach is stronger than testing in that it can prove strong correctness properties and verify that there are no bugs. However, it comes at a high cost: the developer has to write 7–13 lines of proof for every line of code. For example, BilbyFS [12] required 13k lines of proof for 1k lines of implementation code; VeriBetrKV [29] used 45K lines of proof for 6k lines of implementation. Another verified file system, FSCQ [19], has interleaved proof and implementation code that is 10× the size of the most similar unverified system.

This heavy proof burden constrains development in a number of ways. First, building a verified system requires proof-writing expertise, which restricts the set of developers who can work on it. Second, proofs must be written in tandem with the code that they verify, which extends development time. Finally, making changes to the system requires corresponding changes to the proofs, making maintenance slow and preventing rapid updates.

Corundum [32] is a Rust crate for PM systems that, like SQUIRRELFS, uses the Rust type system to enforce certain low-level PM safety properties at compile time. For example, Corundum ensures that every update to PM occurs in a logged transaction, and prevents the storage of pointers to volatile memory in durable structures. SQUIRRELFS was inspired by Corundum and aims to enforce higher-level properties like file-system crash consistency with Rust.

2.2 The opportunity: Rust and PM

We observe an opportunity to ensure file-system crash consistency in a cheap manner.

First, we note that the Rust programming language can **statically enforce a specific order on operations** via its support for the *typestate pattern* [9,27]. Briefly, the typestate pattern enables an object's runtime state to be encoded in its type [54]. This state can be checked at compile time via typechecking, ensuring that a given operation can only occur on a specific type. Typestate information is stored in zero-sized types that incur no runtime overhead.

For example, one consistency rule enforced by soft updates is that a directory entry should never point to an uninitialized inode. Listing 1 shows how typestate is used to enforce this rule. To create a new file, we first obtain a free directory entry and inode. Initially, both objects have typestate `Free`. Then, we initialize the directory entry, transitioning its type to `Dentry<Init>`. The listing then has a bug in which the directory entry's inode number is set by `commit_dentry()` before the inode is initialized, breaking the consistency rule. The Rust compiler catches this bug because the inode's current typestate `Free` does not match the typestate `Init` expected by the function.

Since soft updates is entirely built on ordering updates to file-system objects, we can translate the required partial order into a set of types and use Rust's type checking to enforce the order. Thus, the invariants we want to maintain are translated into something the type system and compiler can enforce. We note that we are able to do this with an *unmodified* Rust compiler; the new types introduced are no different to the

```
fn new_file() {
    // Dentry<Free>
    let d = Dentry::get_free_dentry();
    // Inode<Free>
    let i = Inode::get_free_inode();
    // Dentry<Init>
    let d = d.set_name("foo");
    let d = d.commit_dentry(i);
                   ^ expected 'Inode<Init>',
                     found 'Inode<Free>'
}
```

Listing 1: The listing shows the typecheck process throwing an error when an uninitialized inode is passed to a function that expects an initialized inode.

compiler from existing types in the codebase.

However, implementing soft updates correctly remains challenging even with typestate support. With soft updates, file-system updates are applied to the page cache in DRAM, and then later written to storage in the right order. Determining the right order requires tracking complex dependencies across asynchronous operations. When a single file-system metadata object (such as an inode or a bitmap) is updated multiple times, it can lead to cyclic dependencies.

This leads to our second observation: persistent memory (PM) file systems support synchronous operations thanks to the low latency of the storage media [56, 58]. These file systems write updates directly to storage without first caching them in DRAM [24, 34, 35, 39, 57]. A **synchronous implementation** of soft updates for persistent memory **eliminates** the complexities of asynchronous dependency management, greatly simplifying the mechanism and allowing the relevant invariants to be encoded in Rust's type system.

3 SquirrelFS

We now present the design and implementation of SquirrelFS, a novel file system that uses the unmodified Rust compiler to check its crash consistency. If the compilation is successful, it indicates that the ordering-based invariants hold throughout the file system: in other words, the checking is complete. If compilation fails, the error reported by Rust is useful in figuring out which operations are out of order. Compilation takes only seconds, offering quick feedback to developers.

SquirrelFS is built on two key ideas:

- A novel crash-consistency mechanism, Synchronous Soft Updates, that achieves crash consistency purely via ordering file-system operations (§3.1)
- Using the Rust typestate pattern to encode ordering invariants into the Rust type system (§3.2)

It is important to note that we are not modifying the Rust compiler in any way. To the Rust compiler, it is no different from type-checking any other code base; we are merely using the type checking to ensure that crash consistency holds in the file system.

We now describe the key ideas in more detail.

3.1 Synchronous Soft Updates

We develop *Synchronous Soft Updates* (SSU), a novel crash-consistency mechanism. SSU is based on the traditional soft updates approach, but differs in two key aspects. First, soft updates was designed for asynchronous settings, but all operations are synchronous in SSU. Second, soft updates does not provide atomic rename; a crash during a rename of `src` to `dst` can result in both being present after a crash. SSU fixes this flaw; renames are atomic, and a crash during rename will result in either `src` or `dst` after recovery.

We now discuss why we developed SSU, its key aspects, and how renames are atomic in SSU.

Why a new mechanism? To go with our overall approach of encoding ordering-based invariants into the Rust type system, we needed a mechanism that achieves crash consistency purely via ordering file-system updates. This rules out mechanisms such as journaling and copy-on-write that use writes to a log or an extra copy to obtain atomicity. Soft updates [43] obtains crash consistency by enforcing ordering on in-place persistent updates to file-system objects; thus, it was a good match. However, traditional soft updates suffered from two problems that we needed to tackle. The first challenge was that soft updates had significant complexity arising from needing to track dependencies between asynchronous file-system operations; the presence of cyclic dependencies also requires complex roll-back and roll-forward logic. The second challenge is that soft updates does not provide atomic operations, particularly rename; atomic rename is a crucial primitive for a number of POSIX applications [50]. Thus, we need to modify soft updates to tackle both its high complexity and lack of atomic operations.

Synchronous operations. We observe that the root of complexity in soft updates (such as cyclic dependencies and structures for tracking dependencies) is *asynchrony*. A *synchronous* implementation of soft updates neatly avoids these complexities. All updates would be made durable by the end of each system call, which would eliminate the need to track cross-operation dependencies. Cyclic dependencies would no longer arise because there are no pending updates that can conflict with each other. The SoupFS [23] soft updates file system for persistent memory eliminated cyclic dependencies using fine-grained updates, but still required asynchronous dependency tracking. A synchronous implementation is necessary to overcome both sources of complexity.

A synchronous version of soft updates was not feasible until now, as running this on magnetic hard drives or even solid state drives would be prohibitively slow. However, synchronous soft updates is a good match for persistent memory (PM) due to its low latency; system calls in many existing PM file systems are already synchronous [24, 34, 35, 57].

Similar to traditional soft updates, SSU maintains crash consistency by enforcing ordering among updates to file-system objects. SSU implements the original soft updates rules [26]:

1. Never point to a structure before it has been initialized;
2. Never re-use a resource before nullifying all previous pointers to it;
3. Never reset the old pointer to a live resource before the new pointer has been set.

These rules are significantly easier to enforce in a synchronous setting, as there is no need to track dependencies across asynchronous operations. Like soft updates, SSU focuses on the integrity of file system metadata and cannot guarantee that operations on file data are atomic. SSU could be combined with journaling or copy-on-write to obtain stronger data guarantees.

Atomic rename in SSU. SSU ensures renames are atomic by cleaning up file-system state after a crash. In traditional soft updates, if there is a rename from `src` to `dst`, it is impossible to tell after a crash whether `src` or `dst` should be removed. To resolve this, SSU adds an extra field, called the *rename pointer*, to directory entries in order to persistently save enough information to complete the rename operation after a crash. The rename pointer in the destination directory entry points to the physical location of the source directory entry. The rename pointer allows the file system to follow soft updates rule 3 (never reset the old pointer before the new one has been set) while also retaining the ability to distinguish between `src` and `dst` after a crash.

Note that this is similar to what journaling-based file systems do; they write a log entry specifying `src` and `dst` so that the right clean-up action can be performed. In SSU, the information in this log entry is distributed over the source and destination inodes; taken together, they provide enough information to the file system.

Figure 2 illustrates the process. Step ❶ shows an example system state prior to the `rename` operation. In ❷, `dst`'s rename pointer (dotted line) is set to `src`. `dst` is invalid, and `src` is still valid. In ❸, we make `dst` valid; this also logically invalidates `src`. This is an atomic point; after this step, the file system will always complete the rename operation. If the file system crashes prior to this step, the rename pointer is cleared on recovery. In ❹, we physically mark `src` as invalid. In ❺, the rename pointer is cleared, and in ❻ `src` is fully deallocated. Each step either modifies metadata that is invisible to the user (e.g., deallocating an orphaned directory entry) or atomically modifies a single 8-byte value. All modifications must be durable before proceeding to the next step.

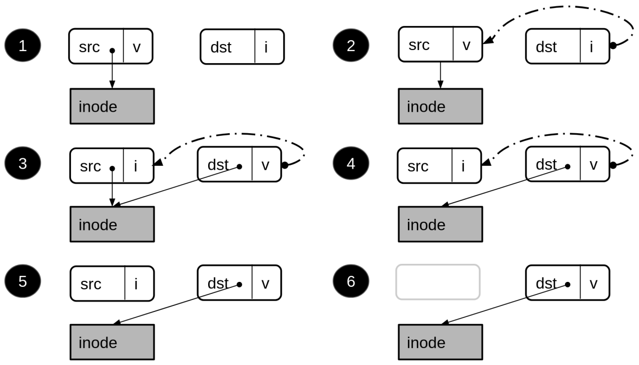


Figure 2: The figure shows the steps in atomic soft updates `rename`. The dotted lines represent rename pointers and the solid lines represent inode pointers. `src` and `dst` are directory entries. The labels "v" and "i" indicate whether a directory entry is valid or invalid.

A question that arises is how the file system finds `src` and `dst`. This is an example of how SSU is tailored for PM file systems. In PM file systems, it is common for the file system to scan persistent objects to construct indexes in DRAM; we add the rename-recovery procedure into this scan. Thus, when building volatile indexes after a crash, the file system also looks for and completes any partially completed rename operations.

3.2 Using Rust to enforce ordering

Rust's typestate pattern can be used to ensure that *a set of functions are always called in certain partial order*. A total order is not necessary, as many operations involve independent updates that can be safely reordered. As we discussed previously (§2), an object's typestate is encoded in generic type parameters in its definition, and the partial order is encoded in the function signatures of its associated functions.

We encode two states (as different type parameters) in the types of persistent objects:

- *Persistence* typestate is a representation of whether an object's most recent update(s) have been made durable. We use three persistence typestates: `Dirty`, `InFlight`, and `Clean`.
- *Operational* typestate represents the operations that have been performed on an object and is used to determine what operations can happen next.

Persistence and operational typestate are separate to capture the fact that most storage devices do not synchronously flush updates. For example, in persistent memory, updates go to

```
impl Inode<Clean, Free> {
    fn init_inode(self, ino: u64, ...)
        -> Inode<Dirty, Init> {...}
}
impl Dentry<Clean, Alloc> {
    fn commit_dentry(
        self,
        inode: Inode<Clean, Init>
    ) -> Dentry<Dirty, Committed> {...}
}
impl<S> Inode<Dirty, S> {
    fn flush(self) ->
        Inode<InFlight, S> {...}
}
impl<S> Inode<InFlight, S> {
    fn fence(self) ->
        Inode<Clean, S> {...}
}
```

Listing 2: Pseudocode implementations of file system objects with persistence and operational typestate. Typestate arguments are shown in bold.

the CPU caches first, and must be explicitly flushed to the persistent media.

Listing 2 shows implementations of several methods of persistent Inode and Dentry types with persistence and operational typestate as generic type parameters. The methods flush and fence invoke a cache line write back and store fence respectively and are generic with respect to operational typestate. These methods must be used to ensure updates are persistent before continuing; for example, commit_dentry() requires an Inode<Clean, Init> to ensure the inode's initialization cannot be transparently reordered with the directory entry updates.

This formulation of persistence typestate has several performance benefits. First, because the flush and fence methods can only be called on an object whose typestate indicates it is not yet persistent, typechecking will prevent redundant persistence operations (thereby improving performance). Second, developers can wait to flush updates until it is strictly necessary and can write additional transitions to enable multiple updates to share a single fence.

Why Rust? In order to obtain useful compiler-checked guarantees from the typestate pattern, each object must have exactly one typestate [54]. Thus, languages with unrestricted aliasing (e.g., C) cannot support the typestate pattern, as different aliases for the same value can have different types. Rust supports the pattern via its ownership type system, which ensures that each value has exactly one owner (and thus exactly one type).

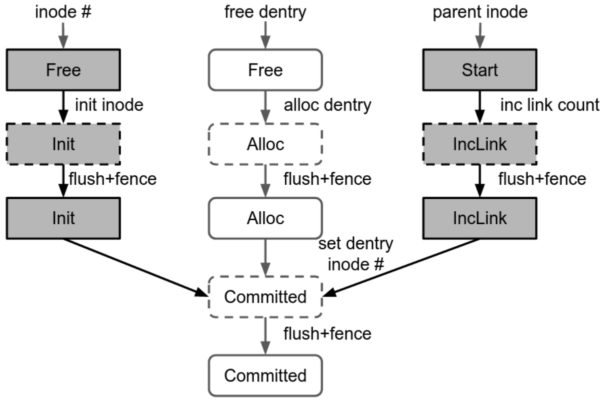


Figure 3: The figure shows the persistent updates and corresponding dependencies made during mkdir. Inodes are dark gray and directory entries are white. Each object is labeled with its operational typestate and its outline indicates whether it is clean (solid) or dirty (dotted).

3.3 Example: mkdir

We use mkdir to illustrate the typestates and dependency rules used in SSU. To be crash consistent, an SSU implementation of mkdir must ensure (1) that a structure never points to an uninitialized resource, and (2) that each inode's link count is greater than or equal to its actual number of links. Both rules prevent dangling links in the event of a crash.

Figure 3 illustrates the dependencies in a mkdir operation. During mkdir, three file-system objects are modified: an inode for the new directory, a directory entry for the new directory, and the inode of the parent directory. Note that all three can be modified at the same time in a concurrent fashion, and can share a single store fence at the end (not shown). SQUIRRELFS uses volatile allocation structures, so they are not persisted during mkdir.

The system first finds the parent inode and obtains a free directory entry in one of the parent's pages as well as a free inode. The inode is then initialized (i.e., setting its inode number, link count, timestamps), the directory entry's name is set, and the parent inode's link count is incremented.

Next, we commit the directory entry by setting its inode number. This makes the directory entry valid and connects the inode to the file system tree. Directory entry commit is dependent on inode and directory entry initialization and parent link increment. Committing the directory entry before initializing the inode can result in a directory entry pointing to a garbage inode; committing before incrementing the parent's link count can lead to dangling links.

3.4 Implementation

We implemented SQUIRRELFS in Rust with 7500 LOC. It uses bindings from the Rust for Linux project [8] to con-

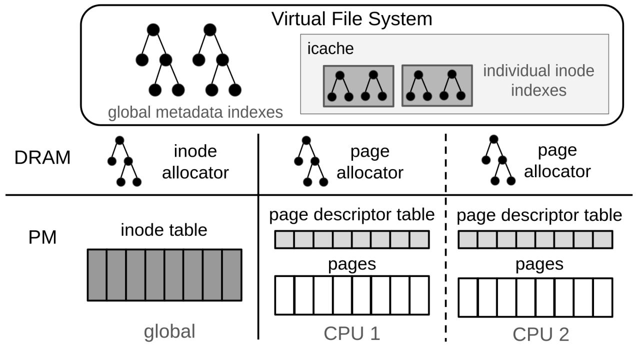


Figure 4: The figure shows the main components of SQUIRRELFS. Each CPU has its own pool of pages and private page allocator. The inode allocator is shared between all CPUs. Volatile indexes are stored in VFS data structures.

nect to the Linux Virtual File System (VFS) layer. Figure 4 shows SQUIRRELFS's architecture. We also built a model of SQUIRRELFS in the model-checking language Alloy [33] to check its design for crash consistency issues. We describe our experience developing SQUIRRELFS in §4.

Overview. The design of SQUIRRELFS combines aspects of FreeBSD's FFS [44] and PM file systems such as NOVA [57] and WineFS [35]. Like FFS, it has a simple on-storage layout, and uses soft updates. Like other PM file systems, SQUIRRELFS uses volatile index structures that are built when the file system is mounted.

SQUIRRELFS's design was primarily influenced by two factors. First, we wanted to keep dependencies as simple as possible and avoid nested persistent structures that are difficult to represent in typestate. Second, we assume the x86 PM persistence model in which only aligned updates of 8 bytes (or smaller) are crash atomic [24]. Under the x86 model, persistent addresses can be accessed via regular memory stores, but the corresponding cache line must be flushed before updates become persistent; a memory barrier like a store fence must also be invoked to correctly order stores [52]. Durable structures may also be updated via cache-bypassing non-temporal store instructions, which still require a store fence for persistence ordering. This programming model influences the structure of persistent objects and restricts the set of legal orderings.

All system calls in SQUIRRELFS are synchronous, meaning that updates to durable structures made by each system call are durable by the time the system call returns. As such, `fsync` is a no-op in SQUIRRELFS. Metadata-related operations are also crash-atomic. Data-related operations are not atomic in the current SQUIRRELFS prototype, which matches the default behavior of other PM file systems like NOVA [57]. These operations could be made atomic by using copy-on-write to update file contents.

Persistent layout. SQUIRRELFS uses a simple layout to reduce the complexity of update dependencies. SQUIRRELFS splits the storage device into four sections: the superblock, the inode table, the page descriptor table, and the data pages. The inode table is an array of all of the inodes in the system. SQUIRRELFS reserves enough space for approximately one inode for every 16KB of data (four pages), the same ratio used by the Linux Ext4 file system.

The page descriptor array contains page metadata. Rather than having inodes point to the pages they own, each page descriptor contains a backpointer to its owner (similar to NoFS [21]) and stores its own metadata (e.g., its offset in the file). This approach simplifies dependency rules for updates involving page allocation and deallocation. All remaining space after the page descriptor table is used for data and/or directory pages.

Volatile structures. SQUIRRELFS's persistent layout simplifies typestate and update dependency rules, but it is not amenable to fast lookups. Therefore, SQUIRRELFS uses indexes in DRAM to speed up lookup and read operations. Each inode in the VFS inode cache has a private index for the resources it owns; index data for uncached nodes is stored in the VFS superblock.

Like many other PM file systems, SQUIRRELFS uses volatile allocators: allocation information is not stored in a persistent manner, but rather rebuilt each time the file system is mounted. It uses a per-CPU page allocator and a single shared inode allocator (which could be converted to a per-CPU allocator to improve scalability). The allocators use free lists backed by kernel RB-trees.

SQUIRRELFS's indexes and allocators are rebuilt by scanning the file system when SQUIRRELFS is mounted. An inode, directory entry, or page descriptor is considered allocated if *any* of its bytes are non-zero. Directory entries and page descriptors are only valid if their inode numbers are set; inodes are valid only if they are reachable from the root. Thus, updates that allocate new structures and set non-inode metadata fields need not be crash-atomic.

Synchronous Soft Updates. SQUIRRELFS uses an implementation of SSU for crash consistency. As shown in Figure 3, operations that involve creation of new objects must first durably allocate and initialize resources before linking them into the file system (setting the directory entry's inode in the example) to enforce rule 1 (never point to a structure before it has been initialized). Deallocation proceeds in reverse; links are first cleared, then the object itself is deallocated by zeroing all of its bytes. SQUIRRELFS enforces rule 2 of soft updates (never re-use a resource before nullifying all previous pointers to it) by treating durable objects that are not completely zeroed out as allocated and by ensuring via typestate that pointers to the object are cleared before the object can be zeroed.

Typestate transition functions. SQUIRRELFS updates the typestate of objects via *typestate transition functions*. These functions take ownership of the original object, modify it, and return it to the caller with the new typestate. These functions are defined only on certain typestates to ensure they are called in a safe order. For example, the typestate transition function `commit_dentry()`, shown in Listing 2, is only defined for directory entries with type `Dentry<Clean, Alloc>`, and also takes ownership of an inode of type `Inode<Clean, Init>`. Calling `commit_dentry()` out of order – e.g., on a directory entry that has not yet been persistently allocated – is a potential crash-consistency bug and results in a compiler error.

Concurrency. SQUIRRELFS supports concurrent file-system operations. It relies on VFS-level locking on durable resources like inodes. This locking, together with Rust's type system, ensures that each resource has only one owner – and only one type – at any time, enabling strong typestate-based compile-time checking. SQUIRRELFS uses internal locks to protect its allocators and indexes.

Building a model with Alloy. While the typestate pattern can enforce a given operation order, it cannot verify that this order is crash consistent. To gain more confidence that SQUIRRELFS's design is crash consistent, we built a model of SQUIRRELFS in the Alloy model checking language [33].

Alloy provides a language for specifying transition systems and a model checker to explore possible sequences of states (traces) of these systems. Alloy's implementation is based on a logic of relations; each system is composed of a set of constraints that define a set of structures and the relations between them, and the model checker uses constraint solving to find traces.

In SQUIRRELFS, there is roughly a one-to-one mapping between typestate transitions in the Rust implementation and the next-state predicates in the Alloy model. Each next-state predicate specifies the states in which the transition may occur and the changes it makes to the model's state.

The model includes next-state predicates for typestate transitions and persistent updates. It also includes transitions that model crashes and recovery, which let us check SQUIRRELFS's design for crash-consistency bugs.

Each persistent structure in SQUIRRELFS is represented by a corresponding structure, also called a signature, in Alloy. The model also includes a `Volatile` signature that is used to model volatile aspects of the file system like its indexes. Each typestate is represented by a signature, and instances of persistent structures are mapped to their current typestate. Each file system operation is also represented by a signature, and relations map system calls to instances of persistent objects they are operating on as well as other volatile state (e.g., the locks held by that operation). We use this to model concurrent file-system operations.

3.5 Limitations of the approach

It is important to note that the typestate-based approach used in SQUIRRELFS is not as powerful as full verification. Fully-verified systems, such as the FSCQ file system [19], use theorem provers that can prove a wide variety of complex properties. For example, a developer could prove, if required, that the system only uses even-numbered inodes for files.

In contrast, our typestate-based approach can only check *ordering-based* invariants. Our approach could be used to verify that functions are called in a specific order; for example, our approach can ensure that a file is not linked into the file-system tree before it is allocated. However, it does not verify the implementation of each function that is called.

Thus, full verification is significantly more powerful and general, but it pays a cost in terms of complexity and development time. Our approach is more targeted and ordering-based, but allows quick feedback and incremental development.

We believe this approach is a valuable addition to the repertoire of tools we have for building correct file systems. This approach should be used alongside runtime testing and model-checking approaches.

3.6 Relevance beyond PM

While we have designed SQUIRRELFS for persistent memory, SQUIRRELFS would be relevant for any storage technology with byte-addressability and low latency. The Compute Express Link [2] standard will support attached memory devices, including PM, via the Type 3 (CXL.mem) protocol. These CXL-attached PM devices will have the same interface and persistence semantics as current NVDIMMs, though performance will be lower [14].

SQUIRRELFS, and SSU file systems in general, could be used on CXL-attached memory. As SQUIRRELFS's mount performance and memory footprint are tied to the size of the device, they may worsen with significantly larger-capacity devices. Further work will be required to optimize file systems based on our approach for such devices.

4 Experience developing SQUIRRELFS

We now describe our experience with designing, developing, and testing SQUIRRELFS. We also discuss the challenges we faced during this process.

4.1 Development process

Designing SQUIRRELFS. Our initial design closely followed that of BSD FFS [43], but most aspects eventually diverged due to differences between storage hardware and typestate considerations. We found that some data structures and crash-consistency properties were better suited for use with the typestate pattern than others. For example, we chose SquirrelFS's backpointer-based page management approach because it simplifies update dependency rules when allocating or deallocating pages. With backpointers, these operations involve a constant number of persistent updates and involve no additional durable structures. In contrast, tree or log-based approaches need extra persistent metadata and may require additional updates to balance the tree or free log space, both of which complicate dependencies and typestate management.

An important design decision we had to make was how granular typestate would be. One option was to use specific typestates to represent each fine-grained operation; e.g., have one typestate for initializing an inode's link count, another for setting its flags, etc. Another was to make each typestate more general, with transition functions potentially performing multiple persistent updates. More general typestates may sacrifice some bug-finding power, but they make the system easier to understand and develop. In SquirrelFS, we aimed to strike a balance by representing only operations that require a specific ordering with typestate. For example, when initializing an inode in SquirrelFS, the order in which the values of most fields are set is not relevant to crash consistency, as the contents of the inode are not visible until it is linked into the file system tree. Therefore, SquirrelFS uses only a single typestate (`Init`) to represent inode initialization, and another (`Committed`) to indicate when it has been added to the tree.

Parallel model and system implementation. We developed the Alloy model alongside SquirrelFS. This created a useful feedback loop in which the model supported the Rust implementation, and questions or changes to the implementation could be quickly reflected and checked in the model. We used an incremental development process, incorporating feedback from the Rust compiler and the model immediately as we implemented the system. Many transitions in the model could be translated directly into Rust typestate transitions, making the model a valuable guide for implementing file system operations. When we made mistakes translating the model into Rust, typestate checking quickly caught these issues.

Alloy also includes a graphical user interface for visualizing traces of operations on the model. This was useful for both investigating invariant violations and seeing the set of transitions that occur in a given file system operation, which could be translated directly into system call handler implementations. It also demonstrated locations where multiple updates could safely share a single store fence, which helped us avoid redundant fences.

4.2 Finding bugs

While developing SquirrelFS, we used a combination of typestate checking, model checking in Alloy, and dynamic testing to find bugs.

Typestate checking. Typestate checking in the implementation was successful at quickly catching both missing persistence primitives and higher-level ordering bugs; we provide an example of each.

- *Missing persistence primitives*. Our initial implementation of `write` was missing flush and fence calls after setting the backpointer of a newly-allocated page. This bug was immediately highlighted as an error by the compiler. Had this bug made it into the implementation, a crash could cause a file to have a size larger than the number of pages associated with it, causing errors when trying to read the file.
- *Incorrect ordering*. Our initial `rename` implementation mistakenly decremented an inode's link count before clearing the corresponding directory entry. A crash could result in a link count that is lower than the true number of links, leading to a dangling link if the inode is subsequently deleted.

Although we did not specifically check execution paths without crashes, the crash-consistency invariants encoded in typestate were general enough to detect some bugs in this code. For example, the compiler caught a bug where pages were not fully deallocated during `unlink`, which did not require a crash to manifest. Typestate-related compiler errors were relatively uncommon overall, since using the model as a guide for implementation helped us get ordering right early. However, it provided a crucial safety net to prevent subtle bugs when we did make mistakes.

Model checking with Alloy. The Alloy model found several high-level issues in SquirrelFS's design that would have otherwise been difficult to detect and time-consuming to fix, including the following examples.

- We initially believed that crash recovery would not be needed other than to fix space leaks. Alloy found an instance of the model where a crash during rename followed by deallocation of the destination directory entry could cause an invalid directory entry to reappear. Fixing this required the addition of recovery transitions.
- Early designs for SquirrelFS stored . and .. directory entries durably. We discovered via model checking that our original dependency rules for handling these directory entries during more complicated operations like `rename` were not correct. Ultimately, we decided to not store these entries, since they can be constructed at runtime using indexed information.

Testing. Neither the typestate pattern nor the Alloy model eliminated the need to test SquirrelFS. Our primary goal was to check crash-consistency, and we did not check any invariants that only impact regular, non-crash execution. We used handwritten tests and the xfstests suite [10] to test these unchecked parts of the code.

All bugs found through testing were in parts of SQUIRRELFS that were not checked by typestate or directly modeled in Alloy. Most bugs were related to updating volatile indexes or VFS inodes, e.g., failing to remove a deallocated object from an index or setting the wrong value in the VFS inode. There were also bugs in the implementations of typestate transitions, which are not themselves verified; for example, the transition that wrote new file data to a page did not always calculate the offset for non-aligned writes correctly. Implementing bug fixes was quick since we did not need to modify the typestate-restricted interface to objects and there were no proofs to update.

4.3 Challenges encountered

Challenges with typestate. It is easier to write typestate-checked code than it is to write verified code, but this comes at the cost of less powerful compile-time checking. For example, checking universally-quantified formulas (e.g., all pages in a file are allocated) is undecidable, and unlike verification-aware languages, the Rust compiler has no heuristics to attempt to solve them. As a result, we cannot ensure invariants such as "all objects in a set are in a certain typestate"; specifically, we can't encode this in typestate because the number of objects in the set is not known at compile time.

This became a problem when implementing file-system operations like `unlink`, where we would like to e.g., check that the backpointers of all pages belonging to the file are cleared before deallocating the inode. Such a check ensures that the system always follows soft updates rule 2 (never reuse a resource before nullifying all previous pointers to it); by clearing all of the page backpointers before deleting the inode, we ensure that none remain when the inode is eventually reused. However, it is impossible to check this property on arbitrary sets of pages if each page has its own typestate. We experimented with several workarounds, including forcing `write` operations to update no more than one page at a time (which was prohibitively slow and did not solve the problem for `unlink`), and storing typestate in page structures at runtime and manually adding assertions (which also impacted performance and lost the benefit of static checking). Ultimately, we decided to use a single piece of typestate to represent *ranges* of pages (e.g., all of the pages in a file or a contiguous subsection). Each typestate operation on such a range performs the operation on all pages in the range. This moved some logic into the typestate transitions, making the transition functions themselves more complicated but making page-management logic more centralized and easier to manually audit.

Challenges with Alloy. As SQUIRRELFS grew in complexity, it became harder to maintain the model and get useful feedback quickly. The model checker uses a SAT solver to check invariants, and the formulas representing a large model can take days or weeks to solve. We checked that traces with multiple concurrent operations were crash consistent, which increased the size of the problem further. To get faster feedback, we built a custom utility to run multiple independent instances of the model checker in parallel and split larger predicates into smaller, more concrete sub-checks.

It could also be difficult to determine whether a reported failure was a false positive. A particular challenge was dealing with *frame conditions*, predicates that specify what should not change in a given transition. Alloy is free to arbitrarily change any state that the current transition does not explicitly mention, so frame conditions are crucial to constrain the model to real traces. This behavior helps Alloy find corner-case bugs, but it also leads to false positives. To overcome this challenge, we built a syntax-based checker that parses the model using Alloy's API and checks that each transition explicitly mentions all mutable structures in the model. The current version of the checker cannot detect all issues, but it detected many missing conditions that would have otherwise taken hours to catch via model checking.

4.4 Typestate beyond SQUIRRELFS

Costs and benefits of typestate. We do not have equivalent verified or unverified systems to compare with SQUIRRELFS in terms of development and debugging effort; however, in the authors' experience, designing and implementing SQUIRRELFS required more effort than a typical unverified system, but far less effort than a verified storage system.[1] We believe that debugging SQUIRRELFS was faster and easier than debugging an equivalent unverified system, as following the typestate-enforced ordering rules made it easier to implement the system correctly in the first place and reduced the number of bugs overall.

Using the typestate pattern for crash consistency represents a useful new point in the tradeoff space between runtime testing and full verification. While it comes at the cost of additional development effort compared to unverified systems to determine correct ordering rules and does not gain the same correctness guarantees as verified systems, it does eliminate an entire class of crash consistency bugs that are otherwise difficult to find and fix [37,41,46]. Furthermore, as the pattern builds ordering rules directly into a system's implementation, the rules will stay up to date and continue to prevent crash-consistency bugs as the system is developed further [30,49].

Broader applicablity. As the typestate pattern is a general approach for statically checking the order of updates to data structures, it is useful in a broad variety of contexts, several of which are described below.

[1] For example, author LeBlanc recently worked on a durable log implemented in a verification-aware programming language, which took about 3 months of full-time work.

- Volatile data structures: SQUIRRELFS does not use typestate to manage updates to volatile data structures, but prior work on typestate verification has focused entirely on such use cases [11,54].

- Other types of storage systems: The typestate pattern could be used to enforce ordering invariants on durable updates in other types of storage systems (e.g., key-value stores) with different crash-consistency mechanisms. We note that crash-consistency mechanisms like journaling and copy-on-write do not achieve consistency entirely through ordering and would require auxiliary techniques to check properties like atomicity.

- Durable layout: SQUIRRELFS's on-storage layout is tailored to reduce the number of durable updates per file-system operation and to simplify ordering rules. Other layouts could also be used in typestate-checked storage systems, although the complexity of the ordering rules would increase.

- Asynchrony: The typestate pattern is compatible with asynchronous systems, although the ordering rules to enforce are much more complicated in such systems, as updates from different operations may be interleaved.

5 Evaluation

We seek to answer the following questions in our evaluation of SQUIRRELFS:

1. What is the latency of different file-system operations on SQUIRRELFS? (§5.2)
2. How does SQUIRRELFS perform on macrobenchmarks? (§5.3)
3. How does SQUIRRELFS perform on real applications? (§5.4)
4. How long does SQUIRRELFS take to mount and recover from crashes? (§5.5)
5. What compilation, memory, and CPU overheads does SQUIRRELFS incur? (§5.6)
6. Is SQUIRRELFS correct? (§5.7)

5.1 Experimental setup

We evaluate SQUIRRELFS on a two-socket, 32 core machine with 128GB of memory and one 128GB Intel Optane DC Persistent Memory Module. The evaluation machine runs Debian Bookworm and Linux 6.3.

We compare SQUIRRELFS against ext4-DAX [3], NOVA [57], and WineFS [34]. We configure all three systems to provide metadata consistency but not data consistency to match SQUIRRELFS's guarantees. We cannot compare SQUIRRELFS to SoupFS [23], the only other soft updates PM file system, as it is not open source. Due to time constraints, we were unable to compare against the recent ArckFS [60] file system. We hope to do so in the future. All reported results are the average of multiple trials. The red errors bars in Figure 5 indicate the minimum and maximum values recorded over all trials.

5.2 Microbenchmarks

We compare each system's latency by testing several file system operations: appending and reading 1KB and 16KB to a file, file creation, directory creation, renaming a directory, and unlinking a 16KB file. None of the tests call `fsync`.

The average latency over 10 trials of the tested operations are shown in Figure 5(a). The lowest latency file system in each test is either WineFS or SQUIRRELFS. Ext4-DAX has the highest latency on many operations because it interacts with the Linux kernel block layer for tasks like block allocation, which incurs additional software overhead. It achieves similar performance to the other systems on operations that do not go through the block layer (e.g., unlink). NOVA has higher latency on `mkdir` and `rename` than WineFS and SquirrelFS because operations that update multiple inodes require journaling in NOVA.

5.3 Macrobenchmarks

We evaluate SQUIRRELFS on the Filebench [4] storage benchmark suite. We run four workloads from the suite – fileserver, varmail, webserver, and webproxy – in their default configurations. Fileserver performs mostly writes with some whole file reads; varmail is half appends and half reads; webproxy appends to each file and reads from it several times; and webserver reads and occasionally appends to a log file. Figure 5(b) shows the average throughput in kops/sec for each file system on each workload. SQUIRRELFS achieves slightly better throughput than the next fastest system on fileserver and varmail (8% and 13% better, respectively) and within 10% of the fastest system on both webserver and webproxy. Fileserver and varmail perform many small appends, which SQUIRRELFS performs well on due to its lack of journaling. Webserver and webproxy are more read-heavy, which all systems perform roughly equally on. Ext4-DAX does not go through the block layer on reads and it benefits from data contiguity awareness, making its performance similar or better than the other systems on these workloads.

5.4 Applications

YCSB on RocksDB. We evaluate the four systems on RocksDB [7] using YCSB workloads [22]. We run all workloads on a 25GB database with 25M records, 25M operations, and 8 threads. All workloads are run using standard workload configurations and the default settings of YCSB, which uses

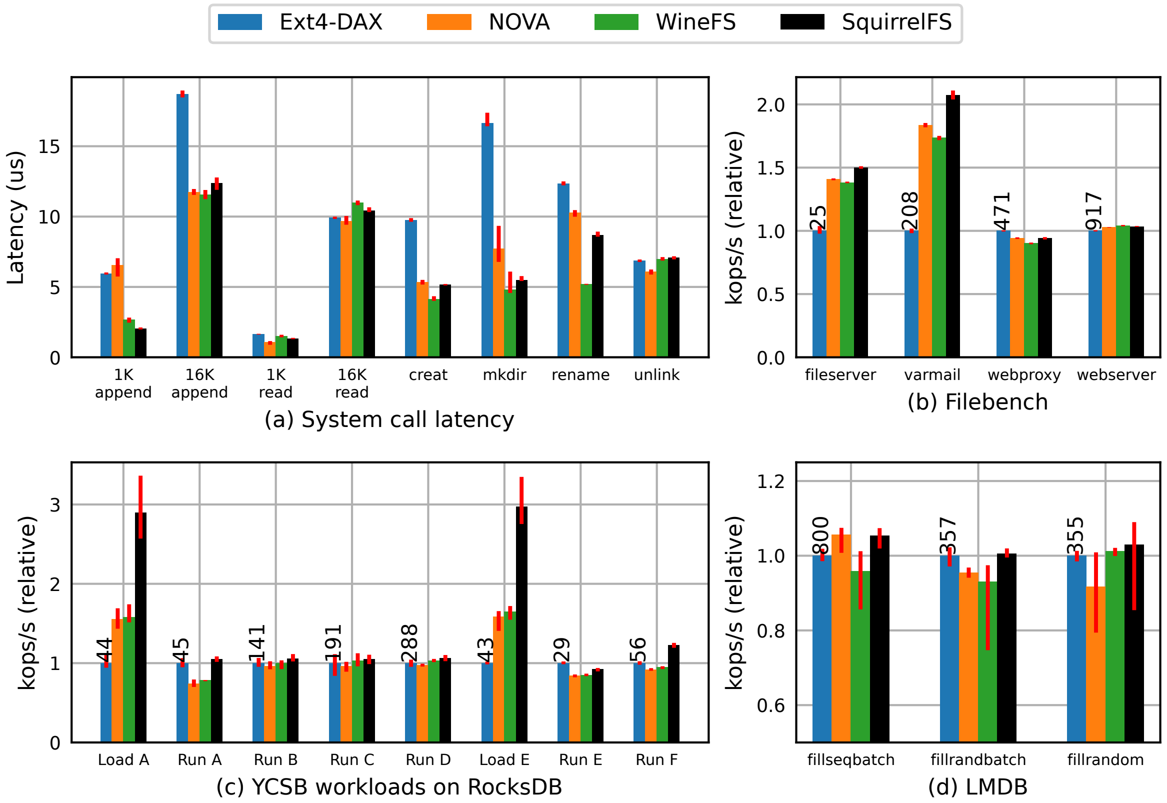


Figure 5: This figure shows the performance of the evaluated file systems on different benchmarks and applications. (a) shows absolute latency of different file system operations; (b), (c), and (d) show the relative throughput in kops/s of each system relative to Ext4-DAX on filebench, YCSB on RocksDB, and LMDB respectively.

system calls for all operations. Figure 5(c) shows throughput in kops/second relative to Ext4-DAX on each tested workload.

SQUIRRELFS outperforms the other systems on Loads A and E, which are 100% small inserts. As seen on the other benchmarks, SQUIRRELFS performs particularly well on small appends due to its lack of journaling or logging. Writes that require page allocation are particularly expensive in the other systems, as journaling/logging the new metadata incurs an additional 2-3us in NOVA and WineFS and 3-4us in Ext4-DAX. Ext4-DAX and NOVA both also journal or log metadata on every append, spending roughly 30% of each non-allocating call (approx 1-1.5us) managing journals/logs.

All file systems are within 10% of Ext4-DAX's throughput on Runs B, C, and D. All of these workloads are at least 95% small (4KB) reads, which all four systems achieve similar performance on.

SQUIRRELFS achieves the best throughput on Runs A and F, which are 50% reads and 50% updates (Run A) or read-modify-write operations (Run F). Ext4-DAX, NOVA, and WineFS all incur logging/journaling on these workloads; Ext4-DAX outperforms NOVA and WineFS because it has less journaling overhead for in-place updates and is more aware of data contiguity on reads.

Ext4-DAX achieves the best performance on Run E, which is 95% range scans and 5% inserts. Ext4-DAX's contiguity-awareness and better fragmentation-prevention mechanisms help it outperform the other systems on larger read operations.

LMDB. We also run LMDB [6], a memory-mapped database, using `db_bench`'s `fillseqbatch`, `fillrandbatch`, and `fillrand` workloads. Each experiment uses 100M keys on an empty file system. Figure 5(d) shows the throughput in kops/sec for each file system on each workload. Each file system has throughput with 12% of the other systems. Most updates are done to memory-mapped files, so differences in the performance of system calls and metadata management designs have a reduced impact.

Git. We also evaluate the performance of SQUIRRELFS by performing `git checkout` of major Linux kernel versions. The time to check out a given version in each file system is within 8% of the other systems.

5.5 Mount time

SQUIRRELFS takes longer to mount than other PM file sys-

	System state	Mount time (s)
Normal mount	mkfs	5.80
	Empty	5.51
	Full	30.50
Recovery mount	Empty	5.76
	Full	55.50

Table 2: Time in seconds to mount SQUIRRELFS file system images in differrent states. Times in the recovery mount column come from mounting a cleanly-unmounted file system that runs a recovery scan in addition to normal rebuild scans.

tems because it must rebuild volatile indices for the entire file system. Table 2 shows how long it takes to mount SQUIRRELFS on a 128GB PM device with different contents. The ≈ 5.5 seconds it takes to initialize or mount an empty system is the overhead of zeroing or scanning the metadata tables and creating volatile allocators. We also measure the time to mount a system with 100% data and inode utilization. Most of this time is spent allocating space for and managing the volatile indexes and allocators.

If SQUIRRELFS detects that it was not unmounted cleanly, it constructs additional structures to keep track of orphaned objects and the true link count of each inode. It fills in these structures during the regular rebuild scan and uses them to free orphans and correct link counts at the end of the mount process. SQUIRRELFS also checks each directory entry for non-null rename pointers and either rolls back or completes any interrupted renames. Table 2 reports the time it takes SQUIRRELFS to perform recovery scans on a cleanly-unmounted device. Mounting with recovery takes longer than a standard mount because the file system must construct orphan-tracking structures and do an extra iteration over all directories to check for rename pointers in addition to building the volatile indices and allocators.

SQUIRRELFS's mount time could be improved by parallelizing some of its rebuild and recovery logic. For example, the inode and page descriptor table scans are completely independent and could be done in parallel. The file system tree rebuild logic could also be distributed across multiple threads.

5.6 Resource usage

Compilation. SQUIRRELFS takes approximately 10 seconds to compile on our test machine, including typestate checking. This compares well to fully-verified systems; FSCQ [19] takes about 11 hours to verify, and VeriBetrKV [29] takes 1.8 hours (10 minutes when parallelized).

SQUIRRELFS also compiles faster than the other tested systems on the test machine. Table 3 shows the size of each system in lines of code and how long it takes to compile. SQUIRRELFS's more complicated typechecking does not noticeably impact its compilation time.

System	LOC	Compile time (s)
Ext4	45K	38
NOVA	16K	20
WineFS	9K	13
SQUIRRELFS	7.5K	10

Table 3: Time to compile different PM file systems as loadable kernel modules. Ext4's line count includes interleaved DAX and non-DAX code.

Memory. SQUIRRELFS maintains indexes for fast lookups of files and directory entries. Each regular file has an index mapping its inode number (8 bytes) to each of its pages and their offsets (16 bytes total). Thus, the index entries for a 1MB file use about 4KB of memory. Each directory has a similar inode to page index (without offsets), plus a mapping from directory entry names to metadata like their location on PM and inode number. The current maximum name length is 110 bytes (which makes directory entries 128 bytes) and SQUIRRELFS does not currently hash or compress names. Therefore, each directory entry takes up approximately 250 bytes in the index.

CPU. SQUIRRELFS does not start new threads in any of its operations. We leave the use of more threads for operations like freeing pages, running crash recovery, etc. to future work.

5.7 Correctness

Model checking. We check that a correctness invariant always holds in all traces of our Alloy model. We bound traces to include two operations (which may be concurrent), 10 persistent objects, and up to 30 steps. The invariant includes both sanity checks on the model as well as file system consistency checks. The sanity checks ensure, for example, that objects will never end up with conflicting typestates. The consistency checks ensure that 1) objects always have a legal link count, 2) there are no pointers to uninitialized objects, 3) freed objects do not contain pointers to other objects, and 4) there are no cycles of rename pointers and directory entries are pointed to by at most one rename pointer.

Testing. We test SQUIRRELFS using a set of handwritten tests and the xfstests [10] test suite. SQUIRRELFS currently passes all supported tests (67) from xfstests' `generic` test suite. The rest of the tests use system calls or arguments that are currently not supported by SQUIRRELFS.

Crash consistency. We used Chipmunk [41] to test SQUIRRELFS for crash-consistency bugs. We modified Chipmunk's test generators to remove several system calls that SQUIRRELFS does not currently support but otherwise ran its full suite of systematically-generated tests and fuzzed the system for approximately 24 hours. Chipmunk did not find any ordering-related crash-consistency bugs in SQUIRRELFS, providing evidence that typestate-checked SSU is an effective mechanism for preventing such bugs. Chipmunk did find four crash consistency bugs in unchecked parts of SQUIRRELFS code, three in its rebuilding of volatile data structures and one in the body of typestate transitions in which a cache line flush was issued to the wrong address. As these are not caused by incorrect update ordering, the typestate pattern did not catch them at compile time. We found that using the typestate pattern in SQUIRRELFS made locating and fixing these bugs faster and easier, as we could focus on the specific regions of code that are unchecked and are thus more likely to have bugs.

5.8 Summary

SQUIRRELFS provides comparable performance to other PM file systems, while providing strong guarantees about its crash consistency. Due to the innovative use of typestate checking, we were able to implement SSU and gain confidence in its correctness. SQUIRRELFS gains an advantage over other file systems in write-dominated workloads, since soft updates avoids writing to a log or to a second copy of the data. The design of SQUIRRELFS trades off good common-case performance for slightly longer mount times compared to other file systems; we believe this is acceptable since crashes are rare. SQUIRRELFS compiles at the same rate as other PM file systems, despite the strong type checking.

6 Related work

Rust for PM. SQUIRRELFS was inspired by Corundum [32]. Corundum builds data structures whose low-level properties are checked using Rust's type system. For example, Corundum ensures that there are no pointers to volatile memory stored in persistent memory, and that persistent state is only updated in transactions. It focuses on lower-level persistent memory programming errors and cannot prevent higher-level logical bugs. Corundum also requires *all* updates to PM to be in transactions, which is overly restrictive for many systems. In contrast to Corundum, SQUIRRELFS checks high-level file-system crash-consistency properties using type-checking without placing constraints on how the file system is used.

Soft updates for PM. Two PM file systems use soft updates for crash consistency: SoupFS [23] and ArckFS [60]. Unlike SQUIRRELFS, SoupFS is asynchronous and uses background threads to flush updates. It uses byte-addressable updates to eliminate cyclic dependencies. ArckFS is a user-space PM file system built on the Trio architecture that uses synchronous, soft-updates-esque updates for simple operations (e.g., creating a file) and undo journaling in more complicated cases. Unlike ArckFS, SQUIRRELFS uses only synchronous soft updates for its crash consistency; the novel way in which SQUIRRELFS implements atomic rename (without journaling or copy-on-write) further differentiates it from ArckFS. Both SoupFS and ArckFS are written in C, and do not use Rust's type system to check their crash consistency.

Storage systems in Rust. Bento [45] is a framework for building in-kernel file systems in Rust. The corresponding file system from the Bento project, BentoFS, was designed for block devices. Bento does not utilize the type system of Rust to check file-system properties.

ShardStore [15] is a Rust key-value store used in Amazon S3 that uses an asynchronous soft-updates-inspired crash-consistency mechanism. The rules for when something should be written to storage in ShardStore were checked with DepSynth [55], a tool for synthesizing soft updates dependency rules. Unlike ShardStore, SQUIRRELFS uses a synchronous version of soft updates, and provides higher-level primitives like atomic rename; ShardStore does not utilize the type system to perform higher-level checks.

7 Conclusion

This paper presents a new methodology for crash-consistent file system development. We propose the use of the typestate pattern in Rust to statically check crash-consistency invariants with low proof burden. We also introduce a novel crash-consistency mechanism, synchronous soft updates, that is well-suited to enforcement with the typestate pattern and that eliminates many challenges associated with the original soft updates technique. We develop SQUIRRELFS, a new file system for persistent memory that uses statically-checked synchronous soft updates for crash consistency. SQUIRRELFS achieves comparable or better performance than other PM file systems and required no language modifications or verification expertise to build. SQUIRRELFS, its Alloy model, and our Alloy utilities are available at https://github.com/utsaslab/squirrelfs.

Acknowledgments

We thank our anonymous shepherd, OSDI reviewers, and the members of SaSLab and LASR at UT Austin for their insightful comments and feedback. This work was supported by NSF CAREER #1751277, NSF CCF #2124044, and donations from Amazon, Toyota, and VMware.

References

[1] BTRFS documentation. https://btrfs.readthedocs.io/en/latest/.

[2] Compute Express Link (CXL) specification. https://www.computeexpresslink.org/download-the-specification.

[3] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[4] Filebench. https://sourceforge.net/projects/filebench/.

[5] Linux test project. https://linux-test-project.github.io/.

[6] LMDB. http://www.lmdb.tech.

[7] Rocksdb. https://rocksdb.org/.

[8] Rust for linux. https://rust-for-linux.com/.

[9] Typestate programming. https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html.

[10] xfstests. https://github.com/kdave/xfstests.

[11] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 175–188, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Valerie Aurora. Soft updates, hard problems. https://lwn.net/Articles/339337/, July 2009.

[14] Piotr Balcer. Exploring the Software Ecosystem for Compute Express Link (CXL) Memory. https://pmem.io/blog/2023/05/exploring-the-software-ecosystem-for-compute-express-link-cxl-memory/, May 2023.

[15] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 836–850, October 2021.

[16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 83–98, Atlanta, GA, USA, April 2016.

[17] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[18] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[21] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, July 2017. USENIX Association.

[24] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran,

and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 307–320, New York, NY, USA, 2007. Association for Computing Machinery.

[26] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association.

[27] Jon Gjenset. *Rust for Rustaceans*. No Starch Press, 2022.

[28] Robert B. Hagmann. Reimplementing the cedar file system using logging and group commit. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 155–162. ACM, 1987.

[29] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015.

[31] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*, pages 235–246. USENIX Association, 1994.

[32] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Daniel Jackson. *Software Abstractions*. The MIT Press, 2016.

[34] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

[37] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Ubuntu Bugs LaunchPad. Bug #317781: Ext4 Data Loss. `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all`.

[41] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 718–733, New York, NY, USA, 2023. Association for Computing Machinery.

[42] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

[43] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[44] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[45] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 65–79. USENIX Association, February 2021.

[46] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), apr 2019.

[47] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.

[48] Roger M. Needham, David K. Gifford, and Mike Schroeder. The cedar file system. *Communications of the ACM*, March 1988.

[49] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 2015.

[50] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 433–448. USENIX Association, 2014.

[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash consistency. *Commun. ACM*, 58(10):46–51, 2015.

[52] Andy Rudoff. Persistent memory programming. *;login:*, (42):34–40, 2017.

[53] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[54] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[55] Jacob Van Geffen, Xi Wang, Emina Torlak, and James Bornholt. Synthesis-Aided Crash Consistency for Storage Systems. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[56] Jian Xu, Juno Kim, Amir Saman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 427–439. ACM, 2019.

[57] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[58] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[59] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, page 273–287, USA, 2004. USENIX Association.

[60] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles*,

SOSP ’23, page 150–165, New York, NY, USA, 2023. Association for Computing Machinery.

High-throughput and Flexible Host Networking for Accelerated Computing

Athinagoras Skiadopoulos*[1] Zhiqiang Xie[1] Mark Zhao[1] Qizhe Cai[2] Saksham Agarwal[2]
Jacob Adelmann[3] David Ahern[3] Carlo Contavalli[3] Michael Goldflam[3] Vitaly Mayatskikh[3]
Raghu Raja†[4] Daniel Walton[3] Rachit Agarwal[2] Shrijeet Mukherjee[3] Christos Kozyrakis[1]

[1]*Stanford University* [2]*Cornell University* [3]*Enfabrica*

Abstract

Modern network hardware is able to meet the stringent bandwidth demands of applications like GPU-accelerated AI. However, existing host network stacks offer a hard tradeoff between performance (in terms of sustained throughput when compared to network hardware capacity) and flexibility (in terms of the ability to select, customize, and extend different network protocols).

This paper explores a clean-slate approach to simultaneously offer high performance and flexibility. We present a co-design of the NIC hardware and the software stack to achieve this. The key idea in our design is the physical separation of the data path (payload transfer between network and application buffers) and the control path (header processing and transport-layer decisions). The NIC enables a high-performance zero-copy data path, independent of the placement of the application (CPU, GPU, FPGA, or other accelerators). The software stack provides a flexible control path by enabling the integration of any network protocol, executing in any environment (in the kernel, in user space, or in an accelerator).

We implement and evaluate *ZeroNIC*, a prototype that combines an FPGA-based NIC with a software stack that integrates the Linux TCP protocol. We demonstrate that *ZeroNIC* achieves RDMA-like throughput while maintaining the benefits of robust protocols like TCP under various network perturbations. For instance, *ZeroNIC* enables a single TCP flow to saturate a $100 Gbps$ link while utilizing only 17% of a single CPU core. *ZeroNIC* improves NCCL and Redis throughput by 2.66× and 3.71×, respectively, over Linux TCP on a Mellanox ConnectX-6 NIC, without requiring application modifications.

1 Introduction

Modern datacenter applications, such as artificial intelligence (AI), data analytics, and distributed storage, are increasingly reliant on moving massive amounts of data over the network. As a result, datacenter operators are deploying systems capable of hundreds of Gbps of host networking. For instance, the latest NVIDIA DGX-B200 is capable of $3.2 Tbps$ of networking – $400 Gbps$ for each of the 8 GPUs [26]. As compute, memory, and link throughput continue to scale, driven by technologies such as accelerators [5,25,49], the *end-host* network stack is rapidly becoming a dominant bottleneck for these applications [12,13,72,95]. Therefore, the problem of designing host network stacks has come to the forefront.

Existing host network stacks offer a hard tradeoff between performance (in terms of sustained throughput when compared to network hardware capacity) and flexibility (in terms of the ability to select, customize, and extend different network protocols). On the one extreme, RDMA-based host network stacks [8, 34, 38, 54] are able to achieve high performance, but provide minimal to no flexibility. With network protocols baked into the hardware, adapting the protocol to better suit the needs of emerging applications or deployments is either not feasible or requires the time-consuming process of hardware modification. As a result, existing RDMA-based deployments remain fragile due to the possibility of head-of-line blocking, deadlocks, congestion spreading, and/or host congestion [1,2,44,45,63,65,74,96]. On the other extreme, the Linux network stack provides flexibility with a variety of time-tested protocols [4,11,14,39,47,68] and mechanisms that enable the incorporation of new protocols [2,10,13]. Unfortunately, the current Linux stack falls significantly short of exploiting the high-throughput capabilities of modern network hardware [12]. Recent host network stacks [72,89] offer operating points between these two extremes, but suffer from a similar performance-flexibility tradeoff.

We present a clean-slate co-design of the host network hardware and the software stack that simultaneously achieves high performance and flexibility. Our design's key driving

*Work partially done while interning at Enfabrica.

†Affiliated with Amazon Web Services, work done while at Enfabrica.

idea is the physical separation of the data path (payload transfer between network and application buffers) and the control path (header processing and transport-layer decisions) within the host. Specifically, our NIC hardware enables a *high-performance data path* between the network and the application. The NIC splits the headers from the payload, and directly transfers the payload from/to application buffers, without requiring any intermediate data copy (*zero-copy*). Our software stack enables a *flexible control path*. Users can plug in existing transport stacks, which operate on packets (sans payloads) as before, to make decisions on when to send data (e.g., congestion and flow control) and notify applications upon completion (e.g., acknowledging in-order byte streams). The software control stack orchestrates memory management and signaling between the NIC, the transport stack, and applications. Importantly, our design is independent of the location of application buffers (CPU, GPU, FPGA, or other accelerators) or the transport protocol's execution environment (in the kernel, in user space, or even in an accelerator).

The key challenge in realizing the physical separation of data and control paths is to maintain correct semantics (in-order, exactly-once data delivery) even in presence of network perturbations (data corruption, drops, replication, reordering, etc.). Our hardware implements the bookkeeping needed to correctly transfer incoming data to their designated memory destination even in the presence of network perturbations, while our software stack coordinates across the hardware and the application layer to maintain correct protocol semantics.

We demonstrate the benefits of our approach using an end-to-end prototype, *ZeroNIC*. Our prototype combines an FPGA-based NIC connecting to CPU and GPU memory, with a software stack integrating in-kernel Linux TCP. Our prototype realizes two APIs: the *libibverbs* API [67] used by current RDMA applications and a streaming API for general-purpose socket applications. For both APIs, *ZeroNIC* supports zero-copy data transfers between the NIC and application buffers in CPU or GPU memory. We evaluate *ZeroNIC* across a variety of workloads and network conditions. *ZeroNIC* achieves RDMA-level throughput with low CPU utilization. For instance, we show that *ZeroNIC* allows a single TCP flow to saturate a $100 Gbps$ link while utilizing only 17% of a single CPU hyperthread. In comparison, the Linux host network stack on a Mellanox ConnectX-6 NIC achieves at most $50 Gbps$ for a single TCP flow at 100% CPU utilization. We also demonstrate that *ZeroNIC* enables a high-performance zero-copy data path between GPU devices, achieving 2.66× higher throughput in NCCL benchmarks [24], NVIDIA's core AI networking library. Finally, we show that *ZeroNIC* benefits from the use of robust network protocols such as the TCP implementation in Linux. *ZeroNIC* maintains its performance under drops and fairness across flows.

To the best of our knowledge, our work is the first to support both send and receive-side zero-copy for reliable protocols like TCP with no constraints (e.g., MTU alignment, API modifications). It supports accelerator devices (e.g., GPUs) and enables protocol termination anywhere (e.g., CPU or control-plane accelerators) without limiting protocol semantics.

2 Motivation and Background

Our goal is to enable high-performance host networking regardless of the data destination (host or accelerator memory) and where/how the control plane is implemented. This flexibility allows the development and tuning of network protocols that improve fabric behavior and overall network efficiency as applications evolve and systems scale.

2.1 RDMA: Performant but Inflexible

Many network-intensive applications, such as AI training using GPUs, frequently use RDMA solutions such as InfiniBand (IB) [6] or RoCE [7]. RDMA solutions bypass the OS network stack and its CPU overheads by terminating the network protocol in specialized hardware and firmware in RDMA NICs (RNICs). RNICs enable high throughput by DMAing network payloads directly from/to application buffers in CPU or GPU memory using information encoded in send/receive requests.

The disadvantage of RDMA solutions is the lack of flexibility. To achieve high throughput, RDMA solutions typically required a lossless fabric such as IB, reliant on certified (short-distance) cabling and specialized switches. Such networks were forced to adopt a restrictive topology, avoiding over-subscription and adding many redundant paths to avoid drops [21,93]. To provide a similar quality-of-service on lossy fabrics, RoCE solutions have increasingly required secondary mechanisms to eliminate drops in the face of congestion, such as priority flow control (PFC) [46] and watchdogs [8, 38]. RoCE solutions still suffer from a host of well-documented challenges, such as end-host congestion [55], major performance degradation under unavoidable network perturbations (e.g., packet drops or reorderings) [44, 103], and excessive buffer requirements [44]. Addressing these challenges is arduous because RoCE's control path is explicitly tied to the implementation of the RNIC, requiring collaboration with and intervention by RNIC vendors. For example, Microsoft required support from its RNIC vendor to address livelocks caused by go-back-0 retransmission [38].

2.2 Kernel Networking: Flexible but Slow

The Linux network stack, built around the TCP/IP protocols, runs on a vast range of commodity hardware, supports diverse topologies, and can adapt to highly-variable network conditions and failures. Its resiliency stems from the fact that developers can optimize network protocol parameters including the congestion scheme and buffer sizes for the needs of emerging applications and deployments.

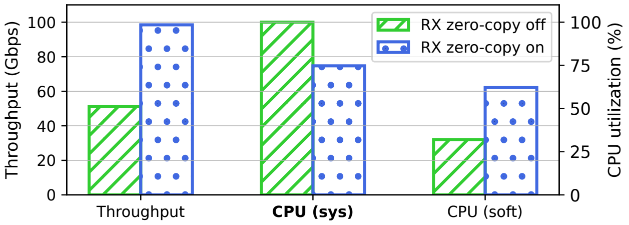


Figure 1: Throughput and receiver CPU utilization with and without receive-side (RX) emulated zero-copy. 100% means that a hyperthread is fully utilized. "CPU sys" refers to the hyperthread running protocol processing. "CPU soft" refers to the hyperthread running the application and the software interrupt handler.

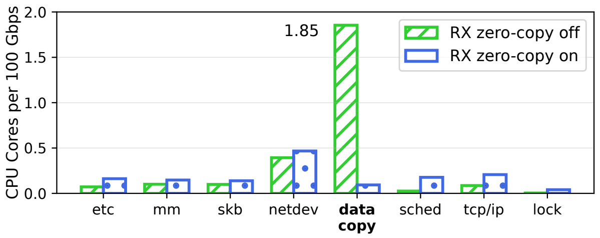


Figure 2: CPU breakdown for the TCP receiver with and without emulated RX zero-copy, normalized to the CPU utilization needed to achieve 100*Gbps* of throughput.

Unfortunately, the Linux stack cannot achieve high throughput ($\geq 100Gbps$) as single-thread CPU performance is a key bottleneck [12]. Specifically, receive-side data copies from kernel to application buffers dominate end-to-end performance, limiting a single TCP flow even after major optimizations (e.g., TSO/GRO, jumbo frames, and packet steering).

We built a proof-of-concept experiment to showcase the single-flow performance potential of removing the data copy. We modified the iperf benchmark [28] to support send-side zero-copy via the Linux sender ZC API [53]. We emulated receive-side zero-copy by truncating payloads in the kernel, avoiding the additional copy to user space [1]. We enabled TSO, GRO, and jumbo frames. We also pinned the iperf process and steered the receiver flow so that the interrupt handler (`soft`) and TCP processing (`sys`) are located in the two hyperthreads of the same physical core (sharing the L1 cache).

Figure 1 shows the sustained throughput and CPU utilization. Even with send-side zero-copy on, regular kernel networking can only achieve 50*Gbps* for a single flow. Similar to [12], we observe that CPU utilization is the bottleneck – specifically the TCP protocol processing receiver core (`sys`). Throughput cannot scale and the interrupt handling thread (`soft`) is underutilized. Figure 2 shows that the majority of CPU cycles for TCP processing are spent on data copies. Enabling receive zero-copy eliminates data copy overheads and drastically improves throughput, saturating the 100*Gbps* link. This experiment suggests that a flexible receiver zero-copy mechanism that copies data to application buffers in CPU, GPU, or storage devices can enable a wide range of protocols/stacks, including Linux TCP and other user space or hardware protocols/stacks [32, 48, 51, 72, 76, 77, 90], to meet the throughput requirements of network-intensive applications.

2.3 Towards Control & Data Path Separation

The core challenge with existing network solutions is the tight integration of the control and data paths, leading users to either integrate the data path into the kernel, sacrificing performance, or embed the control path in hardware, sacrificing flexibility. We propose the physical separation of these two paths. The data path provides robust support for zero-copy from NICs to application buffers on devices like CPUs and GPUs. The control path supports various transport protocols executing in software or hardware. This separation allows the control path to be optimized without overhauling the efficient data path.

There are many implementations of send-side (TX) zero-copy such as those in RDMA NICs, the MSG_ZEROCOPY flag in the Linux send system call [27], and the io_uring API for asynchronous I/O [18]. In contrast, existing *receive-side (RX) zero-copy* approaches are severely limited.

The challenge of page alignment. Linux includes a page-remapping mechanism for RX zero-copy in socket APIs [17, 59]. It allows the NIC to DMA the entire payload to a memory location and then remap the payload's physical address to the application buffer's virtual address at page granularity. This approach requires page-aligned payloads, making it difficult for applications to transmit arbitrary data lengths, as they can do with the socket or verbs APIs. The page-alignment requirement may also be incompatible with GPUs or flash devices [3], limiting the applicability of this approach. Moreover, page-remapping incurs high CPU overheads due to the need for TLB flushing after altering page table entries [59, 94].

The challenge of API compatibility. Several proposals facilitate RX zero-copy by altering application interfaces [9, 50, 79, 101, 102]. They require extensive changes to applications using common APIs like sockets or IB verbs, which typically rely on read/write operations from a contiguous buffer. These proposals asynchronously transfer packets from the NIC to application buffers, either as a linked list of scattered payloads or with headers and data interleaved in a buffer, which are released after being processed by the application. Hence, applications must adapt to handling non-contiguous data addresses during read operations.

The challenge of packet perturbations. Packet reordering, drops, and retransmissions disrupt the expected order of packet arrivals and complicate the correct copying of payloads into application buffers. A simple solution, employed by many

[1] Code available at https://github.com/enfabrica/iperf

RNICs, is to discard out-of-order packets and default to a go-back-N retransmission strategy, at the expense of throughput (§2.1). An alternative is to temporarily buffer out-of-order packets in the NIC until the missing packets arrive, potentially through a selective retransmission mechanism. This approach can quickly exhaust the SRAM capacity of state-of-the-art NICs [74, 96], especially in large bandwidth-delay-product (BDP) environments such as hosts with 400*Gbps*+ networking per GPU, and limits the effective rate at which data is transferred to the application.

The challenge of reliable protocols. Some systems sacrifice reliable transport semantics, directly copying incoming payloads to the next-available application buffer. This limits RX zero-copy support to unreliable protocols like UDP [15, 57]. Recent attempts to support reliable connections (RC) have constrained applicability. SRNIC [96] handles sequential and out-of-order packets via separate fast and slow data paths. IRN [74] requires the sender to explicitly define a receiver buffer identifier in the header. 1RMA [91] requires application involvement for managing ordering and handling failure recovery. Flor [62] separates the control and data paths for RDMA transports to reconcile the control path differences across different RNIC generations. However, Flor primarily supports unreliable connections (UC). To extend to reliable semantics, Flor uses an additional reliability sequence number in the RDMA work request and requires the sender and the receiver to establish a common chunk size for data transfers. Flor must dynamically tune the chunk size to tradeoff between high throughput (larger chunks) and managing congestion, drops, and retransmissions (smaller chunks).

Other related work. Nicmem [80], PayloadPark [36], and Ribosome [87] have recently explored separating the control and data paths in distinct contexts from our goals. They focus primarily on NFV (Network Function Virtualization) workloads that do not process payloads, but rather operate only on metadata to deliver packets to their next destination. To optimize resource usage such as PCIe traffic, they split packet headers and payloads and send only the header to the host. SplitRPC [56] uses a control and data path separation, but it is limited to unreliable protocols like UDP and use-cases like end-user requests/responses for AI inference. Our work tackles a broader range of applications that continuously process payloads and benefit from transport layer functionalities.

3 Performant and Flexible Host Networking

We co-design the hardware and software to *physically separate the data and control paths* in host networking, but *logically couple them after separation*. The physical separation enables a high-throughput, zero-copy data path to application buffers for payloads, and an independent control path for header processing. Figure 3 provides a high-level view of our approach. The data path connects to *any endpoint* (e.g., accelerators, storage, host memory, etc.), and the control path executes *arbitrary transport protocols* in *any execution environment* (in user or kernel space software on a CPU, in SmartNIC software or hardware, or even in a protocol accelerator), as illustrated by the two examples in Figure 4. The logical coupling allows the control path to have full control of protocol semantics, i.e., when data is correctly received or sent, how to handle events like reorderings and retransmissions, and when to notify the application – even if the transport protocol and data live in completely different devices.

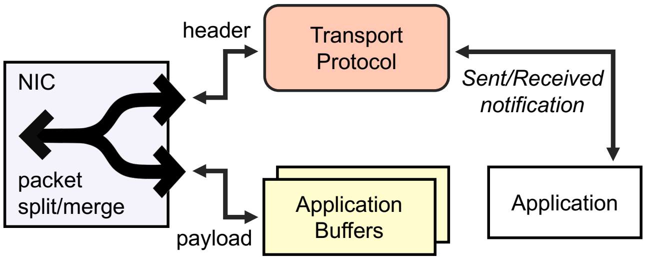
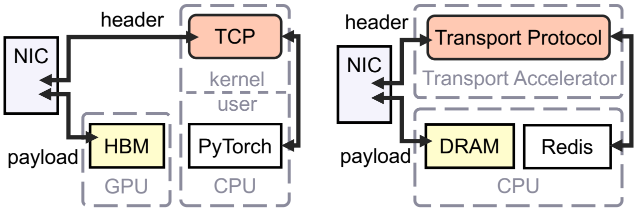


Figure 3: Host networking with physically separated control and data paths.



(a) Application buffers in GPU memory (HBM), protocol (in this case, TCP) in kernel space.

(b) Application buffers in CPU memory (host DRAM), protocol in a transport accelerator.

Figure 4: Examples of control and data path separation.

The key challenge in providing a zero-copy data path managed by transport protocols external to the NIC is that the NIC must decide *if, when,* and *where* to copy incoming payloads, *prior* to the transport protocol addressing out-of-order deliveries and retransmissions. Additionally, regardless of when data is copied, it should only be exposed to the application when protocol semantics allow (e.g., in-order delivery).

We begin by reviewing how packets travel throughout our network stack (§3.1). The NIC (§3.2) splits and merges headers and data to enable zero-copy data transfers directly to arbitrary devices (e.g. GPUs), even under reorderings, retransmissions, and drops. Our software stack is composed of the control stack and the provider library. The control stack (§3.3), which can execute in an arbitrary execution environment (e.g., in the kernel as a driver) is the coordinator between the NIC, an arbitrary transport protocol, and the application. The transport protocol acts only on packet headers, while the control stack proxies its actions to data in remote memory (i.e., NIC or application buffers). Our provider library implements both

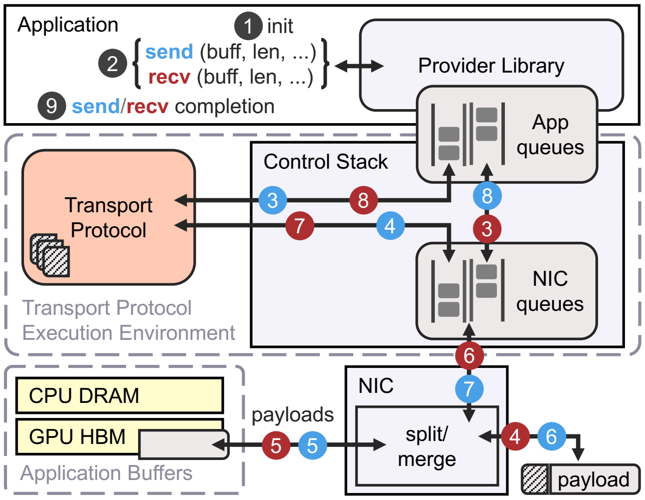


Figure 5: End-to-end send (blue) and receive (red) paths.

message and streaming APIs (§3.4) to allow an easy mapping of popular networking libraries onto our design. Finally, we discuss how we address various challenges in the host network stack such as retransmissions and compatibility with current optimizations (§3.5).

3.1 Receive and Send Path Overview

Figure 5 illustrates the receive (RX) and send (TX) paths through our stack.

Receive path. ① A receiving application begins by performing an initialization step using the provider library. As usual, this step establishes a connection and binds to a network interface. It also allocates application and NIC queues (§3.3) to coordinate between the control stack, the provider library, and the NIC hardware. The application also registers shared memory with the NIC for zero-copy transfers. These memory buffers can be anywhere in the system (e.g., GPU memory). Applications can periodically register (and deregister) shared memory space as needed. ② After initialization, the application invokes receive calls and the provider starts polling for completions. For every receive call, the provider enqueues an RX request entry into the application queue. RX request entries contain the receive call's buffer location and length. ③ The control stack steers the entry to the appropriate NIC queue. The NIC parses RX request entries to store application buffer information into dedicated hardware structures.

④ As packets arrive in the NIC from the sender, the NIC parses their headers and decides on dropping, buffering, or accepting each packet (§3.2). When a packet is accepted, the NIC splits it into the header and the payload. ⑤ The NIC identifies the payload's correct memory location in the designated device buffer and DMAs it accordingly. ⑥ The NIC creates and forwards RX header entries, composed of headers and metadata, to NIC queues leading to the control stack. ⑦

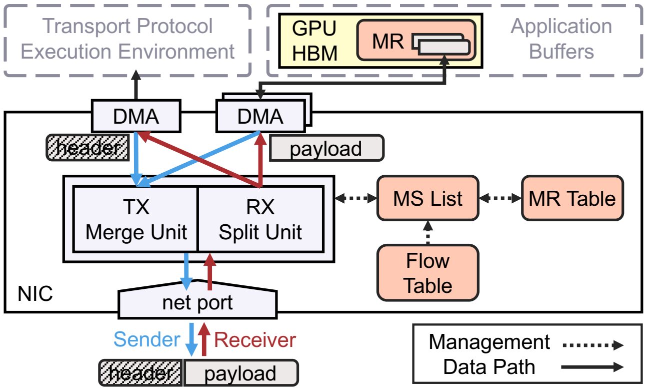


Figure 6: NIC hardware block diagram.

When the data DMA completes, the control stack forwards the headers from each RX header entry to the transport protocol. The protocol processes the header, reasoning about data acknowledgment (e.g., ACKing only in-order data). ⑧ When the protocol allows, the control stack posts a completion entry into the application queue.

Send path. The send path is also designed so that the transport protocol maintains control over data transmission. ① As with the receive path, the application begins with initialization steps that establish a connection, bind with a network device, and allocate and bind with the application and NIC queues needed for coordination. ② Upon a (non-blocking) send, the provider library enqueues a TX request entry to the application queue. The TX request entry contains the application buffer's location and length. ③ The control stack then forwards the entry to the transport protocol. ④ The transport protocol creates packet headers and allows progress according to its flow and congestion control mechanisms. When transmission is allowed, the control stack forwards the constructed header alongside the TX request entry to the NIC queue.

⑤ The NIC parses the TX request entries in-order and DMAs data from the application buffers directly into NIC memory. ⑥ The NIC then merges data with headers to form packets, optionally applying optimizations such as TSO, and transmits packets over the network. ⑦ Upon transmission, the NIC enqueues completion entries back to the NIC queue. ⑧ The control stack polls for completions and forwards them to the application queue. ⑨ Finally, the provider library polls for entries and notifies the application upon completion.

3.2 NIC Hardware Design

Figure 6 presents the NIC hardware design that implements key data structures to split (merge) packets, transfer headers and payloads to (from) the control stack and application, and track payload placement on a per-flow basis so that data can be zero-copied to their correct application buffer.

Memory management hardware data structures. The NIC

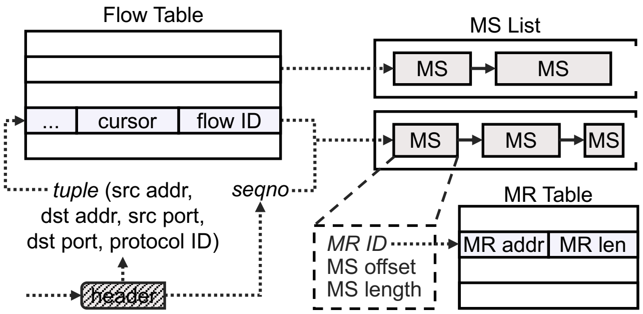


Figure 7: NIC hardware data structures.

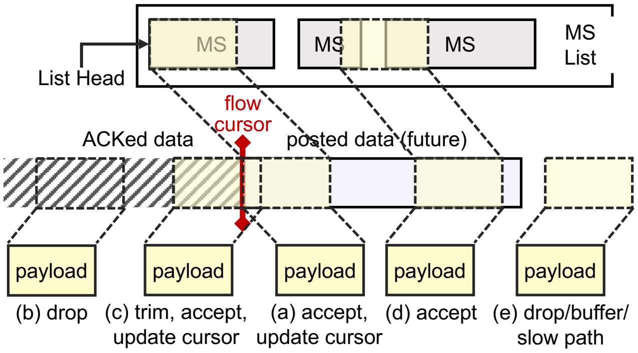


Figure 8: Flow cursor logic given packet arrival conditions.

implements the data structures shown in Figure 7 to track application buffers. At initialization, the application registers a set of *Memory Regions (MR)* using the provider library. An MR is a contiguous part of the application's virtual address space. The NIC maintains an MR Table entry for each registered MR. Virtual MR addresses are translated via an IOMMU in the NIC, which caches translations for efficiency. MRs can be anywhere in the system reachable by the IOMMU. For example, in the context of GPUs, CUDA allocates GPU memory, associates it to the PCIe address space (PCIe BAR), and maps it to host application buffers. In this case, our NIC IOMMU stores CPU to PCIe address space translations. Meanwhile, the NVIDIA driver translates between PCIe and GPU memory, as in GPUDirect.

The control stack creates a *Memory Segment (MS)* for each send and receive operation. An MS corresponds to a contiguous user buffer and is defined by its MR ID, its offset within the MR, and its length[2]. As the application makes asynchronous send and receive calls, the control stack enqueues the RX and TX MSs to the NIC into a per-flow *MS List*. The MS List is essentially a linked-list containing the application buffers involved with pending requests. Note that each flow (and thus MS List) maps to a distinct NIC queue in the control stack.

The *Flow Table* tracks flow metadata used for incoming packets. The most important fields in each *Flow Table* entry are the flow ID used to index to the flow's corresponding MS List, and the *flow cursor*. The flow cursor is the sequence number corresponding to the last in-order consumed packet in the flow. The MS List and the flow cursor are combined to make decisions over the packet's payload, as explained below.

Receiving a packet. The hardware structures described above allow the NIC to map incoming packets to application buffers for zero-copy DMA, as shown in Figure 6. We focus on the process of handling reliable protocols. Unreliable protocols simply land data in the next available MS.

The NIC parses the header of an incoming packet to construct a tuple that indexes the Flow Table (Figure 7), and obtain the corresponding MS List and flow cursor. The MS List and flow cursor are combined with the header's sequence number to derive the packet's position in the flow. The NIC then decides one of four actions: *accept*, *drop*, *buffer*, or *defer*.

We begin with Case (a) in Figure 8. In the absence of network perturbations, the arriving packet contains the next in-order unconsumed payload that the flow is expecting, i.e., the data immediately following the flow cursor. The packet is accepted: the hardware examines the top MS in the flow's MS List and uses the MS and MR information to derive the application address to DMA (zero-copy) the payload. The cursor is updated to reflect the next unconsumed position. MS boundaries and the size of the payload do not have to align. The packet's payload may consume a fraction of the MS or may need to span into the next MS in the list. Fully consumed MSs are retired when the flow cursor passes them.

Case (b) receives a packet with a sequence number that suggests all the bytes in the payload have been previously received and ACKed according to the flow cursor. This may be due to a re-transmission when an ACK is lost or delayed. This packet is dropped and no further action is taken. Case (c) receives a packet that includes some bytes that are previously ACKed and some new bytes. The hardware drops the repeated part and accepts the rest of the packet as in Case (a).

Case (d) receives a packet beyond the flow cursor (i.e., with a hole). This may be the result of packet reordering or a drop of an earlier packet. The hardware will walk the MS List, and by accumulating MS lengths, it will identify the right segment for the data. The data will be accepted and DMAed to the proper application buffer address. Even if future MSs are fully filled, they will not be retired until the cursor passes them. The control stack periodically sends the latest acknowledged byte to the NIC to update the cursor.

Finally, under rare conditions, a packet may not match any MS (Case (e)). This may be the result of excessive drops or the receiver posting buffers at a slow pace. We can drop the packet, buffer it in NIC memory and retry later, or defer the packet to a non-zero-copy path. We implemented the last option (defer) in our prototype system (§4). We also use this approach if a packet arrives for a flow that has no Flow Table

[2] For simplicity of presentation, we assume contiguous Memory Segments, although MSs may map to more complex data structures.

entry (i.e., the table has reached its capacity limits).

When a packet is accepted, it is passed to the Split/Merge unit shown in Figure 6, which splits the packet into header and payload. A DMA engine copies payload data directly to application memory (e.g., in CPU user space or GPU device memory). Another DMA engine then forwards headers to the control path address space (e.g., in CPU kernel memory).

Note that the NIC lands data in application memory before any transport processing happens. It also allows the overwrite of future data if a packet is transmitted multiple times since their MSs are not retired. However, correctness is maintained because the application is notified when it is safe to use its buffer by the control path, after protocol processing is done.

Sending a packet. The send hardware path is simple, as packets are sent in the order of requests. Upon receiving a header from the control stack, MSs enter the MS List and are handled by hardware in FIFO order. A DMA engine copies the payload directly from the corresponding application buffer. The Split/Merge unit merges the header and payload to form a packet, which is transmitted over the network.

Hardware requirements for scalability. Maintaining per-flow state raises scalability concerns. Our design supports 10K high-performance flows with ~10MB of NIC memory.

Most proposed structures have a low memory footprint. To support 10K flows, the Flow Table and MR Table require ~700KB and ~100KB respectively, to store all necessary metadata. MS Lists are the most resource-intensive structures. To support long, potentially out-of-order, packet runs with low memory footprint, our NIC does not buffer payloads. Instead, the NIC DMAs future payloads to their correct memory destination by finding the correct MS. For maximum efficiency, we allocate a minimum number of MS List entries per flow to keep the flow pipeline humming, and pull additional MS List entries as needed (a CIR/PIR – committed/peak information rate system) [43].

For example, a large bandwidth-delay-product (BDP) of $100Gbps \cdot 0.2ms = 2.5MB$ would require $\frac{2.5MB}{4KB} = 625$ MSs. Instead of allocating 625 entries for all 10K MS Lists ($10K \cdot 625 \cdot 8B = 50MB$), we allocate a minimum of 128 committed entries to each MS List, while supporting thousands of peak entries (e.g. 8K) that are allocated to flows on demand from a large entry backing store (e.g. 1M entries). For 10K high-performance flows, the total buffer requirement is $\max(1M \cdot 8B, 10K \cdot 128 \cdot 8B) = 9.77MB$. Downsizing MS Lists adds the additional requirement to buffer RX NIC requests during the lifetime of their respective MSs. Besides supporting thousands of zero-copy flows, our design additionally supports non-zero-copy flows that do not occupy the newly proposed data structures.

The required hardware resources for our NIC are significantly lower than those of most RNICs [96]. Modern SmartNICs also require several processor cores, tens of MBs of processor caches, and external memory like DDR, LPDDR, or HBM that can handle payload buffering for high BDPs.

3.3 Control Stack Design

The goal of the control stack is to enable an arbitrary transport protocol with our zero-copy data path, while maintaining efficiency and correctness. The control stack does so by separating and defining a clean interface between three components: *a) the application, b) the transport protocol,* and *c) the NIC*. The control stack maintains connections between each application and the NIC using two sets of queues.

Application queues. The control stack is co-located with the transport protocol [3], *application queues* are allocated in shared memory between the control stack and the application, and connect the provider library with the control stack. Each set of application queues contains a send queue, a receive queue, and their respective completion queues. TX and RX requests are enqueued by the provider library into the send and receive queues, respectively, while the control stack notifies applications upon completions via the completion queue.

NIC queues. The control stack also establishes a set of *NIC queues* for TX and RX requests, incoming RX header entries, and completions. NIC queues connect the control stack to the NIC. They are implemented in the control stack and are accessed by the NIC via DMA. In the send direction, the control stack enqueues MSs and headers to the NIC, constructed from TX requests enabled by the transport protocol. In the receive direction, the control stack sends MSs from RX requests to the NIC. As they are consumed by incoming data, the corresponding headers are split from incoming packets to form RX header entries directed to the control stack.

Supporting arbitrary transport protocols. Current solutions that leverage a single queue pair to provide zero-copy functionality struggle to support protocols not executing in either end of the queue (in the NIC or in user space). In contrast, our control stack uses two separate sets of queues to interpose the transport protocol between the application and NIC. The control stack polls the application send queue for requests and the NIC queue for receive-path headers and invokes the transport protocol to generate send-path headers and acknowledgments, respectively. The control stack can support arbitrary transport protocols by translating application requests to the respective transport API (e.g., TCP sockets).

Enhancing efficiency. In addition to eliminating data copies, the control stack benefits from reduced system call and interrupt overheads when submitting work and receiving completions. Specifically, polling on application queues avoids system calls, resulting in performance benefits similar to mechanisms such as `io_uring` [18]. Since the control stack is co-located with the transport protocol, it directly invokes it without system calls. Similarly, the control stack polls the NIC for completions and headers, avoiding software interrupts. To address applications with sparse communication, mechanisms such as combining polling and doorbells can also be applied.

[3] We assume the control stack executes as a kernel module; §6 discusses supporting transport protocols external to the kernel (e.g., in user space).

Maintaining correctness. Finally, the control stack maintains correctness by logically coupling the physically separated control and data paths. On the sender side, the control stack invokes the protocol's flow and congestion control to enqueue control entries in the NIC and to trigger data DMAs. This is equivalent to the congestion control algorithm acting on headers physically accompanied by their data. On the receiver side, payloads are separated from their headers in the NIC and DMAed directly to application buffers. RX header entries sent to the control stack incorporate information (e.g. sequence number) to bind to their corresponding data. Data becomes visible to the user upon consulting the protocol's acknowledgment policy (e.g., in-order delivery). Thus, the transport protocol maintains ownership of the data without ever touching the data itself, allowing us to reap all the benefits of current transport protocols (robustness, fairness, etc.).

3.4 API Design

The primary goal of our API is to allow current applications (more precisely, current networking libraries) to use our network stack with minimal effort. Current applications use either *message-based* or *streaming* semantics. Message semantics (e.g., RDMA verbs) require the network stack to deliver messages corresponding to contiguous memory buffers. A message size is well-defined by the side initializing communication (one- or two- sided). Streaming interfaces (e.g., sockets) allow senders to continuously transmit byte streams of arbitrary length. The receiver can keep invoking receive calls to consume data in the stream as the network stack progressively signals reception on a byte-stream basis. The stream memory layout can be irregular (non-contiguous) and different on the sender and receiver sides. Our design explicitly supports *both* message-based and streaming semantics. We implement the *libibverbs* API [67] and a socket-like interface.

Supporting message interfaces. Most high-performance applications rely on message semantics [24, 33, 37, 71, 73]. We support their transparent interoperability by implementing the libibverbs API. We dynamically link the libibverbs `verbs_context_ops` to our provider library. The provider in turn connects to our control stack and exposes our application queues to the user as `struct ibv_qp`.

Supporting streaming interfaces. Our design also supports streaming applications by exposing a socket-like API, with slight modifications to support our software stack. The application performs initialization similar to *libibverbs* (find a device, allocate a protection domain for memory regions, and initialize queues). Connection is established via the ordinary socket API (not requiring our fast data path). The above structures are wrapped in a `struct comm_ctx`. `send` and `recv` calls are asynchronous and extended with an argument containing the `comm_ctx`. To relieve the responsibility of registering and de-registering memory regions from the application, our `send` and `recv` calls post their buffer argument as an MR on their invocation. MRs can reside within any endpoint.

3.5 Addressing Challenges

Retransmissions. Section 3.2 explains how the NIC chooses the correct MSs, including when packets are retransmitted. However, in the presence of potential retransmissions, an already consumed MS may need to be reused multiple times. Both TX and RX sides post buffers that ultimately create MSs which must therefore be carefully retired or replenished.

On the sender side, the control stack clones the socket buffer (containing only metadata), before sending it to the transport stack and keeps it alive until the protocol receives an acknowledgment. If the transport decides on retransmitting the packet, the socket buffer is cloned again. The hardware will create the same MS and the retransmission will be accommodated. On the receiver side, the NIC only retires MSs directly following the flow cursor (§3.2), allowing overwrites of future retransmitted data. Permitting overwrites simplifies our retransmission handling logic, especially when arriving packets contain both new and previously delivered data.

Multiple flows. Multiplexing flows in the same NIC queues or MS Lists creates significant complexity in tracking which flow is served on each access. We bypass this issue by assigning NIC queues on a per-flow basis. Before binding with a NIC queue, the control stack creates a flow entry rule in the NIC Flow Table. The unique *flow ID* is used to index NIC queues and MS Lists, ensuring exclusivity. Hence, zero-copy flows are limited to the number of queues supported by the NIC. Despite this issue, we support large enough flow counts with moderate resource requirements (§3.2).

Associating messages with application buffers. Our design supports message semantics with a streaming protocol underneath. In contrast to streams, messages do not have to fully consume a user buffer before using the next one. Thus, there is no clear signal to determine if an incoming packet is the continuation of the currently served message (and MS) or refers to the next message (and MS). We address this issue by adding a message sequence number within the packet transport header (e.g. in the "options" field for TCP). Combined with the stream sequence number, we can point to the correct MS and retire previous MSs that can be consumed even if they were not fully filled.

Compatibility with offload mechanisms. Popular offload optimizations such as GRO/LRO (receiver) and TSO (sender) are compatible with our design. The control stack can transparently support software offload mechanisms like GRO; consecutive headers will be merged into a single socket buffer while their payloads have already been DMAed to consecutive Memory Segments. The user is notified about the latest in-order data, as usual. Similarly for LRO, headers are combined in the NIC after they are split from their payloads. For TSO, to support headers corresponding to more than an MSS (maximum segment size), the NIC segments them into smaller headers. The MS in the send request will now serve for multiple DMAs, one for each segmented header.

4 Implementation

We implemented *ZeroNIC*, an end-to-end prototype of our proposed design. *ZeroNIC* consists of an FPGA-based NIC that implements the key hardware functionality (§3.2), a software control stack that uses TCP as the transport protocol (§3.3), and a provider library exposing the API (§3.4).

NIC. We built a 100*Gbps* Ethernet NIC prototype using a commodity Xilinx Virtex UltraScale+ FPGA [97]. The NIC has three x16 PCIe 3.0 links that connect to an x86 CPU socket and two NVIDIA GPUs (Quadro RTX 4000). In essence, our NIC also acts as a switch between the CPU and the GPUs.

The NIC-to-CPU link is controlled by the QDMA IP [99] from Xilinx that presents the NIC as an endpoint device to the CPU. The CPU is the PCIe root port device. The QDMA block allows for DMA transfers in both directions at full PCIe bandwidth (100*Gbps*). Each NIC-to-GPU link is controlled by the Xilinx XDMA IP [98] that presents the NIC as the root port device to the GPU. The GPU is a PCIe endpoint device. Unfortunately, under this configuration the XDMA block supports a limited number of outstanding PCIe transactions. This imposes a hardware limit on the sustained PCIe bandwidth for DMA transfers between the NIC and the GPU. When moving data from the GPU to the FPGA (GPU is the sender), the maximum PCIe bandwidth is 85.0*Gbps*. When moving data from the FPGA to the NIC (GPU is the receiver), the maximum PCIe bandwidth is 38.6*Gbps*. This limitation of our FPGA system and IP blocks is not fundamental to our design. An ASIC implementation of our NIC would saturate available bandwidth for transfers to GPUs.

Our NIC implements a split/merge unit for the 100*Gbps* Ethernet port. We use context-addressable memories to implement the MR Table and the Flow Table that is addressed by the 5-tuple from the TCP/IP header (source and destination addresses and ports, and protocol ID). The split-merge unit connects to the PCIe ports. The *ZeroNIC* design is modular and can be extended to support multiple 100*Gbps* Ethernet ports using replicated split/merge units. It can also support more root-ports in order to connect more than two GPUs.

Control Stack. We implemented the control stack as a Linux kernel driver, which binds the provider library with the NIC. The driver directly invokes the *unmodified Linux kernel TCP stack* for protocol processing, translating application requests and NIC queue entries into Linux TCP socket calls. While our design supports arbitrary protocols and execution locations, we select the kernel TCP protocol for the first prototype as it is robust, but challenging to make performant (see §2.2). Application queues live in shared memory between the kernel and provider library. NIC queues live in the kernel's virtual network device. Both are implemented as ring buffers of user-configurable sized entries.

Table 1: Evaluation system setup.

System	TCP / RoCE Baselines	*ZeroNIC*
NIC	Mellanox ConnectX-6	Prototype built on Xilinx Virtex UltraScale+
Topology	2-node direct-conn 100G eth	2-node direct-conn 100G eth (38.6G max for GPU)
Protocol	TCP bbr / RoCEv2 RC	TCP bbr
Setup	TCP: TSO, LRO, 9K MTU RoCE: 4K MTU	TSO, GRO, 9K MTU (always TCP)
CPU	32 core AMD EPYC 7502 L1,L2,L3: 2MB,16MB,128MB	32 core AMD EPYC 7502 L1,L2,L3: 2MB,16MB,128MB

5 Evaluation

We evaluate the efficiency of our host networking approach using the *ZeroNIC* prototype, with the Linux kernel's TCP transport in our control stack. We compare the performance of *ZeroNIC* against two popular baselines: a TCP baseline that uses the Linux network stack without our high-performance data path, and a RoCE baseline that terminates the transport protocol in the NIC. Both baselines use a Mellanox ConnectX-6 NIC. We summarize the specific configurations of these systems in Table 1. All *ZeroNIC* measurements utilize large-segment offloading (TSO and GRO) and jumbo frames, unless otherwise specified. We do not require MTUs to be page-aligned.

5.1 *ZeroNIC* Throughput Evaluation

***ZeroNIC* provides RDMA-level throughput to application buffers in CPU memory at low CPU utilization.** Table 2 shows the throughput achieved by *ZeroNIC* for a single flow between a sender and a receiver application using CPU memory. We compare against the Mellanox RoCE baseline (MLX RoCE), as well as against kernel TCP using the Mellanox NIC (MLX TCP) with and without send-side zero-copy (TX ZC on/off). We enabled send-side zero-copy for Mellanox TCP as discussed in §2.2. For *ZeroNIC* and Mellanox TCP, we pin the protocol processing thread, and either the queue polling (for *ZeroNIC*) or the software interrupt handling (for Mellanox TCP) thread to hyperthreads in the same physical core to maximize cache locality.

Table 2 breaks down the receiver-side CPU utilization between the kernel (`sys`) and other CPU cycles (`usr/soft`). `sys` includes protocol processing and the *ZeroNIC* driver, while `usr/soft` includes the *ZeroNIC* provider library, interrupts, and the application itself. Note that 100% CPU utilization means that *a single CPU hyperthread* (2 per core) is fully utilized. RoCE offloads protocol processing to the RNIC and

Table 2: Throughput and receiver-side CPU utilization for CPU-to-CPU transfers. "CPU sys" refers to the hyperthread running protocol processing and *ZeroNIC*'s driver. "CPU usr/soft" refers to the hyperthread running the application, software interrupt handler, and *ZeroNIC*'s provider library.

System	Throughput (Gbps)	CPU sys (%)	CPU usr/soft (%)	Estimated max Tput
MLX TCP TX ZC off	43.89 ± 1.35	94.15 ± 3.45	29.55 ± 2.62	46.61
MLX TCP TX ZC on	50.63 ± 0.55	100.0 ± 0.00	32.36 ± 0.80	50.63
MLX RoCE	98.03 ± 0.00	N/A	9.58 ± 0.81	N/A
ZeroNIC	96.37 ± 0.60	17.20 ± 1.96	33.50 ± 1.11	560.29

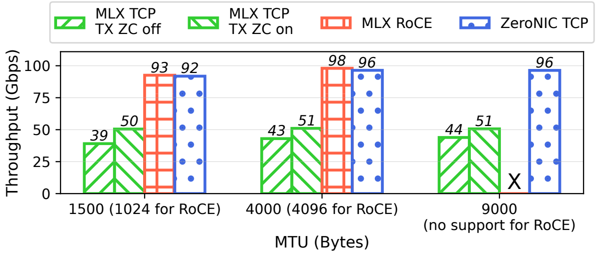


Figure 9: Achieved throughput across multiple MTU sizes. RoCE only supports power-of-two MTU sizes up to 4096B.

lands data directly into application buffers. Thus, we do not observe `sys` CPU utilization. For *ZeroNIC* and RoCE, the throughput benchmark polls for completions. We do not include cycles spent on the polling loop for *ZeroNIC* and RoCE, as cycles spent polling do not limit throughput (e.g. the RoCE application hyperthread shows as 100% utilized).

We observe that Mellanox TCP is constrained by CPU utilization, despite the optimizations used (TSO, LRO, 9K MTU). The Linux TCP stack uses a single thread for protocol processing for each flow. With send-side zero-copy disabled, the sender-side protocol processing thread saturates (not shown), while the receiver thread almost reaches full utilization at 43*Gbps*. Enabling sender zero-copy exposes the receive-side bottleneck as the receiver thread saturates at 50*Gbps*.

ZeroNIC copies RX data directly to user space application buffers, eliminating the CPU cycles spent on data copy as shown in Figure 2. This reduces the protocol processing thread's CPU utilization from 100% at 50.63*Gbps* to 17.20% at 96.37*Gbps*. *ZeroNIC* also eliminates the majority of system calls via the control stack's polling architecture (§3.3), achieving an even lower `usr/soft` utilization than what baseline TCP is projected to need at 100*Gbps* (§3.3, Figure 1). This allows *ZeroNIC* to reach a throughput comparable to RoCE. However, *ZeroNIC* maintains the flexibility of the Linux stack, while RoCE implements its entire control path in the RNIC.

Table 3: Achieved throughput and receiver-side CPU utilization for communication across different CPU/GPU endpoints.

System	Throughput (Gbps)	CPU sys (%)	CPU usr/soft (%)	Estimated max Tput
ZeroNIC CPU-CPU	96.37 ± 0.60	17.20 ± 1.96	33.50 ± 1.11	560.29
ZeroNIC CPU-GPU	84.78 ± 0.41 [4]	16.31 ± 0.54	36.33 ± 2.21	519.80
ZeroNIC GPU-GPU	38.59 ± 0.07 [4]	9.12 ± 0.21	32.50 ± 2.07	423.14

ZeroNIC is now bound by the link capacity. Given additional or faster links, *ZeroNIC* can scale beyond 100*Gbps*. The last column in Table 2 estimates the maximum throughput that *ZeroNIC* can achieve with the kernel TCP stack, by scaling the protocol processing thread (`sys`) to saturate CPU utilization (indeed, as we will see in Table 3, the `usr/soft` thread has minor variations for different peak bandwidth settings). *ZeroNIC* is projected to scale to > 500*Gbps* for a single flow of the kernel TCP stack. This is a 11× higher throughput than the current TCP network stack achieves for a single flow using the Mellanox NIC.

Finally, Figure 9 demonstrates that *ZeroNIC*'s benefits hold across various MTU sizes. For smaller MTUs (1500 or 1024 bytes), throughput on both *ZeroNIC* and RoCE slightly reduces due to higher packets-per-second DMA overheads.

***ZeroNIC* enables high-throughput data transfers directly to device (GPU) memory.** *ZeroNIC* is able to extend zero-copy benefits to arbitrary endpoints, including GPUs. Hence, *ZeroNIC* can directly transfer data from and to GPU HBM, bypassing the host CPU memory, similar to GPUDirect [22]. Table 3 presents *ZeroNIC*'s single-flow throughput for CPU-to-CPU, CPU-to-GPU, and GPU-to-GPU communication. In all cases, the control path uses the Linux TCP stack. *ZeroNIC* is able to saturate the bandwidth supported by the hardware on all paths, given the prototype IP limitations discussed in Section 4: ~100*Gbps* for CPU-to-CPU, 85*Gbps* for CPU-to-GPU, and 38.6*Gbps* for GPU-to-GPU transfers.

To validate that the *ZeroNIC* design scales to higher throughput in the absence of prototype limitations, Table 3 also reports CPU utilization. As in Table 2, we split CPU utilization between protocol processing and driver (`sys`) and other cycles (`usr/soft`). As we can see by comparing the CPU-to-CPU and GPU-to-GPU results, the `usr/soft` CPU cycles do not strongly scale with maximum throughput. The limiting factor for higher throughput for a single flow would be the protocol processing overheads of the kernel's TCP

[4] This is the maximum throughput supported by our hardware prototype due to FPGA IP limitations (Section 4).

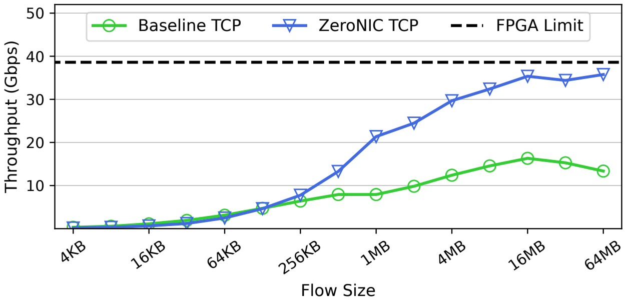


Figure 10: NCCL throughput using *ZeroNIC* across different flow sizes.

stack. Hence, we calculate the maximum throughput *ZeroNIC* can reach when the kernel thread saturates (100% utilization) to be above 400*Gbps* for CPU-to-CPU, CPU-to-GPU, and GPU-to-GPU flows, This means that *ZeroNIC* can replace the RoCE back-end network in GPU-based AI clusters that is used to communicate activations and gradients during AI training [75]. A *ZeroNIC*-based system with 8 GPUs would require 8 CPU hyperthreads to support a total of 3.2*Tbps* of GPU-to-GPU networking, while gaining the benefits of using any network protocol, such as the robustness of TCP.

5.2 End-to-End Workloads

ZeroNIC supports popular APIs (§ 3.4) that enable application integration *without modifications*, simply by linking against our provider library. We demonstrate this using three important workloads; NCCL benchmarks, PyTorch, and Redis.

NCCL. NCCL [24] is the dominant communication library for distributed AI using GPUs. It implements and optimizes collective communication primitives that are commonly used in AI training and multi-GPU inference. Different phases of inter-node collective communication (all-reduce in data parallelism, all-to-all in expert parallelism, point-to-point in pipeline parallelism, etc.) use tens of megabytes as their collective bucket size [58, 64, 85]. The number of flows scales with the number of nodes (N). For example, NCCL's tree algorithm, the predominant inter-node collective implementation algorithm, creates $2\log N$ flows per node [19]. Exposed communication increases with system size [88], making network performance critical, especially as cluster sizes increase beyond 10,000 GPUs and 1,000 nodes [20]. Improving NCCL performance directly reduces exposed communication, leading to faster AI training and inference [41, 70, 82, 100].

We ran the broadcast NCCL benchmark [23]. For 2 nodes, broadcast sends the full collective size unidirectionally between two *ZeroNIC* GPU servers. Since broadcast is the core primitive used to build other collectives, improved broadcast throughput directly translates to higher collective throughput in general. We compare *ZeroNIC* to a baseline TCP implementation which uses the *ZeroNIC* NIC hardware, but always forwards the entire packet directly to the unmodified Linux network stack (no zero-copy).

Table 4: Average training epoch latency (in seconds) for different PyTorch distributed data parallel models using RoCE GPUDirect and *ZeroNIC*.

System	ResNet50	ResNet101	ResNet152
MLX RoCE	3.52 ± 0.04	6.12 ± 0.07	8.80 ± 0.04
ZeroNIC	3.57 ± 0.02	6.22 ± 0.08	8.83 ± 0.08

Figure 10 shows the throughput achieved by NCCL as we vary the collective size. For small sizes, *ZeroNIC* and the baseline deliver the same throughput. The throughput bottleneck for small collectives is actually NCCL itself. It implements a higher-level protocol with significant processing overheads that cannot saturate the link with a single flow for small collectives, regardless of whether RDMA or TCP is used. As the collective size increases, the bottleneck becomes packet processing in the TCP stack. For the baseline TCP (no zero-copy), NCCL saturates at ~16*Gbps* for flows beyond 16*MB*. For large collective sizes, *ZeroNIC* manages to hit the maximum throughput allowed by our FPGA prototype, 2.66× higher than the baseline. If the FPGA limitation is removed, *ZeroNIC* will saturate the Ethernet link. These results show that the *ZeroNIC* data path is especially powerful for devices such as GPUs. It eliminates two data copies: a copy from the kernel buffer to the application buffer and a copy from the CPU-based application buffer to a GPU buffer.

PyTorch. PyTorch [78] is the most popular AI framework. For distributed training, PyTorch implements various parallelization strategies, leveraging communication backends such as NCCL. For example, in data parallelism, training data is partitioned while each node holds a full copy of the model. During each iteration's backward pass, all model gradients are averaged across all ranks using the all-reduce collective.

We trained different sizes of ResNet [42] using PyTorch's distributed data parallelism [81] with NCCL. We compared the average training epoch latency on two *ZeroNIC* nodes using TCP, against two Mellanox nodes using RoCE with GPUDirect [22]. Our baselines, ResNet50, ResNet101, and ResNet152 are composed of 25.6, 45.5, and 60.2 million parameters, and require synchronizing 51.2, 91.0, and 120.4 MBs worth of gradients in every iteration, respectively. Each epoch is composed of 100 iterations. Table 4 shows that *ZeroNIC* achieves GPUDirect-level performance, within 2% of RoCE's latency.

As NCCL supports the IB verbs API, we ran both the PyTorch and NCCL experiments on *ZeroNIC* without any application/library modifications. These results demonstrate that our design can be effortlessly used in AI clusters that rely on high performance, while maintaining the flexibility and robustness of the Linux network stack.

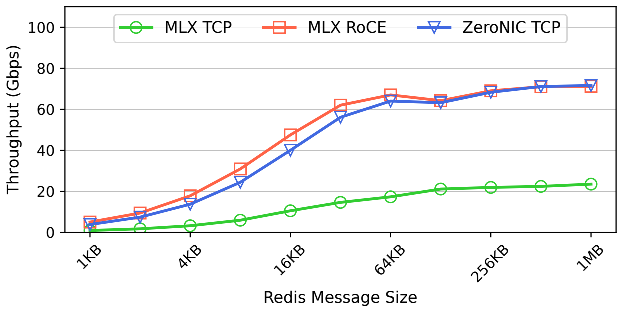


Figure 11: Redis throughput across different payload sizes.

Redis. Redis [83] is an in-memory key-value database widely used for in-memory caching. Redis' performance is crucial for a wide range of data-intensive web-scale applications. We used Redis with the *libibverbs* API, and we evaluated its throughput on *ZeroNIC* against Mellanox RoCE (MLX RoCE) and kernel TCP using the Mellanox NIC (MLX TCP). We ran the *redis-benchmark* [84] using the `SET` operation and varied the payload size from 1*KB* to 1*MB*. We used a total of 4 clients with 10 outstanding requests per client in order to saturate the Redis server thread.

Figure 11 illustrates that *ZeroNIC* achieves end-to-end performance on par with RoCE, averaging 89% of RoCE's throughput across all the payload sizes. Compared to Mellanox TCP, *ZeroNIC* achieved a 3.71× higher throughput on average, benefiting from the lower CPU overheads. For instance, for the 16*KB* message size, 71% of CPU cycles are consumed by networking stack overhead for Mellanox TCP. In contrast, both *ZeroNIC* and Mellanox with RoCE allow for nearly 99% of the cycles to be dedicated to application-level processing.

5.3 *ZeroNIC* Robustness Evaluation

While *ZeroNIC* achieves high throughput, it *also* gains the robustness offered by transport protocols such as TCP. To demonstrate the benefits of a flexible control path, we evaluate *ZeroNIC* under various network perturbations and conditions.

***ZeroNIC* supports interleaved packets across multiple zero-copy and non-zero-copy flows.** To evaluate *ZeroNIC*'s ability to handle and scale to multiple flows, we performed an incast experiment combining 2MB zero-copy flows, and 64KB bidirectional short flows that used the unmodified non-zero-copy socket API. The receiver *ZeroNIC* server used a single core (two hyperthreads) to perform application and network processing for all zero-copy flows.

Figure 12 shows (a) the throughput and CPU utilization for the long flows and (b) the p50 latency for the short flows. *ZeroNIC* is able to steer interleaved incoming packets to their correct NIC queues, avoiding flow collision. It maintains fairness across all flows, evenly distributing bandwidth of up to 8 zero-copy flows, the FPGA's hardware limit. Meanwhile, total CPU utilization (protocol processing, driver, provider, and application) remains approximately constant, and the aggregate throughput across flows saturates the available bandwidth. Overall, *ZeroNIC* achieves high throughput with roughly constant CPU resource demands as the flow count scales. This result, together with our moderate hardware memory requirements to support thousands of high-performance flows (§3.2), validates our design's scalability. Additionally, regardless of the number of high-performance flows, *ZeroNIC* can still concurrently support non-zero-copy flows. Figure 12b shows that their latency is not affected by the number of long flows.

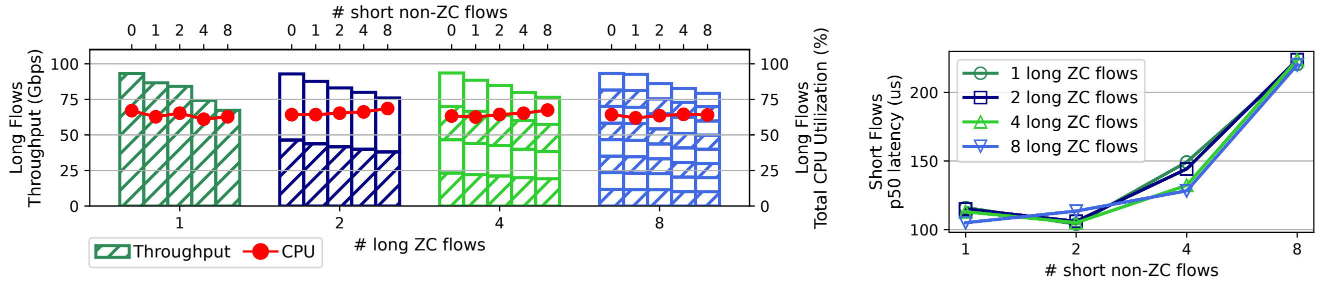
***ZeroNIC* extends the drop-resistance of TCP to GPU-direct data paths.** A primary motivation for physically separating the control and data paths is to combine zero-copy performance with a mature network transport. This experiment examines packet losses, which is a significant problem for RDMA solutions, like RoCE, that were designed assuming a lossless fabric (§2.1). We injected packet drops by adding probabilistic filtering rules at the RX side for both *ZeroNIC* and the Mellanox TCP baseline with TX zero-copy. We could not replicate this experiment for RoCE because none of our available drop rules [35, 66, 92] could intercept RDMA traffic.

Figure 13 shows the throughput of a single GPU-to-GPU flow as we raised the probabilistic drop rate from 0.1% to 10% for the two systems. As expected, both TCP-based systems perform well at low drop rates. *ZeroNIC* maintains near-full throughput even at 1% drops, taking advantage of TCP's mechanisms for drop resistance (retransmitting minimal data). RoCE is known to collapse to near-zero throughput at 1% drops due to the use of go-back-N for retransmissions [69, 103].

Adding drop resistance to GPU-to-GPU traffic is particularly important for AI clusters. It removes the pressure to design a perfect congestion control mechanism and to oversize switch buffers. It also allows switch chips to be configured to use cut-though switching instead of store-and-forward switching. The latter is forced by the need for forward-error-correction (FEC) in order to reduce noise-induced packet losses to zero.

6 Discussion

Zero-copy is critical but is not a panacea. We demonstrated that a data path with both receive and send-side zero-copy improves host networking even with mature network protocols. However, as network links scale to 800*Gbps* and beyond for workloads like artificial intelligence, the control path will become the next bottleneck. Recent proposals to reduce packet processing overheads include hardware offload [40, 60], system call mitigation [18, 31], extending segmentation offload [16, 29, 52], and cache-aligned reorganization [61]. To improve metadata I/O between the NIC and software, systems like Enso [86] and PacketMill [30] introduce optimizations



(a) Long flow (zero-copy) throughput and CPU utilization under collision with short flows.

(b) Short flow (non-zero-copy) p50 latency.

Figure 12: Robustness experiment with long (zero-copy) and short (non-zero-copy) CPU-to-CPU flows colliding in *ZeroNIC*.

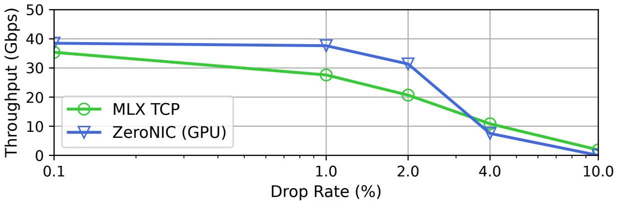


Figure 13: Throughput under instrumented drops in *ZeroNIC* (GPU-to-GPU) and Mellanox TCP (CPU-to-CPU).

for the efficient use of PCIe bandwidth. Our design is well-positioned to help with and benefit from this evolution.

Our design enables an agile evolution of the data and control path. Our design creates a triangle between the NIC, data path devices, and control path devices with well-defined interfaces and responsibilities. This allows the fast and largely independent evolution and optimization of control and data path devices. For example, a host architect can quickly swap GPUs for other AI accelerators without re-implementing or re-optimizing the network data path or the control path.

Control path flexibility is equally important. Achieving the right balance of features and resources integrated in the NIC is difficult. Unlike RNICs that jointly implement the control and data paths in inflexible hardware or opaque firmware, our NIC design implements only a necessary subset of features to support remotely executed control paths. Via our design, a system architect may use a different network protocol in order to improve fabric performance (high utilization, reduced drops, reduced hot spots, etc.) or use a specialized hardware component for faster header processing (CPU with specialized cores, FPGAs, or specialized accelerator). Our design facilitates changes in the control path of the triangle without necessitating changes in the performant data path or the application layer.

The bounds of the maximum bandwidth and minimum latency of communication between the elements of the triangle are set by the link specifications that connect them. Our prototype uses PCIe links and inherits PCIe's bandwidth and latency profiles. As higher throughput and/or lower latency links, such as CXL and NVLink, gain acceptance in industry, our design will benefit from their characteristics.

7 Conclusion

Current end-host network stacks inherently couple the control and data path, resulting in implicit trade-offs between the flexibility and performance of network solutions. In this paper, we showed that a physical separation of the data and control paths allows host network stacks to achieve *both* flexibility and performance. To this end, we presented a co-designed hardware and software stack, which enables a zero-copy receive and send data path between the NIC and any device memory, controlled by any arbitrary transport protocol. We showcased *ZeroNIC*, a prototype that combines an FPGA-based NIC and the TCP stack in the Linux kernel as the transport protocol. Our prototype saturates available network bandwidth on CPU and GPU benchmarks. It improves TCP-based NCCL and Redis throughput by 2.66× and 3.71×, respectively, over Linux TCP on a Mellanox ConnectX-6, all while maintaining the robustness of the TCP transport.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Aurojit Panda, whose comments have greatly helped improve this paper. We would also like to acknowledge the contributions of Antonis Michaloliakos, Grigori Inozemtsev, George Prekas, and many others at Enfabrica who have played a vital role in this endeavor. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research was also partly supported by NSF CNS-2047283. Athinagoras Skiadopoulos was supported by a Stanford Graduate Fellowship. Mark Zhao was supported by a Stanford Graduate Fellowship and a Meta Ph.D. Fellowship. Qizhe Cai was supported by a Meta Ph.D. Fellowship.

References

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204, 2022.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[3] David Ahern and Shrijeet Mukherjee. Merging the networking worlds. In *The Technical Conference on Linux Networking (Netdev 0x16)*, 2023.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[5] AMD. Amd hbm, 2023. https://www.amd.com/en/technologies/hbm.

[6] InfiniBand Trade Association. Infiniband architecture specification, 2023. https://www.infinibandta.org/ibta-specification/.

[7] InfiniBand Trade Association. Infiniband architecture specification: Rocev2, 2023. https://www.infinibandta.org/ibta-specification/.

[8] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[10] Lawrence Brakmo. Tcp-bpf: Programmatically tuning tcp behavior through bpf. In *The Technical Conference on Linux Networking (Netdev 2.2)*, 2017.

[11] L.S. Brakmo and L.L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[15] Solarflare Communications. Onload user guide, issue 20, 2015. Section D.6 at https://www.smallake.kr/wp-content/uploads/2015/12/SF-104474-CD-20_Onload_User_Guide.pdf.

[16] Jonathan Corbet. Jls2009: Generic receive offload, 2009. https://lwn.net/Articles/358910/.

[17] Jonathan Corbet. A reworked tcp zero-copy receive api, 2018. https://lwn.net/Articles/754681/.

[18] Jonathan Corbet. The rapid growth of io_uring, 2020. https://lwn.net/Articles/810414/.

[19] NVIDIA Corporation. Massively scale your deep learning training with nccl 2.4, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[20] NVIDIA Corporation. Massively scale your deep learning training with nccl 2.4, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[21] NVIDIA Corporation. Nvidia dgx superpod: Scalable infrastructure for ai leadership, 2021. https://images.nvidia.com/aem-dam/Solutions/Data-Center/gated-resources/nvidia-dgx-superpod-a100.pdf.

[22] NVIDIA Corporation. Gpudirect rdma, 2023. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[23] NVIDIA Corporation. NCCL Tests, 2023. https://github.com/NVIDIA/nccl-tests.

[24] NVIDIA Corporation. Nvidia nccl, 2023. https://developer.nvidia.com/nccl.

[25] NVIDIA Corporation. Nvidia blackwell architecture, 2024. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

[26] NVIDIA Corporation. Nvidia dgx b200, 2024. https://www.nvidia.com/en-us/data-center/dgx-b200/.

[27] Linux Networking Documentation. Msg_zerocopy, 2023. https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html.

[28] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf – the ultimate speed test tool for tcp, udp and sctp. 2021.

[29] Eric Dumazet and Coco Li. Big tcp. In *The Technical Conference on Linux Networking (Netdev 0x15)*, 2021.

[30] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Packetmill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–17, 2021.

[31] The Linux Foundation. Linux foundation docuwiki: napi, 2016. https://wiki.linuxfoundation.org/networking/napi.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[33] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, pages 97–104. Springer, 2004.

[34] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {RDMA}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[35] Steven Gordon. baidu-research/tensorflow-allreduce, 2013. https://github.com/baidu-research/tensorflow-allreduce.

[36] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, pages 274–281, 2020.

[37] OpenFabrics Interfaces Working Group. Libfabric openfabrics, 2023. https://ofiwg.github.io/libfabric/.

[38] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, jul 2008.

[40] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[41] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[43] Juha Heinanen and Roch Guerin. Rfc2698: A two rate three color marker, 1999.

[44] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, et al. Datacenter ethernet and rdma: Issues at hyperscale. *arXiv preprint arXiv:2302.03337*, 2023.

[45] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 92–98, 2016.

[46] IEEE802. 802.1qbb - priority-based flow control, 2011. https://1.ieee802.org/dcb/802-1qbb/.

[47] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, aug 1988.

[48] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[49] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433*, 2023.

[50] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, 2018.

[51] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *EuroSys*, 2019.

[52] The kernel development community. Segmentation offloads, 2016. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt.

[53] The kernel development community. Msg_zerocopy, 2017. https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html.

[54] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. {FreeFlow}: Software-based virtual {RDMA} networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, 2019.

[55] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[56] Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. Splitrpc: A {Control+ Data} path splitting rpc stack for ml inference serving. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–26, 2023.

[57] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 225–236, 2018.

[58] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[59] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.

[60] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[61] Coco Li. Analyze and reorganize core networking structs to optimize cacheline consumption, 2023. https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/.

[62] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. Flor: An open high performance {RDMA} framework over heterogeneous {RNICs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 931–948, 2023.

[63] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, 2023.

[64] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[65] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[66] NVIDIA Cumulus Linux. Buffer and queue management, 2023. https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-43/Layer-1-and-Switch-Ports/Buffer-and-Queue-Management/.

[67] linux rdma. Rdma-core libibverbs, 2021. https://github.com/linux-rdma/rdma-core.

[68] Shao Liu, Tamer Başar, and Ravi Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pages 55–es, 2006.

[69] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, page 22–28, New York, NY, USA, 2017. Association for Computing Machinery.

[70] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[71] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2005.

[72] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Dave Minturn. Nvm express over fabrics. In *11th Annual OpenFabrics International OFS Developers' Workshop*, 2015.

[74] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326, 2018.

[75] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware codesign for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[76] Zhixiong Niu, Qiang Su, Peng Cheng, Yongqiang Xiong, Dongsu Han, Keith Winstein, Chun Jason Xue, and Hong Xu. Netkernel: Making network stack part of the virtualized infrastructure. *IEEE/ACM Transactions on Networking*, 30(3):999–1013, 2021.

[77] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[79] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[80] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general-purpose on-nic memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1130–1147, 2022.

[81] PyTorch. Pytorch distributed data parallel, 2023. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html.

[82] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 540–553. IEEE, 2021.

[83] Redis. Redis, 2023. https://redis.io.

[84] Redis. Redis benchmark, 2023. https://redis.io/docs/management/optimization/benchmarks/.

[85] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Accelerating collective communication in data parallel training across deep learning frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1027–1040, 2022.

[86] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S Berger, James C Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. {Ensō}: A streaming interface for {NIC-Application} communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1005–1025, 2023.

[87] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A {High-Speed} stateful packet processing approach for tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, 2023.

[88] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[89] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[90] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[91] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *ACM SIGCOMM*, 2020.

[92] Enterprise SONiC. Acl (access control list), 2023. https://support.edge-core.com/hc/en-us/articles/900000214926--Enterprise-SONiC-ACL-Access-Control-List-#h_01FWN7B0ED07H7A9CWD8243KZW.

[93] Mellanox Technologies. Introduction to infiniband. Technical report, 2003. https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[94] PATH to TCP 4K MTU and RX zerocopy. Eric dumazet, 2020. https://www.infinibandta.org/ibta-specification/.

[95] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[96] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[97] Inc. Xilinx. Virtex ultrascale+ fpga product brief, 2021. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/virtex-ultrascale-product-brief.pdf.

[98] Inc. Xilinx. Dma/bridge subsystem for pci express v4.1, 2022. https://www.xilinx.com/support/documents/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[99] Inc. Xilinx. Qdma subsystem for pci express v4.0, 2022. https://www.xilinx.com/support/documents/ip_documentation/qdma/v4_0/pg302-qdma.pdf.

[100] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 22), pages 521–538, 2022.

[101] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[102] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. Deploying user-space {TCP} at cloud scale with {LUNA}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, 2023.

[103] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

INTOS: Persistent Embedded Operating System and Language Support for Multi-threaded Intermittent Computing

Yilun Wu*, Byounguk Min†, Mohannad Ismail‡, Wenjie Xiong‡, Changhee Jung†, Dongyoon Lee*

*Stony Brook University, †Purdue University, ‡Virginia Tech

Abstract

This paper introduces INTOS, an embedded operating system and language support for multi-threaded intermittent computing on a battery-less energy-harvesting platform. INTOS simplifies programming with a traditional “thread” and a “transaction” with automatic undo-logging of persistent objects in non-volatile memory. While INTOS allows the use of volatile memory for performance and energy efficiency, conventional transactions do not ensure crash consistency of volatile register and memory states. To address this challenge, INTOS proposes a novel *replay-and-bypass* approach, eliminating the need for users to checkpoint volatile states. Upon power restoration, INTOS recovers non-volatile states by undoing the updates of power-interrupted transactions. To reconstruct volatile states, INTOS restarts each thread bypassing committed transactions and system calls by returning recorded results without re-execution. INTOS seeks to build a persistent, full-fledged embedded OS, supporting priority-based preemptive multithreading while ensuring crash consistency even if power failure occurs during a system call or while some threads are blocked. Experiments on a commodity platform MSP430FR5994 show that when subjected to an extreme power failure frequency of 1 ms, INTOS demonstrated 1.24x lower latency and 1.29x less energy consumption than prior work leveraging idempotent processing. This trend turns out to be more pronounced on Apollo 4 Blue Plus.

1 Introduction

Instead of using a battery, energy-harvesting systems [24, 30, 40, 54, 57] capture necessary energy from ambient sources (*e.g.,* solar [27], radio frequency [35]) and leverage a small capacitor as energy storage. The ability to offer sustainable and long-term deployment without the need for battery replacements has unlocked a diverse range of emerging applications such as body implants [29], wearables [61], wildlife tracking [67], road monitoring [26], and satellites [5].

Since the capacitor undergoes cycles of depletion and recharge, program execution on an energy-harvesting system is inherently *intermittent*, involving repetitive power interruptions and resumptions. The nature of intermittent computing necessitates crash consistency to guarantee correct resumption throughout frequent power cycles.

An operating system (OS) offers essential services to application developers (users), including multi-threading, queues, semaphores, events, and timers, to assist in creating feature-rich applications. To illustrate, widely-used embedded OSes like FreeRTOS [3] have streamlined the development of diverse embedded applications. Unfortunately, this level of OS-/runtime support is absent in intermittent computing environments. For instance, ImmortalThreads [65] offers a tiny runtime supporting (pseudo) threads with cooperative scheduling; however, its capabilities are limited. It lacks a wait-list for blocking threads. Its event loop is based on polling, wasting microcontroller (MCU) cycles. Many task-based solutions such as Ink [64] and CatNap [52] do not support threads.

There arises a growing need for a more robust OS tailored specifically for intermittent computing. Advancements in hardware technologies, such as ultra-low power microcontrollers like TI’s MSP430FR [6] and ARM’s Cortex-M4 [2], as well as non-volatile memory (NVM) like FRAM [12] and MRAM [10], have empowered intermittent applications to perform more computations. Emerging intermittent applications are becoming increasingly complex, incorporating features like multi-threading, communication, synchronization, and responsiveness to events. We started witnessing machine and deep learning tasks [16, 28, 39, 48] on energy-harvesting platforms. Despite these advancements, users are compelled to manage this complexity without adequate OS support.

Unfortunately, the current crash consistency solutions are hard to adopt or result in inefficient designs when applied to the development of persistent embedded OS kernels. Some approaches [22, 31, 49, 52, 53, 56, 64] require users to decompose applications into a task graph, demanding each task to inherently possess failure-atomicity and idempotence. This poses considerable challenges for programmers [38, 65]. Breaking down a kernel system call, such as creating a thread or blocking on a full queue, into tasks is not trivial. Other compiler-based solutions [15, 18, 19, 37, 47, 50, 55, 62] automatically divide programs (*e.g.,* into idempotent regions) and incorporate checkpoints, requiring little to no user annotations. Thus, they may be used to build a persistent OS. Yet, many (except Chinchilla [50]) assume execution solely on NVM, overlooking potential advantages offered by volatile memory.

This paper introduces INTOS, a new persistent full-fledged embedded OS accompanied by language support for multithreaded intermittent computing (§4). To ease intermittent application programming, INTOS offers a traditional "thread" along with a priority-based preemptive scheduler. INTOS also allows users to define a standard "transaction" with automatic undo-logging to ensure the crash consistency of persistent objects residing in NVM, akin to a widely adopted Intel's Persistent Memory Programming Kit (PMDK) [7]. INTOS places program stacks, encompassing local variables and function frames, in volatile memory to improve performance and energy efficiency. However, their crash consistency in the event of a power failure is not safeguarded by transactions. The absence of volatile states (*e.g.,* stacks) makes it impossible to simply resume from the beginning of a transaction.

To address this challenge, INTOS proposes a new *replay-and-bypass* approach (§5). Upon power restoration, INTOS recovers non-volatile states by undoing the updates of power-interrupted transactions. To reconstruct volatile states, it then restarts each thread from the beginning while bypassing committed transactions and system calls by returning recorded results without re-execution. This approach is grounded in the insight that reconstructing the volatile states with replay-and-bypass is more energy-efficient, compared to alternatives checkpointing volatile states to NVM—since NVM writes are the most energy-consuming in the instruction set architecture.

In particular, INTOS provides a programming model based on Rust, leveraging Rust's type system to enforce various programming rules (§6). These rules are designed to ensure crash consistency: *e.g.,* the prohibition of modifications to persistent objects outside of transactions. The adaptability of this programming model has been showcased through the successful implementation of the INTOS kernel, featuring multithreading, queues, semaphores, events, and timers.

We evaluate INTOS with three single-threaded and eight multi-threaded applications, including those ported from RIoTBench [58], an IoT/Edge stream processing benchmark for real city sensing and fitness sensing data, on MSP430FR5994 [6] and Apollo 4 Blue Plus [1]. We compare INTOS with Ratchet [62] where compiler-based idempotent processing is applied in both the INTOS kernel and application. On the MSP430FR platform without power failures, INTOS exhibited 1.65x lower latency and 1.85x less energy, compared to Ratchet. Even when subjected to an extreme power failure frequency of 1 ms, INTOS demonstrated 1.24x lower latency and 1.29x less energy overhead. This trend became more pronounced on the Apollo 4 platform.

This paper makes the following contributions:

- To the best of our knowledge, INTOS is the first persistent embedded OS that supports priority-based preemptive multithreading and other core features, tailored for intermittent computing with frequent power failures.
- INTOS combines transactional programming with a new replay-and-bypass recovery mechanism to ensure whole-system crash consistency, encompassing both volatile and non-volatile memory states at both user and kernel levels.
- INTOS introduces a Rust-based programming model ensuring crash consistency through the proposed transactions and replay-and-bypass recovery mechanisms.
- INTOS is to our best knowledge the first intermittent system that is evaluated with multithreaded applications.

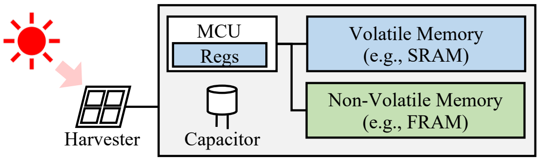


Figure 1: An architecture of energy harvesting platforms (*e.g.,* MSP430FR). Registers and SRAM (blue boxes) are volatile.

2 Background

This section briefly discusses intermittent computing, embedded OS, and transactions.

2.1 Intermittent Computing

Execution on an energy-harvesting platform is *intermittent*, *i.e.,* it abruptly halts upon the depletion of the capacitor and resumes after recharging, typically to the full capacitance. This implies that the program is often power-interrupted, and therefore intermittent computing requires to ensure crash consistency for correct recovery across frequent power cycles.

Figure 1 depicts an architecture of energy harvesting platforms available in TI MSP430 [6] or Ambiq Apollo 4 [1]. The energy harvester gathers ambient energy (*e.g.,* solar, RF) and stores it in a capacitor. Capacitor sizes typically vary from a few to several hundred microfarads (µF). For reference, WISP [57] uses 47 µF. The computing components include an ultra-low power microcontroller (MCU) along with both volatile memory (*e.g.,* SRAM) and non-volatile memory (*e.g.,* FRAM [12] or MRAM [10]). For instance, the MSP430FR5994 features a 16 MHz MCU with 8KB SRAM and 256KB FRAM. Registers and SRAM states (blue boxes) are lost upon a power outage. Previous solutions (§3) have suggested diverse approaches to maintaining the crash consistency of data stored in registers, volatile memory, and non-volatile memory across a power cycle.

Prior Works	Crash Consistency	Multithreads?	Queues?	Semaphores?	Events?	Timers?	Prog. Burden?	Volatile Mem?
Alpaca [49], Coala [53], MayFly [31]	manual task decomposition	no (tasks)	no	no	no	no	high	yes
Chain [22]	manual task decomposition	no (tasks)	limitedly	no	no	no	high	yes
Coati [56], Ink [64]	manual task decomposition	no (tasks)	no	no	limitedly	no	high	yes
CatNap [52]	manual task decomposition	no (tasks)	limitedly	no	limitedly	no	high	yes
Ratchet [62], WARio [37]	idempotent processing	no	no	no	no	no	none, very low	no
Chinchilla [50]	ckpt & undo-logging	no	no	no	no	no	none, very low	yes
HarvOS [15], RockClimb [19]	static energy analysis	no	no	no	no	no	none, very low	no
TICS [38]	ckpt & undo-logging	no	no	no	no	no	low	no
ImmortalThread [65]	ckpt & micro-continuation	yes (pseudo-stackful)	no	limitedly	limitedly	no	low	no
IntOS (ours)	replay & undo-logging	yes (stackful)	yes	yes	yes	yes	medium (transactions)	yes (replay)

Table 1: A comparison of the main features of IntOS with prior intermittent computing solutions.

2.2 Embedded Operating Systems

Embedded OSs [3,13,14,25,41,42] are a specialized software layer that provides essential services for the target embedded system. They empower users (application developers) to create applications with rich features using a conventional thread-based programming model, even within resource-constrained environments. For instance, FreeRTOS [3], widely recognized as one of the most adopted, supports (1) multi-threading with a priority-based preemptive scheduler; (2) synchronization (*e.g.,* semaphores) and communication (*e.g.,* queues) among threads; (3) dynamic memory allocation; and (4) software timers. An embedded OS is intimately linked with the application code and is typically included as part of the firmware image. Existing embedded OS kernels are not designed to be crash-consistent and do not support intermittent computing.

2.3 Transactions for Non-volatile Memory

Transactions stand out as a widely adopted programming model for NVM, as demonstrated by Intel's PMDK [7] for Optane memory [4]. Users can allocate a persistent object using a non-volatile memory allocator. A transaction employs undo logging (or redo logging) to ensure failure-atomicity (the "all-or-nothing" semantic) for operations executed during the transaction. Transactional programming has demonstrated success in the development of complex software such as persistent memchached [9] and redis [8].

3 Related Work

This section initially emphasizes the absence of essential OS features in prior solutions (Table 1) and then delves into the challenges associated with applying existing crash consistency solutions to design persistent OS services.

No OS exists for intermittent computing. As highlighted in the middle five columns of Table 1, current intermittent processing runtimes lack essential features present in modern embedded OSes. ImmortalThreads [65], for example, introduces (pseudo) multithreading with "non-blocking" spin-locks and event buffers. Spinning results in inefficient utilization of MCU cycles. To support "blocking" semaphores, queues, event groups, and software timers in intermittent computing, an OS/runtime should maintain a run-queue, wait-queues, and other relevant kernel metadata in a crash-consistent manner. ImmortalThreads (its runtime) does not offer them.

We believe ImmortalThreads can be extended to implement such missing kernel features using its micro-continuation approach. However, we expect ImmortalThreads would suffer from two fundamental problems. First, ImmortalThreads would incur high performance overhead. Unlike those hardware-based roll-forward solutions [17,21,36,51,66] that detect impending power failure and save registers to resume from the failure point, ImmortalThreads does not (cannot) sense the dying voltage. Thus, it ends up persisting a program counter in every store instruction to enable roll-forward recovery (micro-continuation). Second, micro-continuation only works for non-volatile memory and excludes volatile SRAM available in commodity energy harvesting systems, thereby losing a great opportunity to enable more energy-efficient intermittent computing. JustDo logging [34], from which the micro-continuation idea is inspired, also requires the entire memory hierarchy to be fully persistent. We discuss ImmortalThreads' potential high overhead later in §10.

On the other hand, Ink [64], Coati [56], and Catnap [52] offers partial support for task-based event-driven runtimes, yet they do not accommodate threads and demand a task-based programming model, which we explain next.

Manual task decomposition adds programming burden. For crash consistency, several prior solutions [22,31,49,52,53,56,64] require users to decompose an application into a graph of "tasks". Each task is compelled to inherently guarantee failure atomicity and idempotence in the face of a power failure, leading to considerable programming challenges and design complexities. Some runtime systems employ a cooperative scheduler to execute multiple tasks. However, the manual task decomposition shifts the responsibility of ensuring crash consistency onto users. This has been demonstrated to be a significant burden for programmers [38, 65]. For example, breaking down a kernel system call, such as creating a thread or blocking on a full queue, into tasks is far from trivial.

Within the task-based model, several new features have been introduced. For example, Alpaca [49] suggests task privatization, creating a volatile copy of shared non-volatile variables before entering each task. A task's local computation can run on volatile memory. Upon task completion, updates to shared variables are committed to NVM in a double-buffered manner. Chain [22] abstracts inter-task variable passing with the use of a persistent queue; Ink [64] and Coati [56] support event-driven programming; and CatNap [52] adaptively schedules tasks based on task priority, energy consumption, current energy level, and charging rate.

Automatic checkpointing often does not consider volatile memory. Several works [15, 19, 37, 47, 50, 55, 62] have introduced compiler support to automatically partition a program into multiple regions and insert checkpointing at the boundaries of these regions. Users require little to no annotations, so they can be used to build persistent OS services. Our evaluation (§10) includes a comparison against Ratchet [62] idempotent processing. However, many compiler-based solutions assume a program execution solely on non-volatile memory, foregoing the potential performance and energy efficiency benefits that volatile memory could provide. Experiments with MSP430FR5994 (§10.1) show that executing our 11 benchmarks entirely in non-volatile memory (FRAM) results in 1.11x latency and 1.16x energy overheads compared to running them entirely in volatile memory (SRAM). In this context, only live-in (volatile) registers necessitate checkpointing at a region boundary. A notable exception is Chinchilla [50] which maintains volatile and non-volatile stacks, yet it still involves frequent checkpoints of stacks to NVM.

Ratchet [62] divides a program into idempotent regions [23, 43, 45, 46] with no write-after-read dependencies within a region. Chinchilla [50] selectively skips certain checkpoints based on energy conditions. WARio [37] reduces the number of idempotent regions by reordering instructions and incorporating a loop optimization. Differently, HarvOS [15] partitions a program into regions where the energy required to complete that region is less than the energy buffer size. RockClimb [19] checks the energy level at the region boundary and only proceeds if there is sufficient energy. On the other hand, TICS [38] and ImmortalThreads [65] leverage a compiler for checkpoint instrumentation without region partitioning. TICS employs stack segmentation, where only a working stack (and registers) is checkpointed in NVM via a two-phase commit. ImmortalThreads introduces micro continuation, which checkpoints every memory update, ensuring the idempotence of the execution until the next checkpoint (store).

Other issues Prior works also address data timeliness [31, 38, 52, 64], event-driven programming model [38, 52, 56, 64], and others. Surbatovich et al. [59, 60] use Rust's type systems for data freshness checking and crash consistency. Hardwawre support [20, 32, 36, 44, 63, 68] also exists.

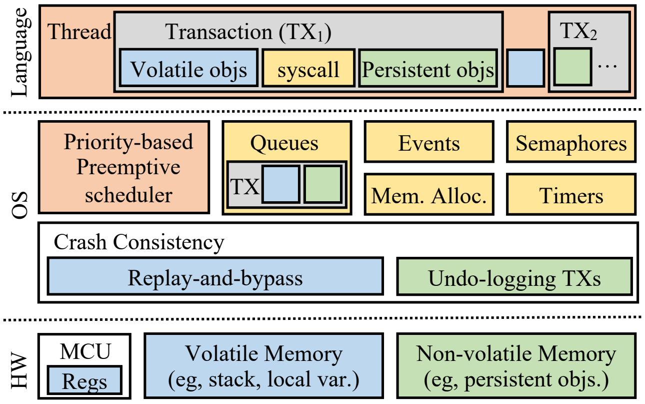


Figure 2: Overview of INTOS

4 Overview of INTOS

Figure 2 shows an overview of INTOS, embedded OS and language support for multi-threaded intermittent computing.

4.1 Multithreading and Transactions

Threads To ease the aforementioned programming burden and keep the same embedded application programming model, INTOS supports a traditional stackful "thread" as a programming unit and a schedulable entity, similar to commodity embedded OSes (*e.g.,* FreeRTOS[1]). Users can generate multiple threads through the system call `sys_create_thread` (Table 2). These threads run concurrently. Users can assign different priorities for threads. The INTOS scheduler employs a priority-based preemptive scheduling policy, a widely adopted approach for real-time capabilities. More discussion on INTOS's real-time capability will follow in §8.

Transactions To facilitate the utilization of both volatile and non-volatile memories while simplifying crash-consistent programming, INTOS offers a conventional "transaction" with automatic undo-logging to ensure crash-atomicity of persistent objects. Program stacks, encompassing local variables and function frames, reside in volatile memory. Users can either annotate a persistent variable (*e.g.,* globals) or employ the `sys_palloc` system call to create a persistent object in non-volatile memory. Both volatile and persistent objects can be used inside a transaction. Users do not need manual undo-logging. INTOS's transactions ensure that updates on persistent objects (not volatile ones) within a transaction are crash-atomic via automatic undo-logging. INTOS leverages Rust to identify the first writable dereference. As persistent objects share a base class or trait in Rust, the logging logic is integrated into the dereference operation within the class/trait. This approach mirrors PMDK's undo-logging support for its

[1] FreeRTOS employs the term "task", but it is technically a preemptive thread. For clarity and to distinguish it from the (cooperative) task in manual task decomposition works (§3), we will refer to it as a (preemptive) thread.

C++ programs. In §7.3, we discuss an undo logging optimization that logs old values only if there is a write-after-read dependency in a transaction.

Language The Rust type system in IntOS guarantees that persistent objects are not modified outside transactions (§6). Conversely, volatile states, such as local variables in program stacks, can be utilized both outside and inside transactions.

Challenges IntOS allows users to employ volatile variables for computing, yet IntOS transactions do not protect them. Consequently, the states of program stacks are susceptible to loss upon a power failure. A program cannot resume from the beginning of the failed transaction due to the absence of stack (and register) states. One solution could involve abstaining from the use of volatile memory, a proposition we oppose for energy efficiency reasons. Another approach might be to checkpoint volatile states to NVM at the onset of each transaction, but this would be expensive.

4.2 Replay-and-Bypass

To address the above challenge, IntOS proposes a novel *replay-and-bypass* approach (§5) to guarantee whole-system crash consistency across a power cycle. IntOS eliminates the need for users to checkpoint or create customized crash consistency solutions for volatile register and memory states. Upon power restoration, IntOS first recovers non-volatile states by undoing uncommitted transactions. Then, the thread is restarted from the beginning, safely resuming with empty registers and stack states. Throughout the execution, committed transactions and system calls are replayed and bypassed by returning the recorded results without re-execution – resulting in a more energy-efficient recovery process. Volatile states are reconstructed, enabling the program to resume beyond the point of power failure.

4.3 Persistent Embedded OS

System Calls IntOS provides comprehensive multithreading features (Table 2), comparable to those found in FreeRTOS. For instance, threads can communicate and/or synchronize with each other using the `sys_queue_*` and `sys_semaphore_*` system calls. A thread might block, for instance, if a queue is either empty or full. Multiple threads may access a shared persistent object by obtaining its reference (inside a transaction). Later in §6, we delve into how IntOS's programming model ensures the obligatory use of a mutex for synchronization via Rust's strong type system.

Kernel Crash Consistency Similar to user threads, IntOS kernel codes, including system calls, utilize volatile and non-volatile memories. The IntOS kernel employs the same undo-logging transactions to ensure crash consistency of persistent kernel objects that undergo updates during system calls. Table 2 lists the number of transactions and examples of persistent kernel data safeguarded by kernel-level transactions. Later in §7.2, we also discuss that the kernel uses optimized transactions (without undo-logging) for frequently used linked lists operations (*e.g.,* `ready-list`, `wait-list`). Using the same replay-and-bypass, IntOS provides a crash consistency guarantee even if a power failure occurs in the midst of a system call and some threads are blocked.

Features	System calls	TXs	Persistent kernel objects
Threads	sys_create_thread	2	ready_list, thread_cnt, heap
	sys_thread_delay	1	delay_list
Queues	sys_queue_create	1	heap
	sys_queue_send_back	2	queue and its waitlist
	sys_queue_receive	2	queue and its waitlist
Events	sys_event_group_create	1	heap
	sys_event_group_wait	3	event_grp and its waitlist
	sys_event_group_set	3	event_grp and its waitlist
Semaphores	sys_create_semaphore	1	heap
	sys_semaphore_take	2	semaphore and its waitlist
	sys_semaphore_give	2	semaphore and its waitlist
Dyn. memory	sys_palloc	1	heap
	sys_pfree	1	heap
Timers	sys_timer_create	1	heap
	sys_start_timer	2	timer_cmd_q and its waitlit
	sys_reset_timer	2	timer_cmd_q and its waitlist

Table 2: IntOS supports full-fledged embedded OS features, akin to FreeRTOS [3]. Some system calls are not listed.

Energy efficient execution Designing an OS and language support for intermittent computing requires more than merely ensuring crash consistency. Both (fail-free) execution and recovery should be energy-efficient. IntOS provides energy-efficient execution by: (1) Utilizing both volatile and non-volatile memories; (2) Avoiding the checkpointing of volatile states; (3) Optimizing undo-logging for non-volatile states (§7.3); (4) Offering blocking/waiting system calls, such as semaphores and events, in contrast to existing approaches like ImmortalThreads [65], which requires spinning and wastes MCU cycles; (5) In the absence of events, with a blocking mechanism, allowing the MCU to enter a deep sleep mode where only a subset of interrupts are monitored.

Energy efficient recovery IntOS offers energy-efficient recovery by: (1) Utilizing replay-and-bypass recovery to avoid redundant execution (§5); (2) Undoing only the non-volatile state relevant to the high-priority thread that will resume during recovery (§5.2); (3) Introducing loop optimization (§7.1).

4.4 IntOS Program Example

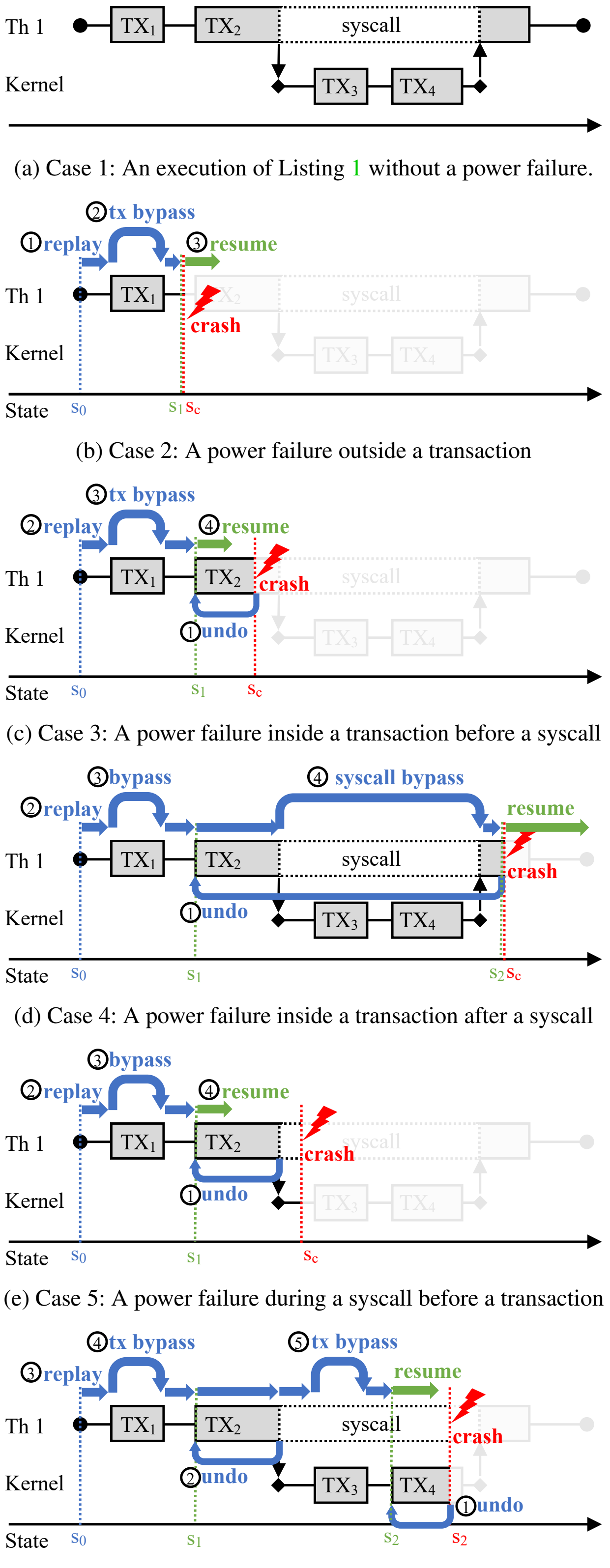
The example presented in Listing 1 illustrates the `recognize` program with two transactions. In this example, a queue is created to enable message passing between two threads, like a Linux pipe. A thread (`recognize`) is reading data from the sensor and sending the data to another thread (not shown) for processing using the queue. `PBox` is a smart pointer for a

```
1  fn recognize(model: PBox<Model>) {
2    let (q,stats) = transaction::run(|j, t| {
3      // syscall to create a queue
4      let q=sys_queue_create::<Result>(Q_SZ, t).unwrap();
5      // syscall to create a persistent object
6      let stats = PBox::new(Stats::new(), t);
7      ...
8      return (q,stats)
9    });
10   transaction::run(|j, t| {
11     // obtain read only ref, no logging
12     let mdl_ref = model.as_ref(j);
13     // syscall to perform I/O
14     let reading = sys_read(SENSOR_0);
15     // data processing in volatile buffer
16     let mut window = [AccelReading::new(); 3];
17     init_window(&readings);
18     transform(&mut window, j);
19     let feature = featurize(&window);
20     let class = classify(&feature, mdl_ref);
21     // obtain mutable ref, auto. undo logging
22     let mut stats_ref = stats.as_mut(j);
23     stats_ref.cnt[class] += 1;
24     // syscall to send result
25     sys_queue_send_back(q, class, WAIT_TIME, t);
26   });
27   ...
28 }
```

Listing 1: An example INTOS program with transactions.

persistent object. Users can enclose a program region with the transaction construct, `transaction::run(|j,t|{ ... })`, where `j` represents the journal object and `t` is the system call token. The journal object enforces restrictions, ensuring that persistent smart pointers like `PBox` cannot be dereferenced outside a transaction, while the system call token restricts system calls to occur exclusively within a transaction. Further details on this will be provided in §6.

The first transaction (Lines 2-9) involves creating a queue with a size of `Q_SZ`. This queue contains objects of type `Result`, and a persistent object (`stats`) that holds counts (`cnt`) for each class/result. The transaction returns them after some processing. The second transaction (Lines 10-26) reads an ML model using a read-only reference. This eliminates the need for undo-logging. Following I/O, it conducts data processing (Lines 16-20) such as filtering, normalization, and classification, notably on a volatile buffer. This strategy enhances performance and energy efficiency compared to conducting all intermediate computations on a non-volatile buffer. Subsequently, the transaction obtains a mutable reference to a persistent object (`stat`), created, and passed from the first transaction, and updates it. As this is the first write after getting a mutable reference, INTOS automatically applies undo logging. Finally, the transaction makes a system call `sys_queue_send_back` to place the result into the queue, maintained by INTOS. Another thread (not shown) can then receive the result from the queue for subsequent processing.



(a) Case 1: An execution of Listing 1 without a power failure.

(b) Case 2: A power failure outside a transaction

(c) Case 3: A power failure inside a transaction before a syscall

(d) Case 4: A power failure inside a transaction after a syscall

(e) Case 5: A power failure during a syscall before a transaction

(f) Case 6: A power failure during a syscall inside a transaction

Figure 3: Replay-with-bypass recovery examples

5 Replay-and-Bypass Recovery

The following two sections demonstrate INTOS's replay-and-bypass approach. along with examples.

5.1 Single Thread Crash Consistency

Let's illustrate INTOS's *replay-and-bypass* recovery mechanism using the `recognize` example in Listing 1, which involves two user-level transactions, $\texttt{TX}_1$ and $\texttt{TX}_2$. Figure 3a depicts an execution of `recognize` without a power failure. For simplicity, the system calls in the first transaction $\texttt{TX}_1$ are omitted, and only the system call `sys_queue_send_back` (Line 23) made by $\texttt{TX}_2$ is highlighted. Assume that the system call includes two kernel-level transactions, $\texttt{TX}_3$ and $\texttt{TX}_4$.

In Figure 3b, we consider a scenario where $\texttt{TX}_1$ has been committed, and then a power failure occurs before $\texttt{TX}_2$ starts (outside transactions). Upon power recovery, INTOS initiates a *replay* of the thread from the beginning (step ①), restarting with empty registers and stack state $\texttt{s}_0$. INTOS's type system (§6) ensures that no non-volatile states are updated outside the transaction. Volatile states are *reconstructed* during replay. Since the non-volatile states at $\texttt{s}_\texttt{c}$ (before the power failure) already incorporate the effects of the committed transaction $\texttt{TX}_1$, re-executing $\texttt{TX}_1$ would be incorrect and non-idempotent. Therefore, INTOS *bypasses* the transaction $\texttt{TX}_1$ (step ②), simply returning the logged return value without re-execution. No system calls are made during bypass, and no kernel-level recovery is required. INTOS ensures that the program reaches the same state $\texttt{s}_1$ as $\texttt{s}_\texttt{c}$, from which it can safely resume.

Now, let's consider a power failure inside a transaction. In Figure 3c, a power failure occurs inside a user-level transaction before a syscall. INTOS's undo-logging transaction ensures the failure-atomicity of non-volatile states changed within the transaction. Upon power recovery, INTOS applies undo-logging (step ①) to roll back the (user-level) non-volatile states from $\texttt{s}_\texttt{c}$ to $\texttt{s}_1$, the state before the transaction begins. Next, INTOS starts a replay from the beginning state $\texttt{s}_0$ (step ②). The committed transaction $\texttt{TX}_1$ is bypassed (step ③), and INTOS reconstructs all volatile states along the way, making the state $\texttt{s}_1$ (after replay) equivalent to $\texttt{s}_\texttt{c}$ (before the failure).

Figure 3d illustrates the actions to be taken if a power failure occurs after a syscall completes (inside a user-level transaction). As usual, INTOS applies undo-logging (step ①) and initiates a replay (step ②). The committed transaction $\texttt{TX}_1$ is bypassed (step ③). Notably, in this scenario, while replaying transaction $\texttt{TX}_2$, INTOS also bypasses the completed system call (step ④). Consequently, INTOS avoids the need to alter kernel states — any changes to kernel-side non-volatile states made during the original system call (before a power outage) can remain unchanged. The INTOS kernel caches the return value of a system call upon its completion (before a power failure). Then it simply returns the cached value during replay. From the user thread's perspective, a system call can be considered as a nested black-box transaction.

Now, let's delve into scenarios where a power failure occurs during a system call. As mentioned earlier, INTOS utilizes transactions ($\texttt{TX}_3$ and $\texttt{TX}_4$) to safeguard kernel-side nonvolatile data. If a crash occurs before (or outside) a kernel-side transaction, as depicted in Figure 3e, the situation is straightforward and aligns with the case presented in Figure 3c. There is no need to undo anything in the kernel. INTOS simply undoes the user-level transaction that invoked the system call (①) and initiates the replay-and-bypass recovery mechanism.

On the other hand, if a crash occurs inside a kernel-side transaction, as illustrated in Figure 3f, INTOS must first undo transaction $\texttt{TX}_4$ (step ①) to roll back the kernel-side state to $\texttt{s}_2$, followed by undoing transaction $\texttt{TX}_2$ (step ②) to roll back the user-side state to $\texttt{s}_1$. INTOS then employs a replay from initial $\texttt{s}_0$ (step ③), bypassing the committed transactions on the user side, $\texttt{TX}_1$ (step ④), and on the kernel side, $\texttt{TX}_3$ (step ⑤). Note that INTOS rolls back the kernel-side transaction first (before any aborted user-level transaction). This has correctness implications in multi-thread scenarios, which will be discussed in the subsequent section.

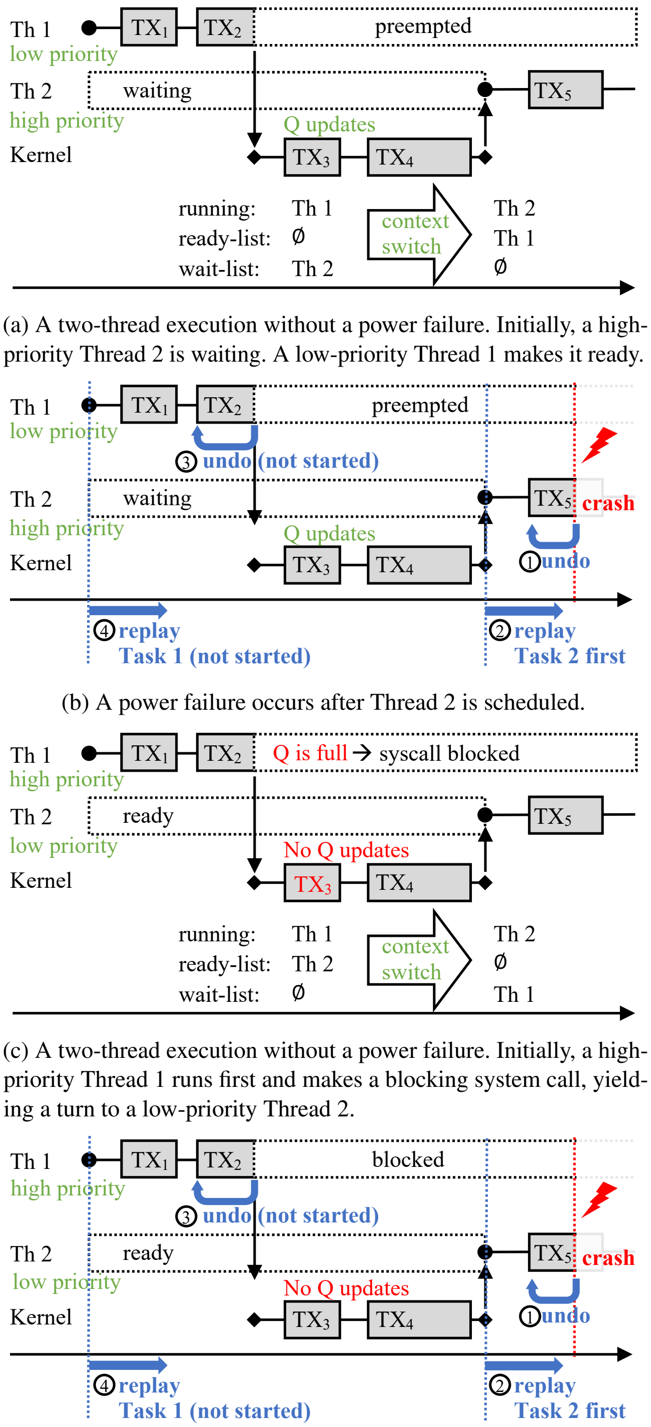
5.2 Multi-Threads Crash Consistency

Next, we discuss INTOS's approach to ensuring crash consistency for multiple threads. Specifically, INTOS employs priority-based recovery and resumption. Upon power restoration, INTOS always recovers and replays the thread with the highest priority among those ready.

Figure 4a illustrates a two-thread execution without a power failure. Initially, high-priority Thread 2 waits, for example, on a queue. A low-priority Thread 1 uses `sys_queue_send_back` to enqueue data, allowing Thread 1 enabled (its waiting condition is satisfied). During the system call, the kernel-side transaction $\texttt{TX}_3$ updates the kernel queue object in NVM. As Thread 1 is awakened and has a higher priority, the INTOS scheduler preempts Thread 1 and context-switches to Thread 2 by modifying thread-related persistent linked lists, such as `ready-list` and `wait-list`, in transaction $\texttt{TX}_4$. It is a common pattern for a system call to update a system call-specific kernel data structure (*e.g.,* queue) in one transaction and to modify schedule-related linked lists in another transaction. After the context switch, Thread 2 runs, and Thread 1 remains on the `ready-list`, awaiting its turn.

Let's first consider a simple power failure case. If power is lost during the system call (*e.g.,* during $\texttt{TX}_3$ or $\texttt{TX}_4$ or between them), it constitutes a single-thread scenario. The recovery protocol remains the same as the case presented in Figure 3f.

Suppose a power outage occurs while running Thread 2 (after the context switch) as depicted in Figure 4b. This makes a multi-thread scenario: Threads 1 and 2 are runnable. Upon power restoration, INTOS recovers and runs Thread 2 — the thread that was running and experienced a power failure. The priority-based scheduler always schedules the thread with



(a) A two-thread execution without a power failure. Initially, a high-priority Thread 2 is waiting. A low-priority Thread 1 makes it ready.

(b) A power failure occurs after Thread 2 is scheduled.

(c) A two-thread execution without a power failure. Initially, a high-priority Thread 1 runs first and makes a blocking system call, yielding a turn to a low-priority Thread 2.

(d) A power failure occurs after Thread 2 is scheduled.

Figure 4: Multi-threads recovery examples

the highest priority among ready threads. Thus, recovering the failed thread implies that when power becomes available, INTOS runs the ready thread with the highest priority. In this example, it is Thread 2. INTOS roll-backs transaction TX$_5$ (step ①) and replays Thread 2 (step ②). On the other hand, INTOS does not eagerly undo transaction TX$_2$ (step ③). Hard-won energy should not be wasted. A system may not possess enough energy to run Thread 1 (after Thread 2). High-priority ready Thread 2 takes precedence over undoing the transaction TX$_2$ of Thread 1. Sometime later, when Thread 1 is scheduled, INTOS then rolls back transaction TX$_2$ on demand (step ③) and replays Thread 1 (step ④).

When managing multiple threads, a blocking system call warrants detailed discussion. Figure 4c depicts another two-thread execution without a power failure, distinct from Figure 4a. In this scenario, Thread 1 possesses high priority, and even though Thread 2 is ready, it is not scheduled. Suppose the queue is already full. Assuming the queue is already full, when Thread 1 employs the `sys_queue_send_back` system call to enqueue data, it discovers the queue lacks space and becomes blocked. Subsequently, the scheduler moves Thread 1 to the `wait_list` in transaction TX$_4$. In this scenario, it is crucial to note that TX$_3$ is indeed a null transaction, making no updates to the queue. An essential invariant established by the INTOS kernel is that a blocking system call, if it actually blocks, does not alter the state of system call-specific persistent objects (*e.g.,* queue). The system call is not considered complete, and no result is cached for bypassing. The impact of a blocking call is confined to schedule-related linked lists in TX$_4$. Given that a blocking system call has no substantive effect on kernel states, it is safe to proceed with the same recovery and replay of the ready thread with high priority — Thread 2 in this example. Any processing for the blocked threads, such as Thread 1, can be deferred, as illustrated in Figure 4d. When Thread 1 is later scheduled for recovery and replay, it will re-invoke the system call as if it had not been issued previously. In the INTOS implementation, those system calls that may block always first check for a blocking condition to uphold this invariant.

6 INTOS Programming Model

INTOS's programming model upholds five rules designed to guarantee crash consistency.

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged. Modifications on persistent objects outside transactions are untracked. Therefore, any update to persistent objects should be confined within transactions. INTOS also prohibits the reading of persistent objects outside transactions to prevent potential divergence in program control flow during replay. When restarting, non-volatile memory states are not rolled back to the thread's outset. For example, in the scenario illustrated in Figure 3c, replay begins with non-volatile memory states still reflecting the state s$_1$ after TX$_1$. Consequently, control flow outside transactions should not rely on persistent objects. To precisely identify a subset of persistent object reads that may influence control flows, one can perform static analysis and selectively prevent them. INTOS, for simplicity, conservatively enforces the restriction of no reads (and writes) outside transactions. This approach does not overly constrain programmability since it is natural to assume that persistent objects are primarily used within transactions. Furthermore, INTOS permits a transaction to acquire references to persistent objects that were created or modified by another transaction and subsequently update them arbitrarily within the executing transaction, as demonstrated in Listing 1 (Lines 6, 22-23).

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value. Rule 2 further enforces Rule 1. Allowing the return of references would potentially enable users to directly modify persistent objects without proper logging. Mutable references should be acquired and dereferenced exclusively within a transaction, as exemplified in Listing 1 (Lines 22-23).

Rule 3: Persistent objects should not contain references to volatile objects. Volatile objects are susceptible to data loss during power failures. Storing their references in persistent objects is thus unsafe.

Rule 4: System calls (excluding Locks) should only be made within transactions. There is, in theory, no fundamental restriction against using a system call outside a transaction for crash consistency. Yet, INTOS mandates adherence to this rule to constrain the length of system call replay and bound memory resources. After a transaction concludes, there is no necessity to replay any system call within that transaction. As a result, the upper limit for system calls to be replayed is determined by the number of system calls in the last uncommitted transaction. INTOS can safely free the system call replay metadata for committed transactions.

Rule 5: Locks should not be used inside transactions. A critical section, defined by locks, should be larger than a transaction. Suppose two concurrent transactions, `TX`$_1$ and `TX`$_2$, utilize a lock when accessing a shared object `X` within transactions. `TX`$_1$ acquires the lock, updates `X`, releases the lock, but remains uncommitted. The concurrent `TX`$_2$ acquires the lock, reads `X`, performs some computation, releases the lock, and eventually commits. If a power failure occurs at this point while `TX`$_1$ remains uncommitted, a data consistency issue arises. This occurs because our transaction lacks “isolation” among concurrent transactions. Rule 5 is enforced to avoid this problem. Ultra-low power intermittent computing systems hardly use multi-cores. Introducing a more intricate yet efficient solution, such as tracking data dependencies between transactions and aborting one if a conflict is detected, doesn’t appear necessary in this context.

Enforcement INTOS employs Rust’s robust type system to uphold the aforementioned rules, akin to [33] that statically prevents common persistent memory programming errors within the realm of server-side (non-energy-harvesting) persistent memory programming. Rules 1-3 resemble those in [33], with INTOS extending Rule 1 to disallow reading persistent objects outside transactions to avoid potential control flow divergence during replay. Rules 4-5 are distinctive to INTOS. The implementation utilizes Rust’s traits.

7 Optimization

INTOS employs three performance optimizations.

7.1 Loop Optimization

Threads in embedded systems often involve loops, such as event loops handling sensor readings or loops with numerous iterations, as seen in matrix multiplication for neural network machine learning threads. Consider a thread with a loop where the loop body comprises T transactions, and a power crash occurs on the N-th iteration. While INTOS’s replay-and-bypass approach can bypass $(N-1)*T$ transactions (in addition to any committed transaction in the last iteration), the overall replay window’s length could potentially be excessively long, leading to substantial energy consumption during replay.

To address this common scenario, INTOS introduces the new `nv_for_loop!` macro, extending the loop construct in Rust to utilize a non-volatile variable as the iteration counter. With the non-volatile iteration counter, INTOS can infer completed iterations (committed transactions therein) during replay, enabling a safe and efficient fast-forward to the last iteration without executing the bypass logics.

INTOS’s Rust language enforces the absence of loop-carried dependent volatile states to safely employ `nv_for_loop!` optimization as it skips iterations during recovery. Users are still able to employ volatile variables within a loop body, provided there is no loop-carried dependency.

7.2 Linked List Optimization

The INTOS kernel extensively utilizes doubly-linked lists to manage threads and scheduling states. Nearly every system call involves the manipulation of these linked lists. Notably, we have identified optimization opportunities, recognizing that linked list updates within the kernel occur within a critical section, eliminating the need to account for arbitrary interleaving. Additionally, there are no intermediate volatile hardware buffers (such as store buffer or cache) between registers and non-volatile memory. Consequently, any store instruction promptly persists as it retires from the pipeline. With these factors combined, we can scrutinize the crash non-volatile state, reason through intermediate linked list update steps, and precisely identify the power failure point.

INTOS presents *crash state analysis*-based roll-forward recovery optimization for linked list transactions. There is no undo-logging. Instead, INTOS records an operation log including the type (*e.g.,* insert) and the node (data) — only one per operation. During recovery, INTOS analyzes the crash

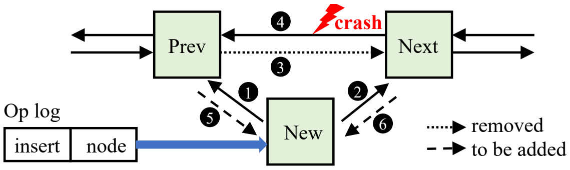


Figure 5: Crash state analysis to roll forward a list insertion

state remaining in NVM to infer the steps that have been completed. Then, it rolls forward the rest of the operation.

Consider an insertion transaction, illustrated in Figure 5. The process of list insertion involves six ordered steps. Initially, we link the node to its previous and next node (step ❶ - ❷). Then, the backward/forward link between the next node and the previous node is removed (step ❸ - ❹). Finally, we insert the new forward/backward link between the previous/next node and the new node (step ❺ - ❻). Suppose a power crash occurs before step ❹. During recovery, the operational log is first retrieved to determine the operation type and the node involved. It is discovered that the link from the new node to the previous/next node already exists, indicating that steps ❶ - ❷ are completed. It is also found that the forward link from the previous to the next node is removed, but the backward link remains intact. This observation suggests that the crash occurred before step ❹ was completed. IntOS can roll forward the operation by executing steps ❹ - ❻.

7.3 Undo-Logging Optimization

The default IntOS transactions automatically perform undo-logging on every first write (after acquiring a mutable reference), as in Listing 1 (Lines 22-23). IntOS introduces another smart pointer type, `Ptr<T>`, providing an option to leverage Rust's type system for the static detection of write-after-read (WAR) dependencies. A transaction utilizing `Ptr<T>` logs an old value only if there is a WAR dependency in the transaction, resulting in fewer logs. Within a transaction, users should dereference a persistent object pointer to obtain a reference. `Ptr<T>` does not provide users with a raw reference and imposes restrictions on the access interface, such as `r.read()` and `r.write()`. Consequently, utilizing `Ptr<T>` involves some additional coding efforts.

8 Discussion

Transactions for Partial vs. Whole System Persistence A crucial distinction between PMDK [7] (`libpmemobj`) and IntOS transactions lies in their persistence guarantees. `libpmemobj` supports "partial" system persistence, only ensuring the recoverability of non-volatile objects within transactions. Thus resuming program execution often requires user-defined custom crash-recovery logic to achieve consistent whole system states including volatile ones. In contrast, IntOS offers "whole" system persistence through the proposed replay-and-bypass mechanism, guaranteeing the recovery of both persistent and volatile states.

Transaction Length To ensure forward progress, IntOS mandates that a transaction must be able to complete with a fully charged capacitor. IntOS handles only one ready, highest-priority thread at a time and employs replay-and-bypass mechanisms to skip committed transactions and system calls, ensuring progress as long as one transaction successfully passes each power cycle. IntOS asks users to ensure this property via profiling. Bounding the size of a program region is a common requirement for many intermittent computing systems (*e.g.,* an idempotent region, a failure atomic section, and a transaction in IntOS) to ensure stagnation-free execution. Consequently, previous solutions including Choi et al. [19] have proposed various dynamic (profiling) and static program analysis techniques considering the worst-case behaviors. IntOS's kernel transactions are intentionally designed to be brief, considering this constraint. Our evaluation (§10) reports the maximum number of cycles per transaction in tested applications is short enough.

Energy-aware Scheduler If hardware provides a capability to monitor the remaining energy in the capacitor, one can design an energy-aware scheduler in IntOS: *e.g.,* not scheduling a thread if it is soon to stop.

Real-time Capabilities IntOS provides real-time capabilities comparable to FreeRTOS as long as the power is on. Yet, IntOS does not provide (hard) real-time guarantees due to the non-deterministic energy nature inherent in intermittent computing, rendering such assurances impossible.

Rust Rust is chosen for static correctness guarantees. Users can use C or other languages, provided they adhere to the programming rules (§6). It is feasible to statically link C programs with the Rust IntOS kernel since the contract/interface between the kernel and a user program is well-defined. Using C would require complex static program analysis to verify adherence to the programming rules. Additional static analysis should be employed for automatic undo-logging.

9 Implementation

We implement IntOS using the Rust programming language, leveraging its strong static type system to uphold IntOS's programming model (§6) with performance comparable to C. The initial implementation of the IntOS kernel mirrors FreeRTOS, having been ported to Rust and extended with transaction and crash consistency support. User threads are also crafted in Rust. Presently, IntOS extends support to two architectures: ARM Cortex-M4 and MSP430. The overall IntOS implementation, excluding testing and benchmark code, encompasses approximately 9900 lines of Rust code. We elaborate on some details below:

Multithreading The INTOS kernel allocates essential data structures, such as the thread control blocks, inter-thread communication objects (*e.g.,* queue, semaphore), and scheduling lists (*e.g.,* ready-list, wait-list) in non-volatile memory. Table 2 (last column) lists some examples.

Replay Tables To support *replay-and-bypass* recovery, INTOS maintains three per-thread replay tables that cache the return values of user-level transactions, kernel-level transactions, and system calls. For each table, the `tail` pointer indicates the last completed transaction or syscall, and the `current` pointer points to the presently executing one. The transaction `tail` pointer contains the commit flag, transaction id, and the pointer to the replay table.

Log Sizes The logged results of system calls generated within a transaction are garbage-collected upon the completion of each transaction, thereby bounding the maximum length of system call logs. Upon the completion of a task, all transaction logs associated with that task can be cleaned. We assume that a task entails a finite number of transactions, which is typically valid given that embedded application tasks often serve as short event handlers. An exception arises with tasks executing transactions within a loop, potentially leading to unbounded logs. This scenario is addressed by the `nv_for_loop!` optimization (§7.1). Transaction result logs from completed (old) loop iterations can be safely discarded, thus capping the transaction log size per loop iteration.

10 Evaluation

We evaluate the performance of INTOS on two platforms: MSP430FR5994 [6] and Apollo 4 Blue Plus [1]. MSP430FR5994 features 256KB of non-volatile FRAM and 8KB of volatile SRAM. We configured its MCU to operate at 16MHz. The Apollo 4 Blue Plus is equipped with an ARM Cortex-M4 processor. It has 384KB of TCM (faster SRAM), 2MB of SRAM, and 2MB of non-volatile MRAM.

The benchmark suite comprises 11 applications. The first group encompasses three single-thread applications (BC, AR, MLP), utilized in previous studies [22,38,49,50,53,56,64,65]. The second group comprises four multithreaded applications (KV, SEN, EM, MQ) designed to evaluate the performance of INTOS's OS features, including locks (semaphores), timers, events, and queues, respectively. The final macro-benchmark group comprises four multithreaded applications (ETL, PRED, STATS, TRAIN), adapted from RIoT-Bench [58]. Table 3 provides the application name, description, the number of threads, transactions, and system calls. The last three columns will be discussed later.

We compare the following four configurations:

- **SRAM (not crash consistent, baseline)**: A vanilla application and INTOS kernel without crash consistency support (*i.e.,* no undo-logging, no replay-and-bypass) operate on volatile SRAM. All the data is in SRAM, while the code is stored on FRAM due to space limitations. In the event of a power outage, both register and SRAM memory states are lost. This configuration serves as the baseline.

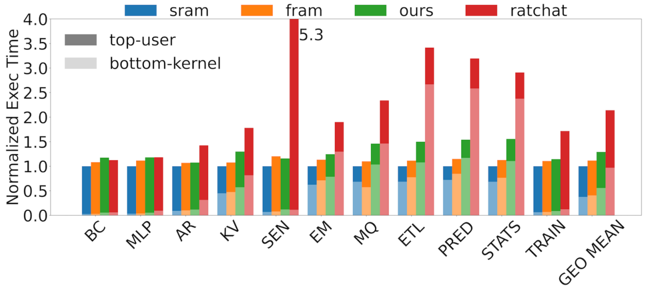


Figure 6: Latency overhead without power failure on MSP430

- **FRAM (not crash consistent)**: A vanilla application and INTOS kernel without crash consistency support (*i.e.,* no undo-logging, no replay-and-bypass) runs on non-volatile FRAM. All data resides in FRAM. Volatile registers remain susceptible to loss. This setup underscores the limitations of not utilizing SRAM and establishes the lower bound for existing compiler-based checkpointing solutions assuming no volatile memory (§3).
- **INTOS (crash consistent)**: This configuration represents our approach using both SRAM and FRAM. It uses INTOS's transaction undo-logging and replay-and-bypass recovery to ensure whole-system crash consistency.
- **Ratchet (crash consistent)**: Ratchet [62] represents a state-of-the-art compiler-based idempotent processing solution that uses non-volatile FRAM only. We used Ratchet compiler to transform a vanilla application and INTOS kernel to idempotent regions — with neither undo-logging nor replay-and-bypass.

It is worth noting that we were unable to compare INTOS with ImmortalThreads [65] due to the incomplete nature of the publicly available code. It offers only the essential logic for micro-continuations and lacks OS/runtime features required by the tested benchmarks (*e.g.,* blocking queues). It was originally evaluated with four simple single-threaded Bitcount (BC), Cuckoo Filter (CF), Activity Recognition (AR), and DNN, which do not involve any application-OS interactions. Thus, conducting a fair comparison becomes impractical without ImmortalThreads' supplementary runtime support. Nonetheless, as discussed in §3, we expect its micro-continuation would suffer from high runtime overhead. For example, ImmortalThreads reports (See [65] Table 4 and Figure 7) that AR incurs 237% overhead with no failure and 300% with 5ms-period power failure. In contrast, we later show that in INTOS, AR experiences 8% and 15% overheads, respectively (See Figure 6 and Figure 8).

App	Description	Threads	TXs	Syscalls	Max Cycles/TX	LoC	Add&Mod
BC	Count the number of 1s in an integer using multiple algorithms	1	8	1	10676	181	32
MLP	Multi-layer perception with two fully connected layers	1	4	2	3488	155	30
AR	Train an activity recognition model and analyze the activities	1	3	3	12060	301	33
KV	Two threads perform concurrent operations on KV Store with locks	2	9	23	6276	325	102
SEN	Periodic Sensing using software timers	2	3	4	6420	107	15
EM	One thread monitors events and notifies other threads with event groups	3	6	12	2592	113	29
MQ	One thread distribute messages to other threads using queues	4	6	13	2532	166	51
ETL	Extract, Transform and Load dataflow in RIoTBench [58] (*e.g.,* range filter, bloom filter, interpolation, join, annotation, kv store)	5	10	23	3580	709	148
PRED	Predictive analysis dataflow in RIoTBench [58] (*e.g.,* average, kalman filter, distinct count, sliding linear reg., kv store)	5	9	26	3808	440	58
STATS	Statistic summerization dataflow in RIoTBench [58] (*e.g.,* decision tree, multivar linear reg., average, error estimation, kv store)	5	11	27	4884	413	46
TRAIN	Model training dataflow in RIoTBench [58] (*e.g.,* multivar linear reg. training, decision tree training)	4	20	28	9472	511	132

Table 3: Description for Benchmarks and Statistics

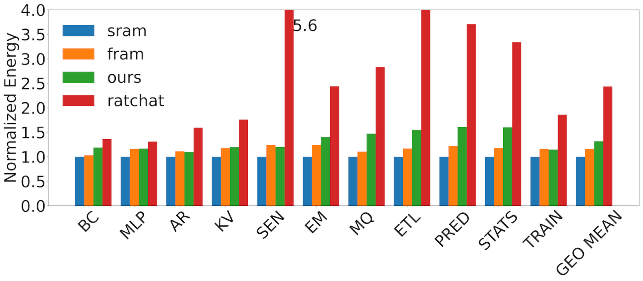


Figure 7: Energy overhead without power failure on MSP430

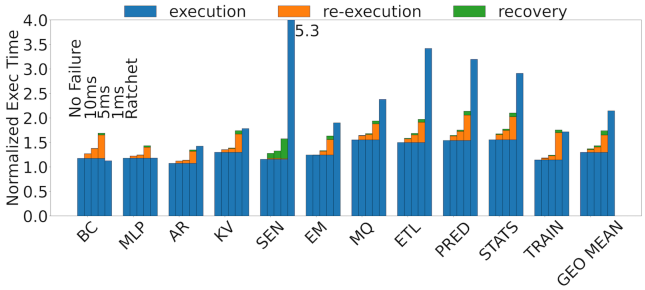


Figure 8: Latency overhead with power failure on MSP430

10.1 Without Power Failures on MSP430

We first measure the performance and energy overhead without power failure on MSP430FR5994.

Figure 6 illustrates the latency overhead of four configurations, normalized to the SRAM baseline. Each bar provides a breakdown between user-level (top, solid color) and kernel-level (bottom, light color) execution times. On a geometric mean, FRAM (the second bar) shows 1.11x latency overhead compared to SRAM. This highlights the performance loss when SRAM is not utilized, as in existing compiler-based checkpointing solutions. This serves as the lower bound for the latency overhead imposed by such tools. Specifically, Ratchet (the last bar) incurs a latency overhead ranging from 1.12x to 5.30x, with a geometric mean of 2.14x. Ratchet's performance is highly dependent on the precision of static analysis and application characteristics.

Contrastingly, INTOS (the third bar) demonstrates substantially lower latency overhead, ranging from 1.07x to 1.55x, with a geometric mean of 1.29x. This showcases the advantages of placing the stack and performing computations on local variables in SRAM while storing persistent objects in FRAM. Notably, for AR, SEN, and TRAIN, INTOS demonstrates comparable or superior performance to FRAM, even considering INTOS's transaction logging overhead.

Regarding the breakdown between user and kernel levels, simple single-thread BC, MLP, and AR predominantly operate in the user level, while multi-threaded RIoTBench's ETL, PRED, STATS, and TRAIN frequently utilize system calls for queues, mutexes, etc. The SEN application conducts periodic sensing using software timers. It displays a small kernel (syscall) time, as the kernel's timer handler indeed runs as a thread and is thus counted as user time.

Figure 7 illustrates the energy overhead of four configurations, normalized to the SRAM-only baseline. We measured the energy consumption for MSP430FR5994 using TI's EnergyTrace tool [11]. The observed trend aligns generally with the latency overhead discussed earlier. The FRAM setting incurs 1.16x (geometric mean) more energy consumption compared to the SRAM setting. The energy consumption gap between INTOS and Ratchet widens, with INTOS consuming 1.31x more energy on a geometric mean relative to the baseline, while Ratchet consumes 2.43x more energy. Notably, across various applications, including MLP, AR, KV, SEN, and TRAIN, INTOS exhibits comparable or superior performance to the FRAM-only setup, even when factoring in INTOS's transaction overhead.

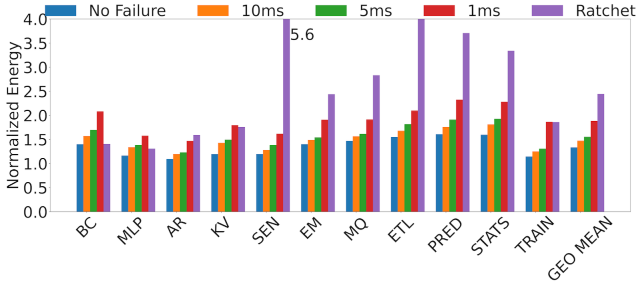


Figure 9: Energy overhead with power failure on MSP430

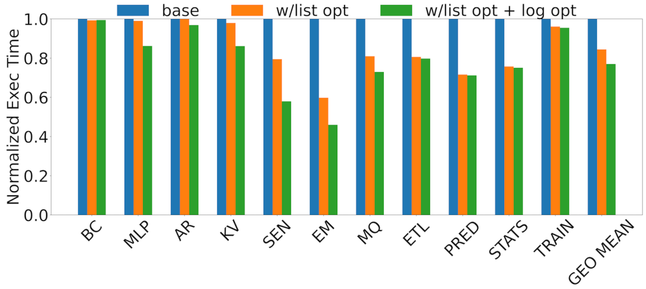


Figure 10: Latency with/without optimizations on MSP430

10.2 With Power Failures on MSP430

In this section, we investigate the latency and energy overhead of INTOS and Ratchet under frequent power failures. We inject controlled power failure at regular intervals of 10ms, 5ms, and 1ms using a soft reset, following the methodology employed in previous works [37,38,49,62,64,65]. Intermittent computing devices continue operation until the energy stored in the capacitor is depleted, subsequently restarting after the capacitor is fully recharged. In cases where the capacitor can recharge during the run, it results in extended run time for the power cycle. Employing a regular power failure interval represents the worst-case scenario, where the capacitor cannot be effectively charged during execution.

To determine and justify the power failure interval, we ran our benchmarks and used EnergyTrace to measure the average power consumption of MSP430FR5994 and the number of MCU cycles spent for each time interval. We considered the maximum power consumption (among applications) and calculated the corresponding capacitor size under a 3V voltage. To sustain continuous operation for 1ms, where MSP430 MCU can run for about 16,000 cycles, the capacitor size required is approximately 4 µF, which is ten times smaller than a typical capacitor size (*e.g.,* WISP [57] utilizes 47 µF). Thus, the 1ms interval represents an extreme case.

INTOS requires each transaction to be completed with a fully charged capacitor to guarantee forward progress. The third last column in Table 3 displays the maximum number of cycles per user transaction in tested applications, indicating that an application can be implemented with a (relatively) short transaction. Should a longer transaction be desired, INTOS might necessitate a larger capacitor.

Figure 8 depicts the latency overhead under power failures on MSP430. Moving from left to right, the bars represent the latency overhead of INTOS in a no-failure scenario, with failure intervals of 10ms, 5ms, 1ms, and Ratchet – all of which are normalized to the SRAM setting (baseline). Note that as an idempotent processing solution, Ratchet exhibits negligible latency difference between with and without power failures. Across different applications, we observed 30-900 power outages with the 1ms failure interval, and 3-60 power failures with the 10ms interval. As anticipated, the latency overhead of INTOS increases with the frequency of power failures. On a geometric mean, INTOS exhibits latency overheads of 1.37x, 1.43x, and 1.73x for 10ms, 5ms, and 1ms intervals, respectively. INTOS's recovery mechanism involves restoring volatile states (while bypassing numerous committed transactions and system calls), making its latency sensitive to the failure frequency. However, for the 10ms, 5ms, and even in the extreme 1ms failure intervals, INTOS demonstrates significantly better performance than Ratchet, particularly when considering realistic complex applications like RIoTBench's ETL, PRED, and STATS, while excluding trivial single-thread applications like BC and MLP.

The figure also provides a breakdown of the latency overhead between re-execution (orange bar) and recovery (green bar). Re-execution overhead involves rerunning an interrupted program region, representing wasted computation, while bypassing committed transactions and system calls. Recovery overhead is incurred by applying undo logging to roll back a failed transaction and executing other basic recovery checking codes. The results indicate that INTOS's latency overhead is predominantly attributed to re-execution overhead. SEN is unique in that it uses software timers, so in most cases, it has no task to run but simply checks for recovery.

Figure 9 illustrates the energy overhead under the same power failure experiments on MSP430. The energy overhead follows a similar trend as the latency overhead. INTOS consistently demonstrates a lower energy profile than Ratchet across all failure intervals, especially when considering realistic applications ETL, PRED, and STATS. On a geometric mean, INTOS exhibits energy overheads of 1.47x, 1.55x, and 1.88x for 10ms, 5ms, and 1ms intervals, respectively. In comparison, Ratchet incurs an energy overhead of 2.43x.

10.3 Optimization Effectiveness on MSP430

This section investigated the impact of linked list optimization (§7.2) and undo logging optimization (§7.3). Each optimization was individually enabled, and the execution time was measured. The results, normalized to INTOS with no optimizations (the first bar), are presented in Figure 10. The second bar illustrates the outcomes with only the list optimization enabled, while the last bar represents the results with both

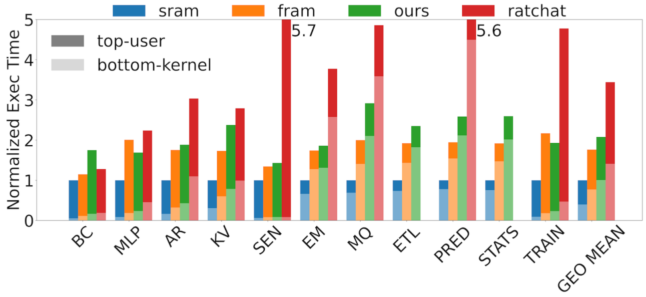


Figure 11: Latency without power failures on Apollo 4

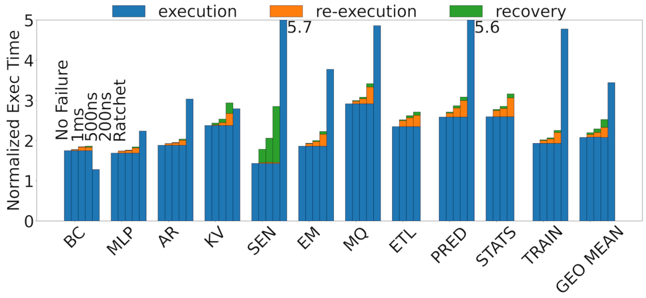


Figure 12: Latency with power failures on Apollo 4

optimizations in use. For applications with frequent system call usage, the list optimization significantly enhances performance, with improvements exceeding 40%. However, for simpler single-thread BC, MLP, and AR, which only utilize memory allocation syscalls, there was marginal improvement. The effectiveness of the undo-logging optimization is highly dependent on application characteristics. MLP, KV, SEN, EM, and MQ have a small fraction of stores with write-after-read dependencies. Thus the undo logging optimization demonstrates substantial improvements.

10.4 Experiments with Apollo 4

Now, we transition our experiment to the Apollo 4 Blue Plus, equipped with an ARM Cortex-M4 MCU, 384KB of TCM (faster SRAM), 2MB of SRAM, and 2MB of non-volatile MRAM. However, it is important to acknowledge that the MRAM in Apollo is presently only byte-readable and not byte-writable. To overcome this constraint, we simulate the execution environment by utilizing (fast) TCM as volatile memory and designating the (slow) SRAM as non-volatile memory. In our experiment, the SRAM is approximately 2-3 times slower than TCM for sequential access, which is bigger than the FRAM-SRAM gap in MSP430. The (simulated) Apollo 4 experiment has two purposes. First, it demonstrates that INTOS can support different MCU architectures: MSP430 and ARM Cortex-M4. Second, it illustrates a scenario in which the latency disparity between volatile and non-volatile memories is more pronounced. The board does not have an on-board debugger probe that allows us to measure the energy, so this experiment focuses on latency comparison.

Figure 11 shows the latency overhead of Apollo 4 Blus Plus without a power failure, normalized to the TCM-only baseline. The ETL and STATS bars are missing for Ratchet because the programs instrumented by Ratchet crashes. With the higher gap between volatile and non-volatile memories (simulated by TCM and SRAM), the result shows higher latency overheads than the MSP430 experiments (Figure 6). INTOS and Ratchet incur 2.07x and 3.44x latency overhead, respectively, where Ratchet is more penalized by slow non-volatile memory.

Figure 12 shows the latency overhead when considering power failure intervals of 1ms, 500ns, and 200 ns. The intervals are set to be much smaller than those of MSP430 as ARM Cortex-M4 in Apollo 4 runs at a much higher clock frequency. 200 ns allows around 19,000 cycle executions. The trend again remains the same. Even in the extreme case of 200 ns failure interval, INTOS incurs 2.52x latency overhead (compared to SRAM). INTOS is 1.37x less than Ratchet.

10.5 INTOS Programming Overhead

The INTOS programming model asks users to allocate persistent objects in NVM and define transactions to ensure crash consistency of updates on persistent objects. Quantifying programming overhead is challenging, but as a proxy, Table 3 presents the lines of source code (LOC) for each application and the added/modified LOC for persistent object allocation and transaction codes. Examining four realistic RIoTBench applications, the table reveals that the extent of modification varies from 11% (STATS: 46/413) to 26% (TRAIN: 132/511) of the source code. Although these percentages may seem large, it is important to note that these changes pertain to persistent object allocation and transaction codes, aspects that we believe are well-understood and manageable.

11 Conclusion

INTOS is a persistent embedded OS and language support for multi-threaded intermittent computing. INTOS uses transactions to ensure the crash consistency of non-volatile objects. Instead of checkpointing volatile states, INTOS proposes a replay-and-bypass recovery mechanism, reconstructing volatile states without re-executing committed transactions and system calls. Evaluation with MSP430FR and Apollo 4 shows that INTOS exhibits lower latency and energy costs compared to compiler-based idempotent processing.

Acknowledgements

We appreciate the valuable feedback from anonymous reviewers and the shepherd. This work is in part supported by the NSF grants CNS-2135157, CCF-2153747, CCF-2153748, CCF-2153749, CNS-2314681, and CNS-2214980.

References

[1] Apollo4 Blue Plus. https://ambiq.com/apollo4-blue-plus/.

[2] ARM Cortex-M4. https://developer.arm.com/Processors/Cortex-M4.

[3] FreeRTOS. https://www.freertos.org/index.html.

[4] Intel Optane Memory. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[5] KickSat. https://kicksat.github.io/.

[6] MSP430FR5994. https://www.ti.com/product/MSP430FR5994.

[7] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[8] Persistent Redis v3.2. https://github.com/pmem/redis/tree/3.2-nvml.

[9] Pmem-Memcached. https://github.com/lenovo/memcached-pmem.

[10] The MRAM on the Apollo 4 Processor. https://www.techinsights.com/blog/memory/disruptive-technology-tsmc-22ull-emram.

[11] TI's EnergyTrace software for MSP430™ MCUs. https://www.ti.com/tool/ENERGYTRACE.

[12] TI's FRAM. https://www.ti.com/lit/wp/slat151/slat151.pdf.

[13] Zephyr. https://www.zephyrproject.org/.

[14] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Gunes, and Thomas Schmidt. *RIOT: One OS to rule them all in the IoT*. PhD thesis, INRIA, 2012.

[15] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. IPSN '17, page 209–219, 2017.

[16] Luca Caronti, Khakim Akhunov, Matteo Nardello, Kasım Sinan Yıldırım, and Davide Brunelli. Fine-grained hardware acceleration for efficient batteryless intermittent inference on the edge. *ACM Trans. Embed. Comput. Syst.*, 22(5), sep 2023.

[17] Wei-Ming Chen, Tai-Sheng Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. Value-based task scheduling for non-volatile processor-based embedded devices. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 247–256, 2016.

[18] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–344. IEEE, 2019.

[19] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–54, 2022.

[20] Jongouk Choi, Qingrui Liu, and Changhee Jung. Cospec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 399–412, 2019.

[21] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. Write-light cache for energy harvesting systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[22] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 514–530, 2016.

[23] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 475–486, 2012.

[24] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3), sep 2020.

[25] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

[26] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: Using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, page 29–39, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021.

[28] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 199–213, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Philipp Gutruf, Vaishnavi Krishnamurthi, Abraham Vázquez-Guardado, Zhaoqian Xie, Anthony Banks, Chun-Ju Su, Yeshou Xu, Chad R Haney, Emily A Waters, Irawati Kandela, et al. Fully implantable optoelectronic systems for battery-free, multimodal operation in neuroscience research. *Nature Electronics*, 1(12):652–660, 2018.

[30] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017.

[31] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, 2017.

[32] Matthew Hicks. Clank: Architectural support for intermittent computation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 228–240, 2017.

[33] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 429–442, 2021.

[34] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, page 193–204, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 330–335, 2014.

[37] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. Wario: Efficient code generation for intermittent computing. PLDI 2022, page 777–791, 2022.

[38] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. ASPLOS '20, page 85–99, 2020.

[39] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. Intermittent learning: On-device machine learning on intermittently powered system. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(4), sep 2020.

[40] Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *2012 IEEE International Solid-State Circuits Conference*, pages 402–404, 2012.

[41] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[42] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, 2017.

[43] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[44] Qingrui Liu and Changhee Jung. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2016.

[45] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Notices*, 50(5):1–10, 2015.

[46] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 228–239. IEEE, 2016.

[47] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 575–585, 2015.

[48] Yubo Luo and Shahriar Nirjon. Smarton: Just-in-time active event detection on energy harvesting systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 35–44, 2021.

[49] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[50] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 129–144, 2018.

[51] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1101–1116, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1005–1021, 2020.

[53] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sen. Netw.*, 16(1), feb 2020.

[54] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, ENSsys '19, page 8–14, 2019.

[55] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. ASPLOS XVI, page 159–170, 2011.

[56] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. PLDI 2019, page 1085–1100, 2019.

[57] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[58] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21), October 2017.

[59] Milijana Surbatovich, Limin Jia, and Brandon Lucia. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 851–866, 2021.

[60] Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. A type system for safe intermittent computing. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[61] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. Capband: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 54–67, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 17–32, 2016.

[63] Harrison Williams, Xun Jian, and Matthew Hicks. Forget failure: Exploiting sram data remanence for low-overhead intermittent computation. ASPLOS '20, page 69–84, 2020.

[64] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 41–53, 2018.

[65] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. Immortal threads: Multithreaded event-driven intermittent computing on Ultra-Low-Power microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 339–355. USENIX Association, July 2022.

[66] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. Replaycache: Enabling volatile cachesfor energy harvesting systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 170–182, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in zebranet. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, page 227–238, New York, NY, USA, 2004. Association for Computing Machinery.

[68] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. Sweepcache: Intermittence-aware cache on the cheap. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1059–1074, 2023.

A Artifact Appendix

Abstract

IntOS is an intermittent multithreaded embedded RTOS based on FreeRTOS for research. It features transactions with replay and bypasses to enable cheap crash consistency. The kernel and user applications are all written in the Rust programming language. The Rust type system is used to enforce safe programming rules defined by the framework to guarantee crash consistency and Persistent Memory safety. Currently, we support three platforms: QEMU, Apollo 4 Blue Plus, MSP430FR5994.

Scope

This artifact contains code to build the crash-safe INTOS kernel and benchmark/user app to run on intermittent computing platforms(e.g. MSP430FR5994). Users can use the artifact to reproduce the results in the paper. For evaluation of functionality, please just follow the instructions for QEMU. This artifact is for research purposes only.

Contents

The artifact contains Rust written kernel, user library code, and the benchmarks used in the paper. The general kernel code is under the `src/` directory. The user library is in `src/user`. Benchmarking code is under `src/benchmarks`. Platform/Architecture-related code is in `src/arch` and `src/board` directory. A simple demo app is hosted under `src/app`.

Hosting

The artifact is hosted in https://github.com/yiluwusbu/IntOS. The branch is master and the commit version is a916c16

Requirements

The OS we use is Ubuntu 22.04. For evaluating the functionality and debugging, QEMU(for ARM) is sufficient. For evaluating the performance, you need to get the MSP430FR5994 or Apollo 4 Blue Plus development board.

Run with Docker

We provide a docker image for users to run the system with QEMU. If you use docker, please skip the dependency/-toolchain installation sections. To build the docker image, run:

```
docker build -t rtosdev .
```

Then, run the docker:

```
docker run -v $(pwd):/repo -it rtosdev bash
```

Install System Dependency

```
sudo apt install curl wget p7zip-full
libncurses5 libncursesw5 build-essential
qemu-system-arm
```

Install Rust Toolchain

```
curl --proto '=https' -tlsv1.2 -sSf
https://sh.rustup.rs | sh -s -- -y
```

Set the compiler version:

```
rustup toolchain add nightly-2022-04-01
```

Install MSP430 Toolchain

Download and install the msp430-gcc toolchain from TI's website. For detailed commands, see README.md in the github repo.

Install ARM Toolchain

Install JLink flasher/debugger and the ARM gcc toolchains:

1. Download the Segger JLink tools (v7.92) on your platform from their website

2. Download ARM (arm-none-eabi) toolchain (version 12.3.Rel1) from the official ARM website

Compile INTOS

You can compile the OS and benchmarks/example applications using the provided Python script:

```
./compile.py --board [qemu|apollo4bp|msp430fr5994]
--bench [app name] [--run (for qemu)]
```

Example:

```
./compile.py --board qemu --bench bc --run
```

Configuration Parameters

To list all the available benchmarks and custom compilation flags, you can run:

```
./compile.py -h
```

Table 3 describes the benchmarks we use in this work. To enable timer daemon, you can pass `--timer_daemon`

Power Failure Injection

To inject soft power failure to the system at a given frequency, you can use the following command:

```
./compile.py --board [board name] --bench [app
name] --fail --pf_freq [frequency: e.g. 1ms]
[--run (for qemu)]
```

Example:

```
./compile.py --board qemu --bench bc --fail
--pf_freq 1ms --run
```

Run Demo App

We give a simple example of two tasks communicating using a Queue (i.e. IPC). The full code can be found in the `app/demo.rs` file.

To run the demo:

```
./compile.py --board qemu --app demo --run
```

Flash and Run App on MSP430FR5994

You can install the TI's Uniflash or CCSTUDIO IDE to flash the application binary (located under target/msp430-none-elf/release/) onto the board.

The application/OS will print debug/perf related information through the UART interface to the host machine. The default Baud Rate is 115200. To view the printed message, you can use any Serial Monitor tools to view the printed message. For example, on Linux/Win, you can install the Serial Monitor Plugin. Termite is another handy tool you can use.

Flash and Run App on Apollo4 Blue Plus

Use GDB and JLink to load and run the application.

1. In one terminal, run `JLinkGDBServer -if SWD -device AMA4B2KP-KXR`

2. In another terminal run `arm-none-eabi-gdb -x apollo.gdb <path/to/binary>`

After the binary is loaded onto the board, enter ’c’ to run. The application will print message to the gdb interface and port 2333 (TCP/IP) .

Data-flow Availability: Achieving Timing Assurance on Autonomous Systems

Ao Li Ning Zhang
Washington University in St. Louis

Abstract

Due to the continuous interaction with the physical world, autonomous cyber-physical systems (CPS) require both functional and temporal correctness. Despite recent advances in the theoretical foundation of real-time computing, leveraging these results efficiently in modern CPS platforms often requires domain expertise, and presents non-trivial challenges to many developers.

To understand the practical challenges in building real-time software, we conducted a survey of 189 software issues from 7 representative CPS open-source projects. Through this exercise, we found that most bugs are due to misalignment in time between cyber and physical states. This inspires us to abstract three key temporal properties: freshness, consistency, and stability. Using a newly developed concept, Data-flow Availability (DFA), which aims to capture temporal/availability expectation of data flow, we show how these essential properties can be represented as timing constraints on data flows. To realize the timing assurance from DFA, we designed and implemented Kairos, which automatically detects and mitigates timing constraint violations. To detect violations, Kairos translates the policy definition from the API-based annotations into run-time program instrumentation. To mitigate the violations, it provides an infrastructure to bridge semantic gaps between schedulers at different abstraction layers to allow for coordinated efforts. End-to-end evaluation on three real-world CPS platforms shows that Kairos improves timing predictability and safety while introducing a minimal 2.8% run-time overhead.

1 Introduction

Recent advances in artificial intelligence and robotics have promoted the integration of various autonomous cyber-physical systems into society, including self-driving cars [94], drones [31], and home-service robots [32]. Unlike conventional systems, CPS has to sense the physical world, compute for the appropriate control actions, and actuate on the physical world in a timely manner. Therefore, the assurance of temporal properties in autonomous CPS is fundamental to the correctness of the system.

System Challenges in Real-time Cyber-physical Systems. Recognizing its importance, the real-time systems community has devoted significant effort to ensuring the timeliness of computation. However, despite the rich literature on the theoretical foundation of real-time computing, such as schedulability analysis [70], mixed-criticality scheduling [43], and compositional scheduling [51, 83], leveraging these results in the development of CPS software remains quite challenging for non-experts. Furthermore, recent advances in multi-core execution and multi-modal sensing also make the problem challenging even for experts, with plenty of open research questions that are actively being investigated [66, 70]. A recent industrial survey [29] (Question 23) also indicates that only a small fraction (9.38%) of systems are designed with commercial schedulability analysis tools.

Understanding Timing Problems in Real-world CPS. To gain a better understanding of system challenges in CPS, we draw inspiration from the recent survey on concurrency bugs [64, 67], and conducted a systematic study of the 189 timing bugs in 7 mainstream open-source CPS software projects. Our goal is to understand the categories of timing bugs in CPS applications, the root causes of each bug category, as well as the challenges developers face in preventing them. We found that most of the timing bugs are caused by misalignment in time between cyber states and physical states. Therefore, building on top of the cyber-physical control loop abstraction, we extracted three most essential temporal properties: freshness, consistency, and stability. Furthermore, we found that many existing mitigations implemented manual checks for data timestamps, inspiring us to model the problem from a data-flow perspective.

Our solution - Data-flow Availability. Motivated by the findings from the timing bug study, we propose Data-flow Availability, a new concept that achieves timing assurance in autonomous systems. Building on the observation that data flow drives cyber-physical control loops in modern CPS, we

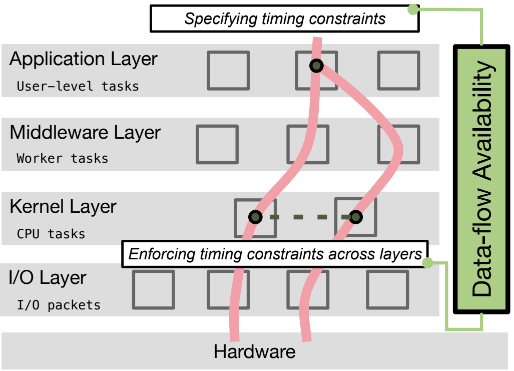


Figure 1: Data-flow Availability in the system stack.

augment data flow with a new temporal dimension, resulting in *Timed Data-flow Graph* (TDFG). Conceptually, each variable (that captures cyber or physical state) would have a time attribute (tag), and information flow among them has to respect the expectation of the software. Therefore, temporal policy is encoded as timing constraints on the edges of the graph.

To realize the concept of data-flow availability in system, we design and develop Kairos, a programming model to enable the automatic detection and mitigation of timing constraint violations. Kairos consists of a DFA-embedding tool for detection of timing constraint violation at run-time and a cross-layer scheduling system for mitigation (as shown in Figure 1). Using the DFA-embedding tool, developers can either manually annotate the source code with APIs provided by Kairos or use the provided dynamic profiler to specify the expected temporal properties. A compiler extension then takes the temporal expectation, expressed as data-flow constraints, and automatically instruments the software to detect timing constraint violations at run-time. However, detection alone does not provide timing assurance. Upon timing constraint violations, actions have to be taken to recover the system. To do so, Kairos builds on the concept of schedulable entity path to construct an association of schedulable entities in different abstraction layers of the operating system for a cyber-physical data flow. This bridges the semantic gap between the abstraction layers, and allows for more effective coordination of schedulers in the system for violation mitigation.

Prototype and Evaluation. To understand the effectiveness of DFA in mitigating the timing bugs, we analytically studied how existing bug fixes can be implemented using Kairos, and found that among the 189 bugs, 111 of them can be mitigated by Kairos. To understand the performance characteristics of Kairos, we built a prototype of Kairos, and evaluated it on three real-world robotic platforms: Autoware [36], Jackal UGV [59], and Turtlebot3 [89], each with distinct workloads and computing power requirements. On these three platforms, we show how TDFG can be constructed and used in Kairos to mitigate the existing timing issues. At runtime, Kairos introduces an average overhead of 2.8% and shows manageable performance under scalability analysis. Under high system overload, Kairos shows a faster and more stable response time in reacting to timing violations compared to other state-of-the-art systems – ROS [80], ERDOS [55], and ghOSt [57]. Furthermore, the end-to-end evaluation shows that Kairos can improve safety under high system overload.

Contributions. We make the following contributions [1]:

- Formulation of Data-flow Availability, a new concept for achieving timing assurance from a data-flow perspective.
- Design and implementation of Kairos, a proof-of-concept realization of DFA. Kairos detects timing violations by embedding a temporal property monitor within the application and mitigates these violations through a cross-layer scheduling infrastructure.
- Evaluation of DFA and Kairos across three real-world robotic platforms, each with distinct workloads and operational domains.

2 Background

Real-time Cyber-physical Systems. A unique characteristic of autonomous cyber-physical systems is their tight connection to physical world processes. Cyber-physical systems software often builds on top of the abstraction of a cyber-physical control loop, which continuously senses the physical world, calculates the appropriate control actions, and then actuates on the system to reach the desired state. The implementation of this control loop is often realized using multiple tasks (processes), where each is modeled as either periodic or sporadic tasks in the real-time models.

Timeliness Abstraction in Cyber-physical Systems. Due to their cyber-physical nature, the correctness of autonomous systems depends on both functional correctness and temporal correctness. To achieve this, real-time schedulability analysis [77] is conducted on each system based on the real-time task models. Meeting deadlines is often considered the most important requirement in real-time systems. Using the task parameters from the schedulability analysis, the scheduler of the system enforces temporal isolation among the tasks, ensuring no task misses its deadline. Based on the ability to tolerate deadline misses, systems can be hard, firm, or soft real-time. Due to various practical challenges, such as difficulty in determining a real-time task model, the efficiency of the processor to achieve system guarantees, and accurate estimation of worst case execution time, many deployed real-time systems are soft real-time systems, according to a recent

[1] The source code, as well as the extended version of this work with additional analysis and experiments is available at https://dataflow-availability.github.io/.

industry survey [29]. Furthermore, timing constraints can manifest in properties other than deadline misses, including but not limited to task response time, execution time, release jitter, and response jitter.

Timeliness Implementation in Cyber-physical Systems. Modern autonomous systems generally involve multiple abstraction layers, as shown in Figure 1. Besides the typical userspace and kernel-space layers, modern CPS software also utilizes middleware, such as the robotic operating system (ROS) [68, 80] to ease programming. Some CPS software even implements their own userspace scheduler within the application, resulting in time management across multiple layers of abstractions. This presents unique challenges for developers in achieving alignment of cyber events with physical world events. Furthermore, there is often a combination of time-driven [33] or event-driven [80] tasks.

3 Real-world Timing Bug Study

Motivation. Real-time theory suggests modeling individual computations as individual tasks. However, developing a real-time task model for modern complex data-driven CPS can be quite challenging for non-experts. Further, the formulation of highly efficient task models often requires deep domain expertise in real-time scheduling. A recent industrial survey [29] (Question 23) also indicates that only a small fraction (9.38%) of systems are designed with commercial schedulability analysis tools. Inspired by existing studies on concurrency bugs [64, 67] that had offered key insights to the community, we conducted a systematic study of timing bugs in 7 open-source robotic software. The goal is to gain a better understanding of the underlying practical challenge faced by developers. As such, the focus of the study is on *timing bugs*, where the bug is caused by non-deterministic timing of data flow within the cyber-physical system.

Methodology. The seven selected open-source GitHub robotic software projects are Autoware [53], MoveIt [2], Google Cartographer [56], Baidu Apollo [37], ORB-SLAM2/3 [3, 4], ROS Navigation [5], and ROS2 rcl [7]. These projects were selected because they represent important subsystems in modern cyber-physical control loops, including perception, localization, planning, and control. Furthermore, they have also been widely adopted [1, 35, 52, 75]. To collect the bugs, a set of keywords (e.g., 'timing', 'sched', 'timestamp', 'temporal', etc.) was used to filter the issues, resulting in a list of 189 bugs.

Summary of Systemization. As shown in Table 1, we find that two categories of root causes account for the majority (169 out of 189) of the collected timing bugs: insufficient specification and enforcement of timing constraints. The rest are design flaws and hardware problems.

Table 1: Timing Bugs in Real-world Applications

Projects	# bugs	Timing Constraint Specification			Timing Constraint Enforcement		Others
		Missing Constraint	False Specification		Missing Constraint	False Enforcement	
			Expressibility	Parameter			
Cartographer [47]	34	14	12	1	1	4	2
Apollo [37]	49	11	23	2	3	0	10
MoveIt [2]	23	4	7	2	2	5	3
ORB-SLAM [4]	6	1	1	0	1	2	1
Autoware [53]	16	6	5	0	1	0	4
Navigation [5]	15	3	3	1	2	4	2
ROS rcl [7]	46	3	3	1	3	35	1
Total	189	42	54	7	13	50	23
Scope of This Work		✓	✓	✓	✓		

3.1 Timing Specification Bugs

The most common cause of timing assurance failure is incorrect specification of timing constraints (103/189 bugs). As discussed earlier, though real-time theory provides a sound foundation for assuring timing behavior, there remains a gap in transitioning the theory into practice for developers without expertise in real-time computing. Without the formal guarantees provided by real-time theory tools (such as schedulability analysis), current practice adopted by developers to mitigate this involves developers tagging data with timestamps when data is created or transferred, and then using these timestamps to check the data's validity (e.g., freshness) when it is used. This data-centric approach to timestamp checking for specifying timing constraints is ubiquitous in the codebases we investigated. For example, Autoware and Google Cartographer use timestamp checks in over 340 and 110 places, respectively, to determine the execution logic. Additionally, state-of-the-art middleware such as ROS [80], ROS2 [68], and ERDOS [55] also incorporate built-in timestamps on data transferred between tasks.

3.1.1 Missing Time Constraints (What to Check)

Figuring out where and how to add the timestamp checks manually is quite challenging due to the complex dependency among data from different tasks [8, 12, 14, 17, 19–21, 23, 50, 54]. Naively, one can simply add timing checks on all instructions. However, that will introduce prohibitive overhead to the system, leading to adverse physical outcomes.

Implication - It is essential to understand not only which program statements need to be checked but also which aspects of temporal properties should be verified, in order to minimize the performance impact of the protection.

To further dive into the root cause of the problem in a principled approach, we went back to the basic abstraction of a cyber-physical control loop to ask the question of what properties are these timing bugs violating. Through the lens of physical world impact, three key properties arise during the analysis of the timing violations.

Freshness - *describes the latency between the occurrence of a physical phenomenon and the consumption of its cyber rep-*

resentation. While data should be as fresh as possible, there will always be some delay due to sensing and computation.

The key is to ensure that the freshness of the particular data is acceptable by the control implementation. Figure 2 shows an example from Cartographer-Pull-153. Cartographer [47] uses a queue to manage and process sensor data streams from multiple sources in a coordinated, time-ordered manner. It then uses the data to construct a robot's trajectory for localization. The code snippet checks if the incoming data is older than the start time of the current trajectory and discards the outdated data if it is.

```
void OrderedMultiQueue::Dispatch() {
  // We take a peek at the time after next data. If it
  // is not beyond 'common_start_time' we drop it
  std::unique_ptr<Data> next_data = next_q->queue.Pop();
  if (next_q->queue.Peek()->time > common_start_time) {
    last_dispatched_time_ = next_data->time;
    next->callback(std::move(next_data));
  }
  // else: drop the data
```

Figure 2: Freshness check where outdated data is dropped. Simplified code snippet from Cartographer-Pull-153.

Consistency - *describes the temporal alignment of the physical world observations in the data flows converging at a specific statement of the program. Ideally, the physical events captured by these cyber states should be as synchronized as possible.*

Figure 3 shows an example from ROS Navigation [5] (Navigation-Pull-1121), where the control task retrieves the robot pose using the tf_ buffer, which maintains historical poses. In the original code (highlighted in red in line 1), it directly uses the latest pose. However, since the tf_ buffer is dynamically updated by other tasks, the timestamp of the current map used by the control task (time) might be older than the latest pose in tf_. This could result in using a pose that is ahead of the current map in time, causing motion planning to produce incorrect paths. To fix this, the code highlighted in green (in line 3) adopts a timestamp-based check that compares the timestamp of tf_ with the control task's timestamp. If time is not newer than the latest in tf_, the lookupTransform() function is called to interpolate the pose that temporally aligns with the current map.

```
 1 tf_.transform(robot_pose, global_pose, global_frame);
 2  // check if curr_time is less than latest update time of tf_
 3 if (tf_.canTransform(global_frame, robot_base_frame, curr_time)) {
 4   // if so, transfomr at the time point of curr_time
 5   transform = tf_.lookupTransform(global_frame,
 6                                   robot_base_frame,
 7                                   current_time);
 8   tf2::doTransform(robot_pose, global_pose, transform);
 9 } else {
10   // use the latest otherwise
11   tf_.transform(robot_pose, global_pose, global_frame_);
12 }
```

Figure 3: Consistency check detecting temporal alignment between the data from two tasks (Navigation-Pull-1121).

Stability - *describes the variation in freshness. This is similar to the concept of jitter in the real-time and control domains, and ideally jitter should be minimized.*

Many control algorithms and systems are designed to have an implicit assumption of not only the boundary of the freshness but also its variation (often relatively small) from loop to loop. In essence, it is about the consistency of data flow in the temporal dimension, as compared to the spatial dimension (consistency as discussed above). Figure 4 shows a code snippet from AutowareAuto-Pull-980, where a timer is added to ensure the stability of control output. The timer checks the elapsed time in a polling loop to trigger the control output function at expected intervals.

```
NERaptorInterface::NERaptorInterface(. . .){
  /* Use a ROS timer to ensure the stability */
  m_timer = node.create_wall_timer(m_pub_period,
                                   std::bind(&cmdCallback, this));
 }

/* In implementation of ROS timer */
while (rclcpp::ok()) {
// Use elasped time to check if timer is ready via a polling loop
  rcl_timer_get_time_until_next_call(m_timer, &time_until_next_call);
  if (time_until_next_call <= 0)    m_timer->call();
}
```

Figure 4: Stability check using a ROS timer to minimize control jitters (AutowareAuto-Pull-980).

Summary - These three key properties present a unique opportunity to address a large number of bugs with a small amount of temporal property checks.

3.1.2 Inadequate Timing Constraints (How to Check)

Even after solving the challenge of what to check, developers also have to tackle the challenge of how to check. There are 61 bugs caused by inadequate specification of timing constraints; among these, we found two common causes. The first category is that some of the hard-coded time constraints may not be appropriate for the deployment. This often happens due to insufficient testing or changing software/hardware [11] or operating environment [55] of the system. The second category is less straightforward. In real-time cyber-physical systems, there are other essential timing dimensions beyond latency (maps to freshness discussed earlier), such as alignment (maps to consistency discussed earlier) and jitter (maps to stability discussed earlier). For examples, issues can arise on arrival jitter [10], detection of data loss [24], processing data time ratios [13, 27], and requests of development of timing utilities [9, 16]. Figure 5 shows a simplified bug example from Google Cartographer [47] (Cartographer-Issue-242) that spans multiple patches before being finally fixed. The code snippet estimates the robot's velocity by dividing the difference in positions between two adjacent frames by the time interval. In this case, the freshness of data is a problem because if the incoming LiDAR frame is older than the latest one (i.e., out-of-order), it causes the time difference (delta_t) to be negative. The freshness check was added in patch [79] (highlighted in yellow on line 3 of the figure). However, another problem beyond data freshness persists even if delta_t is positive. The irregular timing may result in two LiDAR frames being too close in time, causing delta_t to be too short. In such cases, the position difference is divided by a very small `delta_t` value, which can significantly magnify any estimation errors, potentially causing the velocity to become infinitely large. This issue is finally fixed by inserting a check (on line 8) that the frames with intervals less than 1 ms are dropped.

```
 1 // Estimate the velocity estimate.
 2 if (time >  common::Time::min()
 3     && time > last_scan_match_time) {
 4
 5   // Prevent out-of-order data
 6   double delta_t = common::ToSeconds(time - last_scan_match_time);
 7
 8   if (delta_t < 1e-6) return;
 9   // Prevent too short intervals
10   velocity_estimate_ += (pose_estimate_.translation() -
11                          model_prediction.translation()) /
12                          delta_t;
13 }
```

Figure 5: Timestamp checking is incomplete in semantic.

Implication - Given the dynamic range of timing expectations in different platforms and physical environments, it is important to develop a mechanism that simplifies the configuration of these ranges for developers. Ideally, this mechanism should also enable the automatic discovery of the necessary ranges to maintain system safety.

3.2 Timing Enforcement Bugs

There are 62 bugs stemming from inadequate enforcement of timing constraints. Most of these (50/62) are due to conventional software bugs, such as memory corruption in the enforcement infrastructure with schedulers and timers. Another 13 of these bugs are caused by timing constraints not being delivered to the enforcement mechanisms. These are primarily caused by the fact that the specified timing constraints are limited to userspace applications and are not propagated to other scheduling layers. Figure 1 schematically illustrates the scheduling layers involved in designing and deploying autonomous systems. Due to the inadequate support for delivering scheduling contexts across schedulers, timing constraints (or scheduling decisions) specified at one scheduler fail to propagate to others. This type of problem can be observed in issues where priorities are inverted across layers [22], leading to critical tasks not being reliably triggered [26], executing at varying periods [25], or executing out of order [15]. As a result, mitigation methods to maintain relative priority at the user level or middleware alone are often quite challenging if not impossible.

```
void SchedulerChoreography::CreateProcessor() {
  proc->BindContext(ctx);
  /* Reserve a set of CPU cores for the tasks */
  SetSchedAffinity(proc->Thread(), pool_cpuset_, pool_affinity_, i);
  SetSchedPolicy(proc->Thread(), pool_processor_policy_,
                 pool_processor_prio_, proc->Tid());
}
```

Figure 6: Mitigating disconnection across scheduling layers.

Figure 6 is the mechanism adopted to handle Apollo-Issue-9433. It introduces a new scheduling strategy that reserves a set of CPU cores for middleware tasks, allowing them to be directly scheduled on these cores and avoiding disconnection between layers. However, implementing this scheduling strategy necessitates a thorough understanding of the tasks, including their dependencies and execution times.

Implication - Assurance of timing expectations is more effective when scheduling contexts are visible across all layers of abstraction.

3.3 Summary

Table 1 shows the timing bugs we studied, and the scope of the proposed mechanism DFA. Based on the study, we summarize the opportunities and insights that inform the design of DFA:

- Timing expectations are often added by programmers by checking the age of data, hinting at the potential to use information(data)-flow as a mechanism to capture the programmer's intention.
- There are three types of key temporal properties we systematized based on the cyber-physical control loop abstraction, freshness, consistency, and stability.
- Timing enforcement would often benefit from visibility across different layers of abstractions in the OS.

4 Data-flow Availability

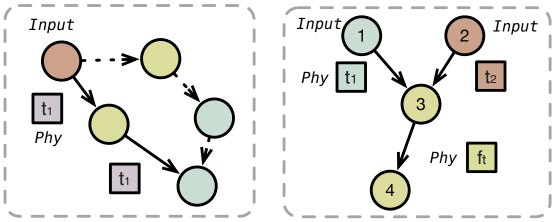
Motivated by the challenges in Section 3, this paper introduces ***Data-Flow Availability*** (DFA), which approaches the policy definition of temporal property from a data-flow perspective.

4.1 Timed Data-flow Graph

A *Timed Data-flow Graph* (TDFG) is a representation of a DFA-enabled program, extended from the program's data-flow graph.

Graph Definition. TDFG is a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{T}, \mathbf{C})$ constructed from program's intermediate representation:

- *Vertex:* Each vertex v in the set $\mathbf{V}$ corresponds to a statement in the intermediate representation.
- *Edge:* Edges $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ represent data dependencies between vertices. An edge is added if the corresponding statements have a data dependency.
- *Timing Tag:* A timestamp $t_{phy} \subseteq \mathbf{T}$ is generated with `def` of memory SSA (Static Single Assignment) in the graph and propagated along edges at runtime. It includes two types of timing, either the physical world sensor reading or the range of a derived value from the timed sensor values.



(a) Sequential propagation. (b) Merge from different inputs.

Figure 7: Two cases of timing propagation in data flows.

```
void EvaluatorManager::DumpCurrentFrameEnv() {
  FrameEnv curr_frame;
  auto obstacles = ContainerManager()->Get(PERCEPTION_OBSTACLES);
  curr_frame.set_timestamp(obstacles->timestamp());
}
```

(a) The timestamp propagates on a single data flow (Apollo-Pull-8503).

```
bool Fusion::GenerateMsg(Obstacles* obstacles) {
  common::Header * header;
  header->set_lidar_stamp(lidar_timestamp * 1e9);
  header->set_camera_stamp(camera_timestamp * 1e9);
  header->set_radar_stamp(radar_timestamp * 1e9);
 /* Processing */
```

(b) The timestamps of multiple data flows merge (Apollo-Pull-5459).

Figure 8: Code examples for timing propagation in data flows.

- *Timing Constraints:* The edges can be assigned timing constraints $C \in \mathbf{C}$. These constraints define the temporal properties that the edge is expected to meet within specified tolerance thresholds. The constraints are evaluated upon information flow. These temporal properties are defined by DFA's metrics, which are detailed in Section 4.2. Note that some of the temporal properties require analyzing the statistics of data flows into a vertex over time/iterations.

Timing Information Propagation. Timing tags can be propagated along edges E at runtime. There are two forms of timing tag propagation patterns (shown in Figure 7) that are common in a cyber-physical system:

- *Propagation of Timing Tag in Single Data Flow.* The timing information is propagated along a single data flow (Figure 7(a)), where edges inherit the timing tag from the predecessor edge, unless the data flow comes from a new sensor reading. Since the timing tag represents the time when the physical world observation is made, the data flow within cyber space does not change the tag. This is the most common case. In practice, developers programmatically add the timestamps to the variables according to data received from the predecessor tasks. Figure 8(a) shows an example from the Baidu Apollo self-driving car project (Apollo-Pull-8503). The prediction task inherits the timestamp of obstacles from the object detection task. The timestamp is then used to calculate the data age of the currently perceived environment upon which the prediction is based.

- *Merging Timing Tag from Multiple Data Flows.* This category (Figure 7(b)) involves merging multiple data flows at a vertex. This is typically required for tasks that fuse information from different sensors. In this case, the resulting memory SSA from the statement inherits the timestamps from its incoming edges, and maintains $t_{phy} = f_v(t^1_{phy}, ..., t^n_{phy})$, where f_v is the merging function for the vertex v. While the figure shows only two data flows, there could be more than two. Note that there is a one-size-fits-all solution in how timing tags can be merged, since it is effectively merging observations on the physical world from different time instances. One common approach is to keep the range of the time tags. The code snippet in Figure 8(b) depicts a fusion task in Apollo, added in Apollo-Pull-5459. Since this task fuses detection results from LiDAR, cameras, and radar, it also incorporates their timestamps to check the temporal alignment later.

4.2 Timing Constraints in TDFG

Based on the timing bug study, three essential temporal properties were formulated based on the cyber-physical control loop abstraction: freshness, consistency, and stability. In the following, we will show how they can be captured using TDFG in the form of *Timing Correctness*.

Freshness focuses on the difference between the time when a physical observation is made and the time when this observation is used by the control system. In cyber-physical systems, this difference often has to be bounded, as any latency increases the temporal gap between the cyber and the physical world, as previously discussed in Section 3. As a result, given an edge e with a maximum tolerable timing threshold θ_f, its freshness is calculated by:

$$C_f = \theta_f - (t_{-} - t_{\text{phy}})) \tag{1}$$

where t_{-} is the current time.

Consistency concerns the time differences between the timing tags from different data flows into a vertex, which intuitively indicates the differences in the physical world status at different times. Generally, the smaller it is, the closer the time stamps are, and the more consistent the physical world observations should be. Consider n edges that have the same egress vertex: $T_{\text{def}} = \langle t^1_{\text{def}}, \ldots, t^n_{\text{def}} \rangle$. Their temporal consistency can be checked by:

$$C_c = \theta_c - \max_{i,j \le n}(t^i_{\text{phy}} - t^j_{\text{phy}}) \tag{2}$$

where θ_c is the tolerable threshold (or range).

Stability captures differences in timing characteristics of data flows into/out of a vertex temporally. Many tasks in real-time systems are implemented as periodic workloads and thus some underlying algorithms/models are designed with the assumption of periodicity, which necessitates periodicity in data usage, such as input (e.g., sensor input) or output data (e.g., actuation command) [71]. For w edges belonging to a

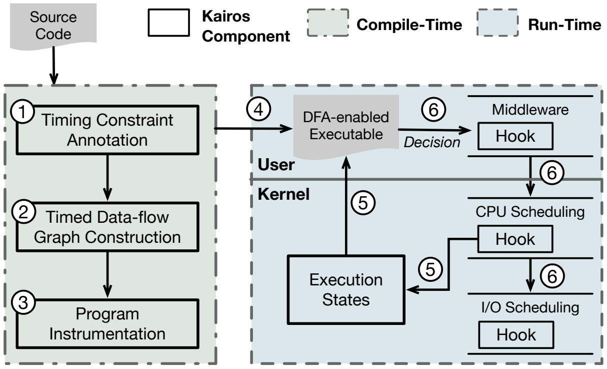


Figure 9: Workflow of Kairos.

set of data flows from sequential loops via the same program point, they have $T = \langle t^1_{\text{phy}}, \ldots, t^w_{\text{phy}} \rangle$. One typical way to check stability is by measuring jitters:

$$C_s = \theta_s - \max_{i,j \le w-1} |D_i - D_j|, D_i = \Delta_i - I, \Delta_i = t^{i+1}_{\text{phy}} - t^i_{\text{phy}}. \quad (3)$$

where Δ represents the interval between two iterations and I is the expected interval. In practice, the form of stability can vary based on the design of the target systems, with alternatives potentially being variations of freshness.

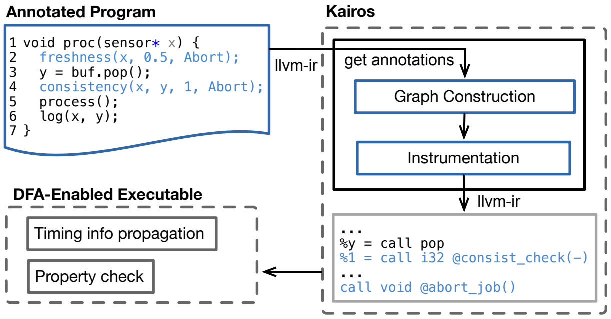
An edge e is evaluated upon the program's execution reaching its egress vertex, and it is considered compliant with timing correctness if all its added metrics meet $C > 0$, namely $(C_f > 0 \wedge C_c > 0 \wedge C_s > 0)$.

5 Design and Implementation of Kairos

Kairos is a proof-of-concept realization of Data-flow Availability. There are two main components, the temporal policy definition using TDFG and the mitigation of policy violation. Kairos is composed of a compiler extension and a run-time system. Figure 9 outlines its key components and workflow. At compile-time, Kairos leverages program analysis and user annotation/automatic annotation ① from profiling to construct the TDFG of the target application ②; Utilizing the TDFG, it instruments code to perform timing information propagation ③; At run-time, tasks update timing information and evaluate timing correctness ④; Upon timing constraint violation, the task triggers a handler to execute the pre-defined policy ⑤; The scheduling decisions from handling policy are then shared with schedulers across different layers ⑥.

5.1 DFA-enabled Application

As shown in Figure 10, there are two main steps in the construction of DFA-enabled application, the construction of TDFG, which defines the temporal properties the application has to follow and the instrumentation of the application to enable detection and mitigation of the property violation.



```
void proc(sensor* x) {
  freshness(x, 0.5, Abort);
  y = buf.pop();
  consistency(x, y, 1, Abort);
  process();
  log(x, y);
}
```

```
...
%y = call pop
%1 = call i32 @consist_check(-)
...
call void @abort_job()
```

Figure 10: Pipeline of DFA-enabled application construction.

TDFG Construction. TDFG captures the expected temporal properties of the developers. Upon extraction of value-flow graph [49, 86], the timing constraints in TDFG are expressed either manually via developer annotations or automatically via dynamic profiling.

Table 2: Kairos API

Function Name	Arguments				Description
	Targets	Tolerance	Window	Handling Policy	
freshness	var	threshold	-	abort	Checks the expected properties. If violated, triggers the handling policy function.
consistency	var, ...		-	prioritize	
stability	var		size	skip-next	

To facilitate manual annotation, three APIs are provided for annotating the source code to express timing constraints over the three key properties (freshness, consistency, and stability) that were previously discussed in Section 3. As shown in Table 2, the functions take four types of arguments: target variables, tolerance threshold, window size (for stability only), and handling policy. The threshold and window size parameters enable the check of timing correctness. The handling policy argument specifies the function to be invoked if timing correctness checks fail.

However, manual annotation often requires strong domain knowledge not only of the physical system but also of the computing stack, which may not always be available. To tackle this, Kairos also provides an option to extract the timing constraint using performance profiles from dynamic analysis. To do so, Kairos needs two key components: first, an oracle (criteria) to determine if the timing behavior of the software needs to be corrected or not; second, inputs to instrument the system such that all potential behaviors can be observed. For the oracle, Kairos borrows existing practice in CPS evaluation, where safety (often measured as control state deviation) is used as the metric. When physical safety (such as vehicles crashing into pedestrian or drones falling from the sky) is compromised by the violation of a specific temporal property, Kairos considers this temporal property to be essential and has to be monitored and checked at runtime. Inputs to the system, i.e., the physical scenarios, to test the system is an open challenge in CPS testing [72]. In Kairos, in addition to relying on the user to supply scenarios that might reveal temporal property violations, we use the performance interference tool [65] to probe the system with different potential timing impacts. To minimize the impact on the timing behavior of the software due to the profiling system, hardware performance monitors and debug functions are used. Among all flows that cause the same property violation, Kairos only adds the check on the first occurrence. It is important to note that dynamic profiling is much more effective in finding the acceptable range of the constraints rather than finding where to add the constraint (which has a much larger search space).

TDFG Embedding. Before constructing the TDFG, Kairos analyzes the source code to identify statements that receive sensor inputs, and automatically instrument them to extract timestamps t_{phy} from the sensing or input payloads. To begin the construction of TDFG, Kairos builds on top of the value-flow analysis in the SVF [86] tool with LLVM-IR [63], then uses a set of python scripts to add the timing constraints and annotation. Additionally, a set of LLVM compiler passes is also developed to instrument the necessary code for timing information propagation and checking. Kairos also leverages several heuristics to reduce the performance overhead. First, to avoid instrumenting every instruction for timing metadata propagation, Kairos automatically bypasses the timed data flows with the same time tag t_{phy} (sensor timestamps). Without loss of generality, a vertex is selected to be in the TDFG based on three criteria: *(i)* it is either a physical input or physical output vertex, *(ii)* it merges multiple data flows, or *(iii)* it is annotated with timing constraints as a vertex of interest.

5.2 Timing Constraint Violation Mitigation

Timing Constraint Violation Handler Policies. Mitigation of timing constraint violations often requires consideration of the physical components, and there is no one-size-fits-all solution. Drawing inspiration from our bug study, prior works in real-time computing [39, 46, 69, 91] and current industrial practices [29], Kairos offers three policies: *abort*, *prioritize*, and *skip-next*. More specifically, *abort* discards the task instances with timing constraint violations. *prioritize* switches the system into a different set of task models, often involving raising the priority of the task. Lastly, *skip-next* allows the delayed task to continue but skips its next instance to recover.

It is important to note that individually, these policies may give rise to further timing constraint violations in a cascading effect. For example, prioritizing a task that has missed its deadline might prevent other tasks from making progress, resulting in subsequent deadline misses. However, if correctly composed, these policies support existing adaptive real-time scheduling paradigms, e.g., elastic scheduling [46] and mixed-criticality scheduling [90].

Under *elastic scheduling*, task utilizations are decreased (typically by increasing the periods at which they are invoked) to avoid deadline misses. Though originally proposed in [46] as a mechanism to adapt to system overload, elastic scheduling has since evolved as a means by which systems can adapt to unexpectedly long task execution times [45] or interference from other tasks [85]. In response to a violation of timing constraints, Kairos can use the algorithm from [84] to quickly recompute task periods, then enforce this with multiple *prioritize* policies to change task priorities or `SCHED_DEADLINE` attributes accordingly.

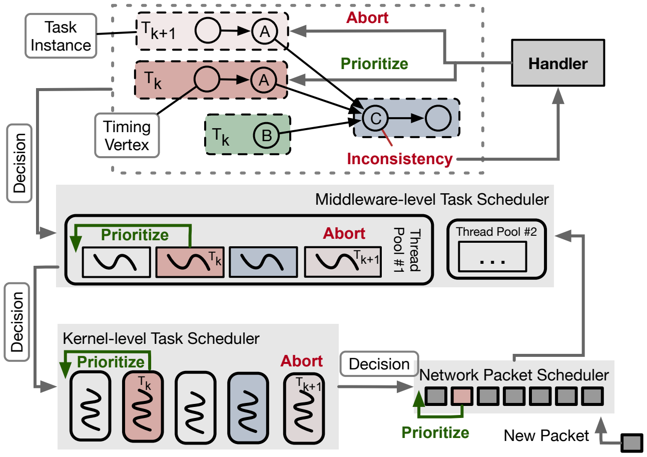


Figure 11: An illustrative case of Path across layers.

In *mixed criticality systems*, non-critical task instances may be dropped in response to timing anomalies in critical tasks. Earliest-deadline first (EDF) scheduling with virtual deadlines (EDF-VD) is an optimal scheduling algorithm for non-clairvoyant mixed-criticality systems (i.e., those for which timing anomalies can't be predicted a priori, but are only identified when they occur) [39]. Under EDF-VD, each critical task is prioritized according to its virtual deadline, which is assigned as a constant parameter. When a critical task overruns its expected execution time, instances of non-critical tasks are dropped to maintain guarantees to critical tasks, and critical tasks are re-prioritized according to their absolute deadlines [39]. Kairos's handler supports this mode switch via a combination of its *abort* and *prioritize* policies (applied to the non-critical and critical tasks, respectively). While developing more sophisticated policies presents intriguing research opportunities, it is left for future exploration.

Implementation. Handlers can be implemented in individual layers or across multiple layers. In our prototype, we implement it as a modification to the kernel schedule where the mitigation mechanism is invoked before the built-in scheduler for proof-of-concept. For task abort, our prototype instruments code to enable early return. However, it is important to note that resource deallocation and inconsistent state removal often require sophisticated management [82]. The skip-next is demonstrated in the middleware by dropping the next task invocation message at ROS.

Cross Layer Scheduling Association. While the handling policies for timing constraint violation are relatively well understood under the real-time task problem formulation at the task level. Existing software ecosystems come with schedulable entities at different layers of architectural abstractions from I/O layer (such as network packet) and operating system (real-time process) to middleware (ROS component) to application (application-specific schedulers). This presents non-trivial system challenges in realizing consistency in the handler policy due to missing semantics across the abstraction layers.

This problem can be observed in 13 bugs [18, 38] from our earlier bug study. For example, the handling of a consistency violation often needs to adjust threads on sensor processing rather than the fusion process, necessitating the correlation of a subgraph of TDFG to the corresponding schedulable entities such that the timing constraint violation handler knows which one to intervene on [41]. Another example that commonly occurs in time-sensitive networking is the need to prioritize specific items in the network queue due to reprioritization of tasks [95], which can be part of the handling process.

To mitigate this, we propose to bridge the semantic gap by associating schedulable entities to data flows in TDFG. This not only allows the handler to know the schedulable entity to operate on, but also allows the other abstraction layers to respond to a handling mechanism much more effectively. To ensure the association is complete, Kairos draws inspiration from the Path concept from Scout system [73], where Path is used to track the components a packet travels through (e.g., network devices or protocol layers) on network appliance systems. In Kairos, upon dispatching, Path is updated to reflect the chain of schedulable entities that leads to execution of the application along a particular path, as shown in Figure 11.

Implementation. Our prototype modifies the data structure of native scheduling entities to store the Path to which they belong. This information is updated in a shared buffer accessible to four layers, i.e. user space, kernel, middleware, and network stack. The method of incrementing Path varies: in the kernel and network stack, it occurs where new tasks or packets are created; while in ROS middleware, it happens as threads are dispatched to execute callback functions.

6 Evaluation

This section seeks to answer the following questions: *(i)* What is the capability of DFA in addressing real-world timing bugs? – Section 6.1; *(ii)* What is the cost and efficacy of Kairos? – Section 6.2; *(iii)* How do DFA and Kairos improve performance/safety in abnormal timing situations? – Section 6.3.

Experimental Setup. The evaluations were performed on synthetic workloads and the workloads of three real-world autonomous systems: (1) Autoware.Auto [36] – an open-source full-stack autonomous driving project, which presents a high-end real-time autonomous system. (2) Jackal UGV – an unguided ground vehicle that represents mid-end autonomous system. It uses Google cartographer [47] for vehicle localization and ROS navigation [6] for path planning and control. (3) Turtlebot3 – a low-end indoor robot that relies on ROS navigation [6] for localization, planning, and control. Given that the software stacks of each system require distinct computing power, we used three different computing units that align with (or are similar to) the official recommendations to better emulate abnormal timing situations. The experimental hardware settings are listed in Table 3. Autoware and Jackal UGV were evaluated using hardware-in-the-loop simulations, while Turtlebot3 was also evaluated using a real robot.

Table 3: Evaluation Platforms

Platforms	Software Stack	Computing	Cores	RAM	Kernel
Autoware	Autoware.Auto [36, 61]	AMD 9 3900X RTX 3070 Ti	12	128GB	Linux 5.11
Jackal	Cartographer [47] & Navigation [5]	Intel Nuc 8	4	16GB	Linux 5.11
Turtlebot3	Navigation [5]	RPi 4B	4	4GB	RPi 5.15
Microbenchmark	ORB-SLAM3 [4]	Intel i9-12900K	12	128GB	Linux 5.11

Table 4: Root Cause Analysis of Bug Fix Capability

Category		**Description**	**Number**
Fixable	Non refactoring	Inadequate timing information/constraints/propagation	104
	Refactoring	Remove built-in conflict logics	5
		Adapt with software semantic	7
Unfixable	Out-of-Scope	Hardware-related timing faults	6
		Algorithm-related timing bugs	8
		Infrastructure bugs (e.g., scheduler crash)	41
	Limited	Concurrency bugs	12
		Performance issue	6

6.1 DFA in Solving Real-world Bugs

A key question to answer in evaluating the efficacy of Kairos is its ability to address the timing problems. To do so, our evaluation leverages the collection from the bug study and analyzes if Kairos can detect the timing problems (through temporal policy defined in TDFG) and mitigate the timing problems (using the cross-layer temporal policy violation handler). Due to the need to use physical system or emulation to exercise the system, most of the results from this evalua-

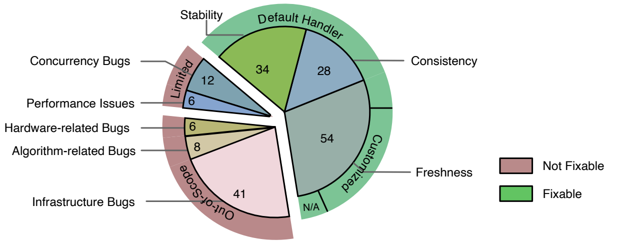
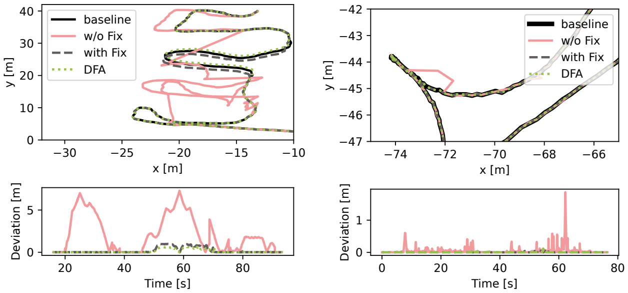


Figure 12: Statistics on bug fixability and root causes.

tion item are acquired through manual inspection by three cyber-physical system developers with 6, 10, and 17 years of experience respectively. A bug is considered fixable if all three developers agree that the mitigation can be expressed using primitives in Kairos. Furthermore, we've also conducted two case studies to demonstrate that Kairos can be used to detect and mitigate violations of the key temporal properties.

The results of the manual inspection are shown in Figure 12. From our earlier case study, there are 189 timing bugs. 116 of them can be detected by DFA, and 23 are not detectable, because they are caused by underlying infrastructure bugs, design flaws, or hardware issues (beyond the scope of Kairos). 76 bugs can be mitigated directly using default mitigation handling policies, while 35 bugs can only be mitigated using customized temporal violation handlers. 5 bugs cannot be mitigated even with customized handlers, because they require adaptation in the underlying design model or algorithms, necessitating a complete software redesign. Table 4 summarizes the reasons on all the bugs that Kairos cannot address. To further understand how Kairos can be used to address real-world bugs, we reproduce two bugs violating stability and consistency respectively, since freshness is often easier to handle.

Case-1: Abnormal timing of LiDAR Pointcloud in Cartographer. This case study demonstrates the effectiveness of Kairos in identifying and mitigating violations of stability timing constraint. Specifically, we evaluate Kairos on issue Cartographer-Issue-242 (code snippet shown in Figure 5) in Cartographer [56], a widely used localization package. According to the original issue report, the LiDAR pointcloud data, which is assumed to arrive at periodic intervals, sometimes arrives more closely than expected, violating the system's stability timing constraints. To reproduce the same impact of the issue, we modified the driver code to induce the same abnormal timing patterns, specifically manipulating the time between two pointclouds to be below 1 ms. Such timing patterns cause the vehicle to deviate from the baseline at most 10.3 m, as shown in Figure 13(a). To solve this issue, the patch in the codebase checks the timestamp of each point and removes abnormal ones with intervals of less than 1 ms. With this removal, the produced localization results are comparable to the baseline (deviation at 0.20 m). With Kairos, we specify stability timing constraint on *ScanMatch()* statement which consumes variable *msg* in task *HandleLaserScanMessage* that receives the point cloud with the annotation API *stability()*. The tolerance threshold argument in the API, which is set between 22 and 33 ms, is obtained through dynamic profiling of the ranges of intervals that do not incur adverse control outcomes. This process takes 18.3 minutes. We use *abort* as the default policy to mitigate the violation. The produced localization result aligns with the baseline at 0.19 m, which is comparable to the official patch as shown in Figure 13(a).

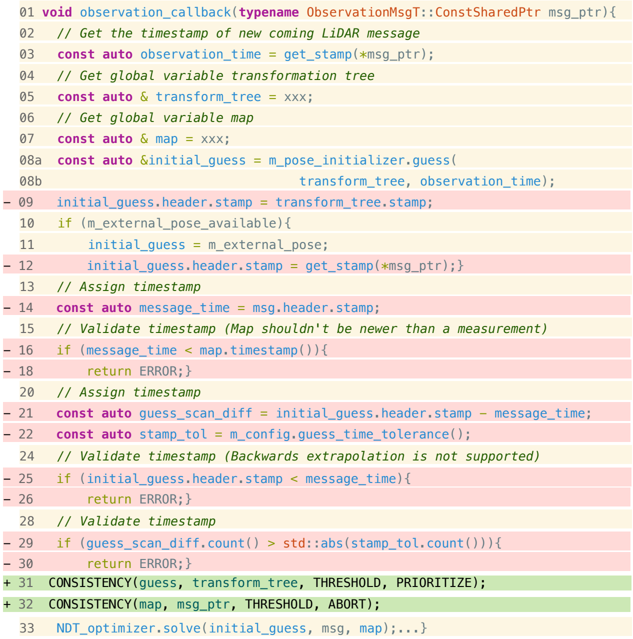


(a) Jackal case study.   (b) Autoware case study

Figure 13: Case studies on fixing timing bugs using DFA.

Case-2: Latency in Updating Location. This case study showcases the effectiveness of Kairos in detecting and mitigating violation of consistency timing constraints. Specifically, we evaluated Kairos on the issue Autoware-Issue-458 in Autoware (code snippet shown in 14). According to the original issue report, the timestamps of the generated LiDAR data and odometer data used by the optimization procedure (line 31), which aims to perform the localization, should be within a threshold but are sometimes misaligned, violating the system consistency timing constraint. Since the mechanism to trigger the bug was not discussed in the original issue submission, we inject an intermittent CPU overload at 60 % level using stress-ng [62] on the cores running localization-related tasks to introduce inconsistency between LiDAR and odometer data. Such inconsistency causes the vehicle to produce a trajectory that deviates from the ground truth by 1.88 m, as indicated in Figure 13(b). To solve this issue, the patch in the codebase tags the LiDAR and odometry data with timestamps. It then compares these timestamps. If the difference between them is more than 1 second, the results are discarded. With this removal, the produced localization results align with the baseline at 0.12 m. To solve this problem with Kairos, we use annotation API *consistency()* (line 31 in Figure 14) to specify the consistency timing constraint on *transform_tree* and *msg_ptr*. The tolerance threshold argument in the API, which is set between 1.2 s, is obtained through dynamic profiling of *the difference between timestamps of two variable generations* that do not incur adverse control outcomes. This process takes 8.5 minutes. We use *prioritize* as the default policy to mitigate the timing constraint violation. The produced localization result aligns with the baseline at 0.091 m, which is comparable to the official patch.

A Programming Example. We use case-2 (Autoware-Issue-458) as an example to showcase how Kairos reduces the effort in programming timing constraints. Figure 14 shows simplified code snippets from Autoware's localization component. In this component, incoming LiDAR pointclouds, HD maps, and the transformation tree (extrapolated pose based on past information) are used jointly, so their timestamps should



```
  void observation_callback(typename ObservationMsgT::ConstSharedPtr msg_ptr){
    // Get the timestamp of new coming LiDAR message
    const auto observation_time = get_stamp(*msg_ptr);
    // Get global variable transformation tree
    const auto & transform_tree = xxx;
    // Get global variable map
    const auto & map = xxx;
    const auto &initial_guess = m_pose_initializer.guess(
                                      transform_tree, observation_time);
-   initial_guess.header.stamp = transform_tree.stamp;
    if (m_external_pose_available){
        initial_guess = m_external_pose;
-       initial_guess.header.stamp = get_stamp(*msg_ptr);}
    // Assign timestamp
-   const auto message_time = msg.header.stamp;
    // Validate timestamp (Map shouldn't be newer than a measurement)
-   if (message_time < map.timestamp()){
-       return ERROR;}
    // Assign timestamp
-   const auto guess_scan_diff = initial_guess.header.stamp - message_time;
-   const auto stamp_tol = m_config.guess_time_tolerance();
    // Validate timestamp (Backwards extrapolation is not supported)
-   if (initial_guess.header.stamp < message_time){
-       return ERROR;}
    // Validate timestamp
-   if (guess_scan_diff.count() > std::abs(stamp_tol.count())){
-       return ERROR;}
+  CONSISTENCY(guess, transform_tree, THRESHOLD, PRIORITIZE);
+  CONSISTENCY(map, msg_ptr, THRESHOLD, ABORT);
    NDT_optimizer.solve(initial_guess, msg, map);...}
```

Figure 14: Simplified code for temporal consistency checks in Autoware. ‘-’ (red) represent built-in checks, while ‘+’ (green) are checks via Kairos’s API.

be checked as aligned. The standard checking mechanism (red lines marked by ’-’) requires developers to identify data provenance, label timestamps, and verify them before use. This often involves frequent jumps to other functions in different contexts, necessitating a deep understanding of data-flow relationships, which is both time-consuming and error-prone. In contrast, by using Kairos’s APIs, users can omit all timestamp assignments and checks and simply add two statements before using the LiDAR point cloud and map (marked by two green lines with ‘+’ in the figure).

Overall, Kairos eases programming with timing constraints in three ways. First, it removes the requirement of programmatically assigning timestamps to variables. Second, it avoids unnecessary or repeated timestamp checks. Third, it does not require developers to thoroughly understand the temporal relationships between different data flows in the source code.

6.2 Cost and Efficacy of Kairos

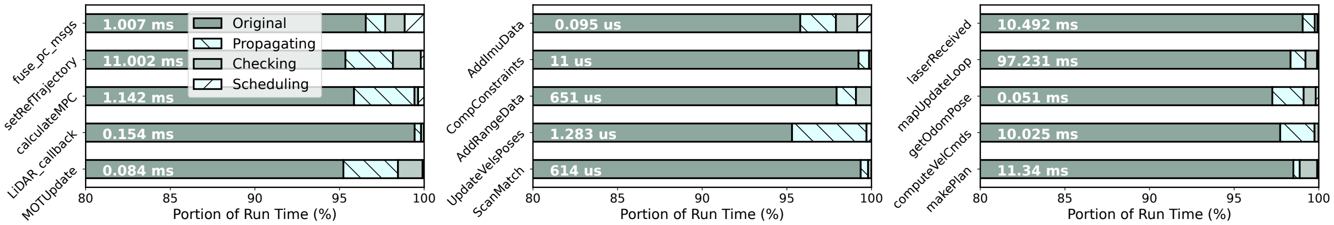
Runtime Overhead on Real-world Applications. The runtime overhead of Kairos stems from three aspects: timing information propagation, timing correctness checking, and the added logic in schedulers. We separately measured the overhead for each aspect on five representative tasks (or functions) per platform, averaging execution times over 100 runs. The results, shown in Figure 15, include original times and proportional increases.

The largest overheads observed in these tasks are *MOTUpdate* in Autoware (4.77%), *UpdateVelsPoses* in Jackal UGV (4.69%), and *getOdomPose* in Turtlebot3 (2.74%). Overall, the increased percentage of execution time is highly related to the number of edges in the task dependency graph (TDFG). Typically, tasks that involve more sensor inputs will introduce more edges. For example, the *MOTUpdate* task has a high overhead percentage because it is the multiple object tracking task in Autoware that fuses multiple pointcloud inputs. Furthermore, the tasks that maintain more historical timing states will also have a higher overhead. An example is the task *AddImuData* in Jackal UGV, which stores hundreds of inertial data frames in a queue, inducing a 4.21% overhead. Breaking it down, most of the overhead comes from propagating timing information along edges which can reach up to 4.39%. We found that Kairos’s add-on logic on schedulers introduces negligible overhead, where the largest overhead is 0.84% from task *AddImuData*. Besides the individual execution times, we also measure the end-to-end latency from sensor input reading to actuation output. The overhead on end-to-end latencies for Autoware, Jackal UGV, and TurtleBot3 are 3.24%, 2.44%, and 2.75%, respectively.

Scalability Analysis. The sensor reading rate, the number of edges in TDFG, and the number of timing tags in timing constraint checking affect the scalability of Kairos. Thus, we evaluate the scalability of Kairos by measuring the runtime overhead with respect to these three factors. We create synthesized workloads with varying scalability impact factors by modifying the original workload of ORB-SLAM3. Specifically, (1) to emulate varying sensor reading rates, we change the replay speed of the recorded sensor data from the original ORB-SLAM3 workload. (2) To adjust the number of edges, we duplicate tasks and annotate timing constraints on their shared data flows. (3) Since only *stability* performs timing constraint checks over multiple timestamps, we adjust its window size to assess the impacts of timing tag size.

Sensor Reading Rate. The sensor reading rate impacts runtime overhead. Figure 16(c) shows a linear increase in execution times for updating and checking timing tags as input frequency rises. The execution time for a single vertex increases from 37.537 μs at a frequency of 20 to 4087.95 μs at a frequency of 500Hz, primarily due to timing checks. A higher sensor reading rate also significantly increases lock wait times, increasing 100 times from 10 Hz to 500 Hz. Yet, CPU usage on a single core remains at just 0.23% even under high input frequency.

Number of Edges in Timing Propagation. We use the number of Paths created per second as a proxy for the size of TDFG. Figure 16(b) presents the runtime and memory overheads. The duplicated tasks are callback workers in the ROS middleware, thus increasing the number of tasks in the middleware but not significantly affecting the kernel scheduler. We observe that middleware scheduling time increases with the number of tasks. CPU usage on a single core peaks at 0.42% with

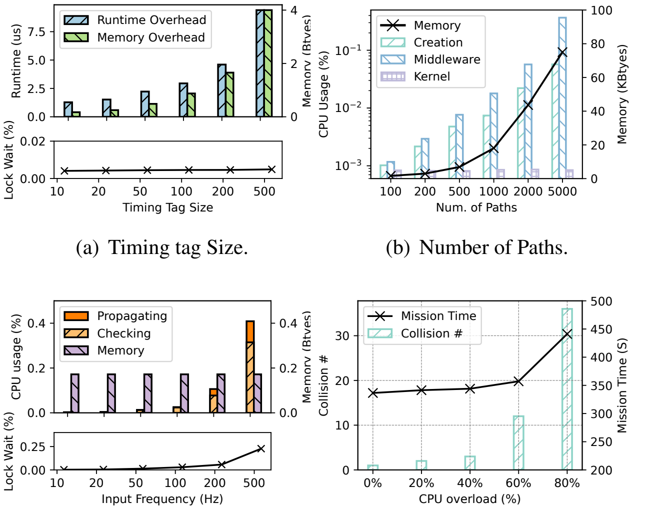


(a) Autoware

(b) Jackal UGV

(c) Turtlebot3

Figure 15: Run-time overhead breakdown. The execution time of the original task, the time spent logging timing information, online checking of timing correctness, and extra scheduling are shown.



(a) Timing tag Size.

(b) Number of Paths.

(c) Input frequency.

(d) Different CPU overloads.

Figure 16: Scalability analysis (16(a), 16(b), and 16(c)) and control impact of CPU overload 16(d).

5000 Paths, while memory overhead reaches 75 KB, which is relatively low given that the target platform typically has over ten GB of RAM.

Number of Timing Tags. Figure 16(a) shows the runtime and memory overhead induced by a single edge with different numbers of timing tags used during timing constraint checks. We can observe that the checking time increases proportionally with the number of timing tags. With a tag size of 100, the average overhead is 2.945 μs of runtime and 896 bytes of memory; increasing to 500, it reaches 9.4 μs and 4096 bytes, respectively. Given that the number of edges typically remains below a few hundred, the total overhead is low. The figure also shows that as the number of timing tags increases, lock wait time slightly rises but remains below 0.05% of CPU usage on one core.

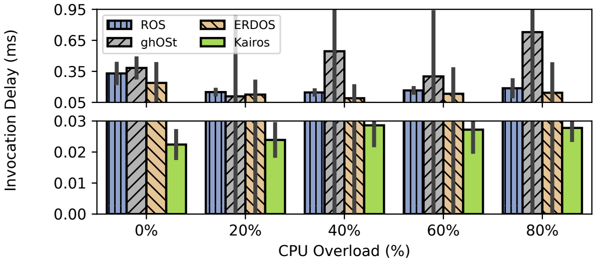


Figure 17: Invocation delays of handlers in different systems.

Invocation Latency in Delivering Execution Decision. In this experiment, we compared Kairos's efficacy to ROS [80], ghOSt [57] and ERDOS [55] in delivering scheduling decisions. ROS features the actionLib library [28], which supports preemptible tasks. ghOSt [57] is a userspace-informed kernel scheduling system that allows scheduling decisions to be made from userspace. EDROS is a robotic middleware that provides programming interfaces to deploy deadline miss handlers. We measure the delay from when the decision is made to the targeted task being executed. The experiments are conducted under different CPU overloads. We use *stress-ng* [62] tool to inject overloads, from 0% to 80%, then compare the increases and variations of latencies in Figure 17.

Under 80% overload, the response times for ROS, ghOSt, ERDOS, and Kairos are 0.186 ms, 0.72 ms, 0.14 ms, and 0.027 ms, respectively. Kairos has the fastest response time in fulfilling an execution decision, at least 2.16× faster than the others under high system overload. To be fair, these systems do not aim to achieve performance under high overload. Kairos has the cooperation in prioritizing the target task across scheduling layers. In contrast, ROS and ERDOS enforce scheduling decisions only at the middleware layers. Similarly, ghOSt enforces them solely through its own scheduler, which is a sub-scheduler with a lower priority than Linux's CFS scheduler. All four systems have stable invocation times while CPUs are idle, with variations of 0.095 ms, 0.094 ms, 0.18 ms, and 0.004 ms, respectively. However, we observe that the invocation latency and variations on ghOSt and ERDOS increase significantly (up to 0.81 ms under 80 % CPU overload) as the system overhead increases. This is because they

Table 5: DFA on Different Platforms

Platforms	Loc	# Inputs	# Tasks	# Vertices	Deviation (m)		# Collisions	
					Native	DFA	Native	DFA
Autoware	92 k	8	16	14	0.67	0.13	45	16
Jackal	68 k	4	6	6	0.27	0.09	12	4
Turtlebot3	34 k	3	4	3	0.87	0.21	35	13

are highly affected by Linux's underlying native scheduler (CFS). We conclude that Kairos takes faster and more stable countermeasures when mitigating a timing violation.

6.3 Effectiveness in Improving Safety

This section evaluates the capability of DFA model and Kairos in improving performance/safety in abnormal timing situations. We performed dynamic profiling on three platforms to identify which code regions required annotated timing constraints and to determine the expected temporal properties. Regarding handling policies, they also require understanding the task model and semantics of the target program. To mitigate the impact of this subjectivity, we employed an automatic strategy to apply three default policies for these timing constraints accordingly. Specifically, we model the target application's tasks as a directed graph. We adopted the *prioritize* policy for tasks on the critical path since aborting them will significantly increase end-to-end response time. For the tasks on the non-critical path, we apply the edges with *freshness* constraints and the *abort* policy for the remaining tasks. This is because violations of *consistency* and *stability* often lead to erroneous computation results, which should be prevented from propagating to downstream tasks. Table 5 shows the number of inputs and tasks on the three platforms as well as the number of edges annotated with timing constraints in TDFG. In generating the timing thresholds for those timing constraints, we observed an average variation of 8.82 ms.

We generated 100 trajectories in each scenario (Autoware in Parking Lot [34], Jackal UGV in Office [48] and Turtlebot3 in House [81] scenarios.)) for the vehicles to follow. During navigation, we injected CPU overload using the *stress-ng* tool [62] to emulate abnormal timings. We selected a 60% overload, as this condition typically triggers notable degradation in control performance. Figure 16(d) shows the number of collisions of Jackal UGV in 100 runs under overloads. We observed a significant increase in collisions at 60 %. Additionally, mission time increased with CPU overload because higher overload often triggered fail-safe, stopping the vehicle during the mission. At 80 % overload, vehicles typically halted, requiring manual intervention to continue.

Control Performance Improvements. The control performance is quantified by metrics (1) the distance vehicles deviate from the reference mission trajectories and (2) the number of collisions. The results are shown in Table 5. We observe that Kairos can considerably reduce control deviations across all three platforms. The lowest improvement of deviation is 2.97× on the Jackal UGV. As to collisions, Kairos reduces the collision by 64.4%, 83.3%, and 62.9% on Autoware, Jackal UGV, and Turtlebot3 respectively. Upon further investigation, the improvement is mainly due to proactively aborting false computational results to prevent the vehicle from outputting erroneous actuation commands. However, this approach will slow down the vehicle, which increases the mission time.

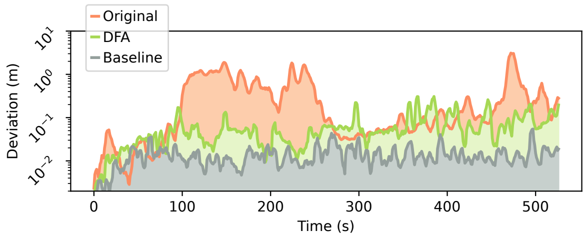


Figure 18: Comparison of control performance on Jackal UGV, with the deviation averaged over a window of 10.

Figure 18 plots the localization errors over time of one test run on Jackal UGV. We see that Kairos's abnormal timing detection and handling mechanism can significantly reduce the magnitude of intermittent computational errors, from ∼ 1m level to ∼ 10cm level. In this case, nearly one-third of the frames are dropped under high system overload, preventing the software from using data with abnormal timings and avoiding erroneous computational results. Aborting or skipping data also slows the robot's movement, reducing collision risks but increasing mission time by 56.8%. This policy may not suit hard real-time systems with strict deadline requirements. However, it effectively reduces adverse control outcomes in soft real-time systems.

7 Discussion and Limitation

Expressing Real-time Computing Constructs using DFA. DFA approaches timing assurance from a data-flow perspective, providing a more intuitive mechanism to express, detect, and mitigate timing constraint violations. However, to leverage this system coherently with existing constructions of real-time systems, DFA has to be able to express traditional real-time primitives. This will allow developers to build on top of the extensive advancements in real-time theory from the past several decades using the DFA-expressed real-time primitives. There are two categories of real-time computing primitives. The first category is execution time constraints, which specify the bounds of the execution period between two statements in the program. With Kairos, developers can use the statements *freshness(var, delay, abort)* and *freshness(var, delay, policy)* as firm and soft deadline specifications, where *var* is the task's output and *delay* is the deadline. The second category is synchronization primitives, which specify the order of shared data accesses among multiple real-time threads. This can be expressed by strictly ordering the timing of two data flows using statements such as *consistency(write_var, read_var, 0, policy)*, where *read_var* and *write_var* are SSA variables in read and written by different concurrent threads. This annotation will ensure that the write operation occurs before the read operation.

Manual Efforts. While Kairos provides tools for profiling through dynamic analysis, the search space for where and what temporal constraints to include is often prohibitively large. As a result, the automated tool may take a long time to identify the appropriate temporal policy. Developer guidance with some manual annotations can quickly narrow down this search space. Furthermore, once a policy is found, deploying it in a safety-critical system may require re-validation or even re-certification of the target system.

Multiple System Components in Violation Handling. Kairos requires seamless collaboration between the software instrumentation and multiple components across different scheduling layers for effective violation mitigation. This interdependency poses two limitations. First one is on reliability, as failures in one part can affect the entire system. Second one is maintainability, as migrating to different platforms may require substantial engineering efforts. However, the modularized design of detection and mitigation of temporal violations allows Kairos to integrate with other existing detection or mitigation techniques. Additionally, the infrastructure that bridges the semantic gap between different abstraction layers also reduces the engineering effort needed to build cross-layer timing mitigation.

Generality of DFA. While DFA is designed for cyber-physical systems, the concept of imposing temporal expectation on data flow generally applies to broader classes of computing, including conventional cyber-only environments such as data centers. For example, DFA's timing constraints on data usage can be adapted to systems with non-determinism to ensure logical correctness, such as the order of input events in distributed systems [74]. It is also possible to leverage DFA to track computation progress through the lens of data flow in distributed workloads.

8 Related Work

Timing Semantics in Programming Model. In data streaming systems, there have been efforts that incorporate timing information into the programming model to represent logical points, such as logic timestamps or watermarks [30,74,88,92]. This facilitates the coordination of computation among distributed nodes. Such extension of timing information on data-flow graphs inspired our design. However, these systems are designed for massive parallel data processing, rather than the cyber-physical timing alignment.

In real-time computing, several programming models have been proposed to react to timing violations [42,44,55,76,87]. In particular, Timed C [76] is a dialect of C that allows the specification of soft and firm real-time constraints. However, compared to these works, DFA introduces a design approach that focuses on the temporal policy on data flows, which builds on top of the cyber-physical control loop abstraction, allowing the detection and mitigation of cyber-physical state (data) misalignment.

Cross-layer Scheduling. There is a large body of work that focuses on cross-layer scheduling. However, existing works often target specific hardware [40,58,78,95], such as NICs. Furthermore, many target server platforms have abundant computation power, thus these solutions may not translate well to resource-constrained embedded systems. Notably, similar to Kairos, Syrup [60] offers programmable abstractions and interfaces for custom scheduling policies. However, it focuses on the rapid deployment of customized schedulers, rather than on enabling cross-layer scheduling actions.

Cross-layer scheduling has also been studied in the real-time community in the context of compositional scheduling [51, 83, 93]. However, deployment of these techniques often requires the target system to be rigorously modeled and deployed as real-time tasks, which may not always fit some of the existing software architectures for CPS.

9 Conclusion

In this paper, we presented data-flow availability, a concept that aims to define temporal policy for data-flow in real-time safety-critical cyber-physical systems. Through a bug study of 189 issues over 7 representative CPS software, three key temporal properties were extracted concerning the alignment of cyber states and physical states in time. To allow for the concrete expression of temporal expectation, we augment data-flow with timing constraints, captured by TDFG. To realize the concept in system, we design and develop Kairos that detects temporal violations by embedding the policy as checks in the application and mitigates them via a cross-layer scheduling infrastructure. Lastly, the system is evaluated on three CPS platforms for feasibility.

Acknowledgment

We express our gratitude to the anonymous reviewers and shepherd for their insightful feedback. We also thank Sanjoy Baruah and Ron Cytron for their valuable discussions. This work was partially supported by the NSF (CNS-2238635) and Intel.

References

[1] Google cartographer ros for the toyota hsr. https://google-cartographer-ros-for-the-toyota-hsr.readthedocs.io/en/latest/. Accessed: 2024-04-18.

[2] Moveit. https://github.com/ros-planning/moveit. Accessed: 2023-04-15.

[3] Orb-slam2. https://github.com/raulmur/ORB_SLAM2. Accessed: 2023-04-15.

[4] Orb-slam3 github. https://github.com/UZ-SLAMLab/ORB_SLAM3. Accessed: 2023-04-15.

[5] Ros navigation. https://github.com/ros-planning/navigation. Accessed: 2023-04-15.

[6] Ros navigation stack. https://github.com/ros-planning/navigation. Accessed: 2023-10-04.

[7] Ros rcl. https://github.com/ros2/rcl. Accessed: 2023-04-15.

[8] Cartographer #153 compute the common start time per trajectory. https://github.com/cartographer-project/cartographer/pull/153, 2016. Accessed: 2023-5-13.

[9] Cartographer #8 adds rate timer. https://github.com/cartographer-project/cartographer/pull/8, 2016. Accessed: 2023-05-22.

[10] Cartographer #242 improve 2d velocity estimation to be less fragile to poor data timing. https://github.com/cartographer-project/cartographer/issues/242, 2017. Accessed: 2023-05-24.

[11] Apollo #4492 timestamp correction in conti_radar. https://github.com/ApolloAuto/apollo/issues/4492, 2018. Accessed: 2024-05-20.

[12] Cartographer #1033 store timestamp of the latest range data in submap*d. https://github.com/cartographer-project/cartographer/pull/1033, 2018. Accessed: 2023-05-22.

[13] Cartographer #1275 add metrics: real time ratio and cpu time ratio. https://github.com/cartographer-project/cartographer/pull/1275, 2018. Accessed: 2023-05-13.

[14] Cartographer #1495 add serialization for timestampedtransform. https://github.com/cartographer-project/cartographer/pull/1495, 2019. Accessed: 2023-05-13.

[15] Moveit #1299 preempt trajectory execution if one controller aborts. https://github.com/ros-planning/moveit/issues/1299, 2019. Accessed: 2023-05-22.

[16] Ros 2 rclcpp #694 fixup time. https://github.com/ros2/rclcpp/pull/694, 2019. Accessed: 2023-11-22.

[17] Autoware.auto #1002 add predict/update functions receiving timestamp to kalman filter. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/1002, 2020. Accessed: 2023-05-22.

[18] Autoware.auto #65 ros 2 and real-time. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/65, 2020. Accessed: 2023-05-22.

[19] Moveit #232 fix race conditions when updating planningscene. https://github.com/ros-planning/moveit/pull/232, 2020. Accessed: 2023-05-22.

[20] Moveit #2395 fix pose tracking race condition. https://github.com/ros-planning/moveit/pull/2395, 2020. Accessed: 2023-05-22.

[21] Orb-slam2 #946 time stamps are not used in motion model part of tracking. https://github.com/raulmur/ORB_SLAM2/issues/946, 2020. Accessed: 2023-05-22.

[22] Ros 2 rclcpp #1121 lock-order-inversion (potential deadlock). https://github.com/ros2/rclcpp/issues/1121, 2020. Accessed: 2023-05-23.

[23] Autoware.auto #605 record replay_planner does not continuously update the trajectory. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/605, 2021. Accessed: 2023-05-22.

[24] Autoware.auto #821 detect when nodes' incoming messages are skipped. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/821, 2021. Accessed: 2023-05-22.

[25] Autoware.auto #980 updated ne raptor interface to send messages periodically. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/merge_requests/980/diffs, 2021. Accessed: 2023-05-22.

[26] Ros 2 rclcpp #1679 action client feedback callback does not reliably trigger. https://github.com/ros2/rclcpp/issues/1679, 2021. Accessed: 2023-05-22.

[27] Ros 2 rcl #967 problems with arguments in rcl_timer_exchange_period api. https://github.com/ros2/rcl/issues/967, 2022. Accessed: 2023-05-22.

[28] Ros actionlib. http://wiki.ros.org/actionlib. Accessed: 2022-10-10.

[29] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11. IEEE, 2020.

[30] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[31] Amazon airprime. https://www.aboutamazon.com/news/transportation/amazon-prime-air-drone-delivery-mk30-photos. Accessed: 2023-11-30.

[32] Amazon astro. https://www.aboutamazon.com/news/devices/meet-astro-a-home-robot-unlike-any-other. Accessed: 2022-01-10.

[33] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 193–202. IEEE, 2001.

[34] Autoware Foundation. Autonomous valet parking demonstration. https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/avpdemo.html, 2020. Accessed: 2023-12-05.

[35] Autoware Foundation. Past, present, and the future of autoware. https://autoware.org/past-present-and-the-future-of-autoware/, 2023. Accessed: 2024-04-18.

[36] Autoware.auto project. https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/. Accessed: 2022-08-15.

[37] Baidu. Apollo self-driving project. https://github.com/ApolloAuto/apollo. Accessed: 2022-08-15.

[38] Baidu. Apollo #9433: Cyberrt, coroutine to thread mapping, 2019. Accessed: 2023-05-22.

[39] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154, 2012.

[40] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[41] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B Brandenburg. Automatic latency management for ros 2: Benefits, challenges, and open problems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 264–277. IEEE, 2021.

[42] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[43] Alan Burns. Mixed criticality systems-a review.

[44] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[45] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23:7–24, 2002.

[46] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic Task Model for Adaptive Rate Control. In *IEEE Real-Time Systems Symposium*, 1998.

[47] Google cartographer. https://github.com/cartographer-project/cartographer. Accessed: 2022-11-21.

[48] Clearpath Robotics. Additional simulation worlds. Jackal Tutorials 0.6.0 Documentation, 2020. Accessed: 2023-12-05.

[49] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[50] davetcoleman. Moveit #294 isvalidvelocitymove() for checking maximum velocity between two robot states. https://github.com/ros-planning/moveit/pull/294, 2016. Accessed: 2023-11-22.

[51] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using edp resource models. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 129–138. IEEE, 2007.

[52] Facebook Engineering. Slam: Bringing art to life through technology. https://engineering.fb.com/2017/09/21/virtual-reality/slam-bringing-art-to-life-through-technology/, September 2017. Accessed: 2024-04-18.

[53] Autoware Foundation. Autoware.auto. Accessed: 2023-04-15.

[54] Gaschler. Cartographer #936 gracefully handle time-overlapping point clouds. https://github.com/cartographer-project/cartographer/pull/936, 2018. Accessed: 2023-11-22.

[55] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022.

[56] Google. Cartographer. https://github.com/googlecartographer/cartographer. Accessed: 2023-04-15.

[57] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[58] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[59] Jackal ugv. https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/. Accessed: 2021-07-30.

[60] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[61] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[62] Colin King. stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng. Accessed: May 20, 2022.

[63] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[64] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, pages 517–530, 2016.

[65] Ao Li, Marion Sudvarg, Han Liu, Zhiyuan Yu, Chris Gill, and Ning Zhang. Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 225–239. IEEE, 2022.

[66] Ao Li, Jinwen Wang, Sanjoy Baruah, Bruno Sinopoli, and Ning Zhang. An empirical study of performance interference: Timing violation patterns and impacts. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[67] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[68] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[69] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-system stability under consecutive deadline misses constraints. In *32nd euromicro conference on real-time systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[70] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[71] Pau Marti, Josep M Fuertes, Gerhard Fohler, and Krithi Ramamritham. Jitter compensation for real-time control systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 39–48. IEEE, 2001.

[72] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1821–1827. IEEE, 2018.

[73] David Mosberger and Larry L Peterson. Making paths explicit in the scout operating system. In *OSDI*, volume 96, pages 153–167, 1996.

[74] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[75] Picknik robotics wins space force, nasa contracts. https://www.therobotreport.com/picknik-robotics-wins-space-force-nasa-contracts/. Accessed: 2024-04-18.

[76] Saranya Natarajan and David Broman. Timed c: An extension to the c programming language for real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 227–239. IEEE, 2018.

[77] José Carlos Palencia and M González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 26–37. IEEE, 1998.

[78] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[79] Cartographer Project. Fix division by velocity. https://github.com/cartographer-project/cartographer/commit/b4b83405ce4009ea0c1ac22c7ab9edeeb9d48a42, 2017. Commit b4b834.

[80] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[81] ROBOTIS. Turtlebot3 simulation. https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/#gazebo-simulation. Accessed: 2024-05-20.

[82] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. Cancellation in systems: An empirical study of task cancellation patterns and failures. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 127–141, 2022.

[83] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67. IEEE, 2004.

[84] Marion Sudvarg, Chris Gill, and Sanjoy Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, 10 2021.

[85] Marion Sudvarg, Ao Li, Daisy Wang, Sanjoy Baruah, Jeremy Buhler, Chris Gill, Ning Zhang, and Pontus Ekberg. Elastic Scheduling for Harmonic Task Systems. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[86] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[87] Milijana Surbatovich, Limin Jia, and Brandon Lucia. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 851–866, 2021.

[88] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[89] Turtlebot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2022-09-10.

[90] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE international real-time systems symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[91] Nils Vreman, Anton Cervin, and Martina Maggio. Stability and performance analysis of control systems subject to bursts of deadline misses. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[92] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta,

et al. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2602–2613, 2021.

[93] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022.

[94] Waymo driveless service in phoenix. https://blog.waymo.com/2020/10/waymo-is-opening-its-fully-driverless.html. Accessed: 2022-01-10.

[95] Chuanyu Xue, Tianyu Zhang, Yuanbin Zhou, Mark Nixon, Andrew Loveless, and Song Han. Real-time scheduling for 802.1qbv time-sensitive networking (tsn): A systematic review and experimental study. In *2024 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.

Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel

Haibo Chen[1,2], Xie Miao[1], Ning Jia[1], Nan Wang[1], Yu Li[1], Nian Liu[1], Yutao Liu[1], Fei Wang[1], Qiang Huang[1], Kun Li[1], Hongyang Yang[1], Hui Wang[1], Jie Yin[1], Yu Peng[1], and Fengwei Xu[1]

[1]*Huawei Central Software Institute,* [2]*Shanghai Jiao Tong University*

Abstract

The virtues of security, reliability, and extensibility have made state-of-the-art microkernels prevalent in embedded and safety-critical scenarios. However, they face performance and compatibility issues when targeting more general scenarios, such as smartphones and smart vehicles.

This paper presents the design and implementation of HongMeng kernel (*HM*), a commercialized general-purpose microkernel that preserves most of the virtues of microkernels while addressing the above challenges. For the sake of commercial practicality, we design *HM* to be compatible with the Linux API and ABI to reuse its rich applications and driver ecosystems. To make it performant despite the constraints of compatibility and being general-purpose, we re-examine the traditional microkernel wisdom, including IPC, capability-based access control, and userspace paging, and retrofit them accordingly. Specifically, we argue that per-invocation IPC is not the only concern for performance, but IPC frequency, state double bookkeeping among OS services, and capabilities that hide kernel objects contribute to significant performance degradation. We mitigate them accordingly with a set of techniques, including differentiated isolation classes, flexible composition, policy-free kernel paging, and address-token-based access control.

HM consists of a minimal core kernel and a set of least-privileged OS services, and it can run complex frameworks like AOSP and OpenHarmony. *HM* has been deployed in production on tens of millions of devices in emerging scenarios, including smart routers, smart vehicles and smartphones, typically with improved performance and security over their Linux counterparts.

1 Introduction

Microkernels minimize functionality in the kernel and move components, such as file systems and device drivers, into well-isolated and least-privileged OS services, achieving better reliability, security, and extensibility than monolithic kernels such as Linux. Thanks to these virtues, state-of-the-art (SOTA) microkernels have been widely deployed in embedded and safety-critical scenarios [30, 52, 54].

On the other hand, while monolithic kernels like Linux dominate in general-purpose scenarios such as servers and the cloud, there are increasingly emerging scenarios such as smart vehicles and smartphones that require better security, reliability, and extensibility in addition to good performance, where Linux is less suitable. While being general, Linux evolves more towards servers and the cloud, making other scenarios less beneficial. For example, it took over 10 years for the preemptive-RT patch [1] to be partially merged, and its evolution is still out of the mainstream, let alone other domain-specific strategies [20, 21]. Moreover, it has been doomed to be difficult (if possible) for Linux to satisfy high-level industry certifications required for such scenarios [98, 113].

However, although microkernels have been extensively studied for decades [16, 28, 30, 46, 49, 52, 52–54, 64, 67, 73, 75, 76, 86], SOTA microkernels mainly target some specific domains, e.g., embedded and safety-critical ones. They usually use static resource partitioning and allocation, and lack general OS functionalities to run commercial off-the-shelf applications. Below, we summarize the major challenges in retrofitting a microkernel as a general OS kernel for such emerging scenarios.

Compatibility: POSIX subset-compliant is not enough. Rebuilding the entire software ecosystem is impractical. Therefore, SOTA microkernels, such as seL4 [67] and Zircon [46], achieve minimal POSIX subset compliance by providing custom libraries, e.g., musl-libc [47], that generate inter-process calls (IPC) to OS services. However, they face deployment issues [6, 116], e.g., not being binary compatible, and implementation challenges, e.g., `fork` and `poll`, in emerging scenarios. Moreover, they can hardly reuse device drivers with affordable engineering effort and uncompromised performance, which are crucial for production deployment.

Performance: IPC is not the only concern. Performance is the top priority in emerging scenarios, directly determining user experiences. While SOTA microkernels like seL4 [67] and recent architectural support [28, 49, 86] have achieved record-high IPC performance, we observe that they still cause non-trivial performance overhead because IPC frequency is significantly increased when microkernels go general (70x higher in smartphones than routers). Further, we observe equally severe performance issues caused by state double bookkeeping due to the multi-server design, which introduces additional performance overhead (2x slower than Linux) and memory footprint (35%). Moreover, capability-based access control, which hides frequently updated kernel objects behind capabilities, can cause significant overhead due to frequent invocations. For example, it causes page fault handling to be 3.4x slower than Linux.

We started the HongMeng kernel (*HM*) project over 7 years ago to re-examine and retrofit the microkernel into a general OS kernel for emerging scenarios. To be practical for production deployment, *HM* achieves full Linux API/ABI compatibility and is capable of reusing the Linux applications and driver ecosystems such that it can run complex frameworks like AOSP [42] and OpenHarmony [35] with rich peripherals. Despite the compatibility goal that may constrain its performance, *HM* still puts performance as its primary emphasis. Therefore, *HM* respects the design principles of microkernels but not to the extreme with careful compromises. Specifically, *HM* makes the following key design decisions.

Minimal microkernel with least-privileged and well-isolated OS services. HM retains the minimality principle by keeping only the necessary functionality in the core kernel, including thread scheduler, serial/timer drivers, and access control, and leaving all other components as isolated OS services (multi-server) outside the core kernel. In addition, *HM* adopts fine-grained access control to preserve the principle of least privilege for better security. As a result, *HM* inherits the security and reliability benefits of microkernels.

Maximizing compatibility by achieving Linux API/ABI-compliant and performant driver reuse. HM integrates existing software ecosystems by achieving full Linux API/ABI compatibility through ABI-compliant shim that identifies and redirects Linux syscalls to IPCs. Moreover, *HM* reuses unmodified Linux drivers via a driver container that provides Linux runtime atop *HM* with minor engineering effort, and eliminates critical path performance degradation by separating the control plane and the data plane with *twin drivers*.

Performance first by structural supports. HM prioritizes performance without violating the architectural principles of microkernels. Specifically, *HM* achieves flexible composition for hierarchically relaxing the isolation between trusted services to minimize IPC overhead, and coalesces tightly coupled services to minimize IPC frequency and eliminate state double bookkeeping in performance-demanding scenarios, while maintaining the ability to separate them in security-critical scenarios. *HM* also supplements capabilities with performant address token-based access control, facilitating efficient cooperation like policy-free kernel paging.

We have deployed *HM* on tens of millions of devices, including smart routers, smart vehicles, and smartphones, which provides not only better security and reliability but also better performance than their Linux counterparts. The critical components of *HM* are semi-formally verified [55] by formally specifying the design and using automated verification and verification-guided testing to validate the crucial security properties, such as free of integer and buffer overflow. *HM* has been certified with ASIL-D [61] (for safety) and CC EAL 6+ [62] (for security). In routers, *HM* allows 30% more client connections by reducing 30% system memory footprint. In vehicles, *HM* achieves a 60% faster boot time and a 60% lower cross-domain latency. In smartphones, *HM* achieves 17% shorter app startup time and 10% less frame drops.

2 The Case for a General Microkernel

2.1 Microkernel Review

A major hallmark of microkernels is the minimality principle [73, 76], which minimizes functionality in the core kernel and moves other functions to userspace services. SOTA microkernels also adopt capability-based fine-grained access control [46, 52, 67, 74] to preserve the least privilege principle. As a result, microkernels are inherently more secure, reliable, and extensible than monolithic kernels [12, 79].

However, although microkernels have been extensively studied for decades [16, 30, 52–54, 64, 67, 73, 75, 76, 122], SOTA microkernels primarily target specific domains, such as embedded and safety-critical systems. Examples include L4-embedded in Qualcomm cellular modem chips [30], QNX[1] in cars and embedded systems [54], and Zircon (kernel of Fuchsia) in smart speakers [46]. There has been little study on how microkernels could be extended as general OS kernels for emerging scenarios like smart vehicles and smartphones.

The industry adopted hybrid kernels such as Windows NT [88] and Apple XNU [4], which combine a core microkernel, e.g., Mach in XNU, with all other services (as a whole) in the kernel space, e.g., Executive in NT and BSD in XNU. Although hybrid kernels also minimize functionality in the core kernel, they do not inherit many advantages of microkernels. For example, OS services in hybrid kernels are not least privileged and not well isolated. Thus, any compromised or buggy OS services can corrupt the system [88], potentially causing severe consequences, such as corrupting user data.

2.2 Demand for a General Microkernel

Emerging scenarios like smart vehicles and smartphones demand rich peripherals and applications. For example, the industry standard of vehicles has evolved to require richer OS

[1] While QNX once supported tablets/phones [14] and ran AOSP apps via virtual machine, QNX discontinued this due to limited compatibility and performance [15, 110] and has fully transitioned to embedded markets [13].

functionalities [7]. Meanwhile, emerging scenarios also emphasize security and safety. For instance, vehicles require high reliability for passenger safety, and smartphones require enhanced security to protect sensitive data. We list the major differences from domain-specific scenarios below.

Software ecosystem. In domain-specific scenarios, applications are mostly *customized and source-available*. Thus, being POSIX-compliant is believed to be sufficient for application transplanting (not even true based on our deployment experiences). However, in emerging scenarios like smartphones, apps and libraries are typically distributed in *binary form*, and frameworks require *more than POSIX compliance* [6], which mandates Linux ABI compatibility.

Resource management. In domain-specific scenarios, there are only a few pre-determined applications, and the hardware resources are limited. Therefore, applications mostly *manage resources themselves*, and the kernel is primarily responsible for reserving resources. In emerging scenarios, however, competing applications require *coordinated resource management*. The kernel requires more fledged functionalities such as efficient resource management and fair allocation.

Performance. In domain-specific scenarios, microkernels prioritize security and strict resource (e.g., timing) isolation for mostly static applications, where performance is *not a primary concern*. In emerging scenarios, however, performance is also *a top priority*, which directly determines the user experience and, thereby, the widespread deployment of the kernel.

The call for integrating both rich software ecosystems and functionalities, as well as security and reliability, makes it difficult for existing OS to satisfy them simultaneously. One approach would be customizing a stock OS such as Linux for such scenarios, which is unfortunately very expensive to evolve with upstream (section 2.3). Previous work also proposes various architectures, including unikernel [65, 81, 102], multikernel [9], exokernel [31], and splitkernel [109]. However, they primarily target server scenarios with clear resource separation while lacking support for efficient and coordinated resource management required in emerging scenarios. Moreover, the synchronization overhead and complexity introduced by split states make it challenging to achieve compatibility.

Therefore, we believe it is worthwhile to explore another avenue of evolving the microkernel into a general OS kernel.

2.3 Issues with Linux

Linux has dominated the server and cloud markets and is increasingly penetrating other domains such as PC and embedded. However, it comes at the cost of compromised security, reliability, and performance, especially in emerging scenarios.

Security and Reliability. Linux modules such as file system (FS) and device drivers cover about 80% of its 30 million line code base. They contribute to the majority of defects and vulnerabilities (90% of the total 1000 CVE [23] in the last 4 years) and significantly reduce reliability and security [19]. Additionally, about 80% of these CVEs are data leaks that can be avoided with proper isolation. Therefore, a long line of research [18,25,38,48,56,83,90–92,100,105,106,112,120,123] aims at isolating the kernel from the modules in a compartmentalized manner. However, the inherent tight coupling requires significant engineering effort and even rewriting [56, 90, 91]. Moreover, the instability of kernel module APIs and security patches force frequent upgrades, making them less practical for real-world deployments.

Generality vs. Specialization. While Linux targets general scenarios, recent patches and features witness that innovations are primarily driven by servers and the cloud, which even hamper the performance of other scenarios [89, 103]. Moreover, the growing diversity of devices with rich peripherals and varied scenarios call for specialized strategies to exploit the performance and energy efficiency headroom, such as allocating resources according to the quality of service [20, 21] or minimizing space usage [119]. However, such strategies require significant engineering effort to customize the kernel due to the inherent tight coupling of kernel modules. While there is much effort [58, 66, 78, 84, 93] to improve customizability, it is hard to integrate them into the mainstream kernel.

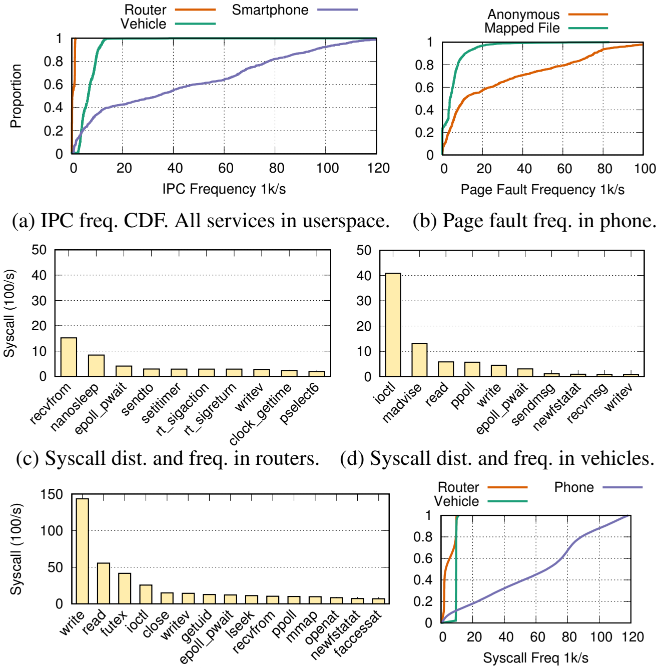
Customization vs. Evolution. Another issue is evolving the custom code. Synchronizing with upstream requires significant effort to reapply the changes, while not synchronizing may expose the system to security vulnerabilities. Years of production experience suggest that it is expensive due to the frequent changes in kernel internal APIs, and performance regressions require substantial effort to locate and even redesign the entire patches. This severely limits customizability in real-world deployments. Hence, a massive amount of products on the market are still running Linux 2.6 [50, 51, 117], which reached End-of-Life (EOL) 7 years ago [114] and has many known security vulnerabilities [24, 117].

3 Revisiting Microkernel for Going General

3.1 Microkernel at Scale

Deploying a microkernel in emerging scenarios poses challenges in both performance and compatibility. Figure 1 presents the observed characteristics of emerging scenarios from deploying *HM* in productions. For routers, we collected data directly from the production environment. For vehicles and phones, we replayed a typical usage (lasting 24 hours) derived from recorded massive amount of real-world executions at scale (anonymous and with user consent).

Observation 1: IPC frequency increases rapidly in emerging scenarios. Figure 1a shows the IPC frequency CDF in *HM* when configuring all OS services to be isolated in userspace. Smartphones (avg. 41k/s) and vehicles (7k/s) have a much higher IPC frequency than routers (0.6k/s, more similar to domain-specific scenarios). Figure 1b, 1e, and 1f illustrate it by showing the minor (i.e., not from disk/device)



(a) IPC freq. CDF. All services in userspace. (b) Page fault freq. in phone.

(c) Syscall dist. and freq. in routers. (d) Syscall dist. and freq. in vehicles.

(e) Syscall dist. and freq. in smartphones. (f) Syscall freq. CDF.

Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

page faults' frequency and the distribution and frequency of syscalls in phones. As shown in the figures, the high IPC frequency is not only caused by the higher syscall frequency (61k/s, 13x higher than routers), but also by invoking massive amounts of file operations (IPC to the FS), and triggering numerous page faults on memory-mapped files (5k/s), which requires an additional IPC roundtrip between the memory manager and the FS. Hence, we should not only optimize IPC performance but also minimize the IPC frequency.

Observation 2: Distributed multi-server causes state double bookkeeping. The minimality principle determines that there is no centralized repository for shared objects, such as the file descriptor (fd) and page caches, and distributes them in multiple places. However, as shown in Figure 1c-1e, applications in emerging scenarios frequently invoke functions like `poll` that rely on the centralized management of such states. Figure 2 further presents the CPU flame graph of application startup, which relies heavily on the performance of file mapping and is crucial to the user experience [45]. As marked in the figure, 16% of the time is spent on handling page cache misses, which introduces an additional IPC roundtrip and is 2x slower than Linux. Moreover, the double bookkeeping of page caches consumes an additional 50MB of memory on top of the 120MB base (FS+mem) in smartphones.

Observation 3: Capabilities inhibit efficient cooperation. Capabilities, which hide the kernel objects behind them, introduce significant performance overhead due to the frequent updating of some kernel objects (e.g., the page table) managed outside the kernel and inhibit efficient cooperation between them. For example, this may cause the handling of anonymous page faults 3.4x slower than Linux, which frequently occurs in smartphones (avg. 27k/s, 80% of minor page faults in Figure 1b) and adds a non-trivial overhead to the app startup time (4% in Figure 2).

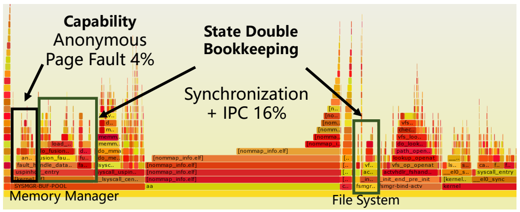


Figure 2: CPU flame graph of smartphone app startup in *HM*. Services coalescing and kernel paging are disabled.

Observation 4: Eco-compatibility requires more than POSIX compliance. Many SOTA microkernels achieve a minimal subset of POSIX compliance by providing custom runtime libraries [47] that link directly to applications and generate IPC to OS services. However, it faces deployment issues of being not binary compatible and requiring a customized building environment. Moreover, since Linux uses file as a unified interface, which no longer exists in the microkernel, it is also challenging to implement efficient fd multiplexing like `poll` and vectored syscalls like `ioctl`, which are frequently used in emerging scenarios as shown in Figure 1c-1e.

Observation 5: Deployment in emerging scenarios requires efficient driver reuse. When deploying *HM* on smartphones, we observe a massive increase in the number of drivers required to function correctly. For routers, fewer than 20 drivers are required (primarily maintained in-house), which increases to more than 700 for vehicles and phones. Our estimates indicate that it would take more than 5,000 person-years to rewrite those drivers, and it takes time to get mature and keep evolving. Thus, reusing device drivers is a more reasonable option. However, previous work, including transplanting the runtime environment of drivers [3, 17, 32, 41, 118] and using virtual machines [72], faces compatibility, engineering effort, and performance challenges (discussed in section 5.2).

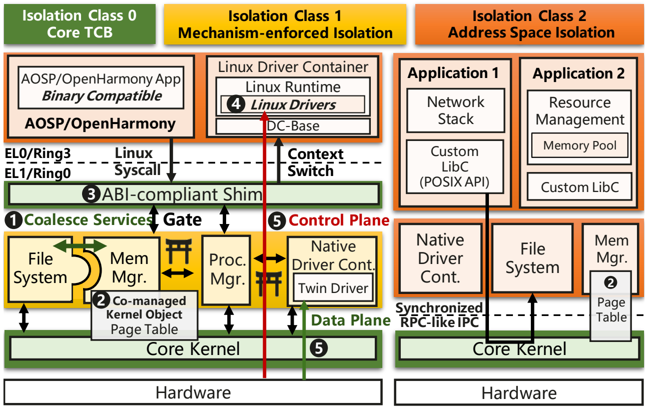
3.2 Overview of HongMeng

HM respects the core design principles of microkernels but not to the extreme, with careful compromises to address the performance and compatibility challenges in emerging scenarios. We summarize *HM*'s design decisions in Table 1 and list design principles below. Figure 3 shows *HM*'s overview.

Principle 1: Retain minimality. The security, reliability, and extensibility of microkernels derive from three fundamental architectural design principles, including separating policy and mechanism, decoupling and isolating OS services, and enforcing fine-grained access control. Hybrid kernels also enforce minimality through code decoupling but without proper isolation. Thus, it fails to inherit the major benefits of mi-

Table 1: Design decisions of HongMeng.

	SOTA Microkernels	Hybrid Kernels	HongMeng's Design
Minimality	Minimal Kernel	Code Decoupling	**Retained:** Minimal microkernel with isolated, least-privileged OS services.
IPC	IPC w/ Fastpath	Function Call	**Enhanced:** Synchronous RPC addresses resource alloc./exhaustion/acct. issues.
Isolation	Userspace Services	Coalesce w/ Kernel	**Flexibilized:** Differentiated isolation classes for tailored isolation and performance.
Composition	Static Multi-server	Static Single Server	**Flexibilized:** Flexible composition to accommodate diverse scenarios.
Access Control	Capability-based	Object Manager	**Extended:** Address tokens enable efficient kernel objects co-management.
Memory	Paging in Userspace	Paging in Kernel	**Enhanced:** Centralized management in a service with policy-free paging in kernel.
App Interfaces	POSIX-compliant	POSIX+BSD/Win	**Extended:** Linux API/ABI compatible via an ABI-compliant shim.
Device Driver	Transplanting/VM	Native Driver	**Enhanced:** Reusing Linux drivers efficiently via driver container with twin drivers.



(a) Smartphone (b) Smart Router

Figure 3: HongMeng overview. (a) and (b) show its composition in smartphones and routers. Different colors imply different isolation classes. ❶ coalesces coupled services. Address tokens enable kernel objects co-management ❷. ABI-compliant shim ❸ enables binary compatibility. Driver container ❹ reuses Linux drivers efficiently via data/control plane separation ❺.

crokernels. Therefore, while emphasizing compatibility and performance, *HM* respects the architectural design principles of microkernels.

HM keeps only minimal and necessary functionality in the core kernel, including thread scheduler, serial and timer drivers, and access control. All other functionality is implemented in isolated OS services, such as process/memory manager, drivers, and FS. Moreover, *HM* adopts fine-grained access control to preserve the least privilege principle. Services can only access strictly restricted resources (kernel objects) necessary for functionality. As such, *HM* inherits the security, reliability, and extensibility of microkernels.

> **Retained:** *Minimal microkernel with well-isolated and least-privileged OS services.*

Principle 2: Prioritize performance. The promising benefits of microkernels are compromised by architecture-inherent performance issues in emerging scenarios. Therefore, instead of enforcing uniform but overly strong isolation, *HM* provides structural support for assembling the system to satisfy both the performance and the security requirements. In particular, besides adopting an RPC-like fastpath that addresses the resource allocation/exhaustion/accounting issues (section 4.1), *HM* proposes differentiated isolation classes to reduce IPC overhead by relaxing the isolation between trusted OS services (section 4.2). *HM* also coalesces tightly coupled OS services (❶ in Figure 3) to minimize the IPC frequency in performance-demanding scenarios (section 4.3). Moreover, *HM* enables efficient kernel objects co-management (❷) by supplementing capabilities with address tokens (section 4.4), which facilitates policy-free in-kernel paging of anonymous memory (section 4.5).

> **Flexibilized:** *Prioritize performance by providing structural supports for flexible assembly to adapt to diverse scenarios.*

Principle 3: Maximizing eco-compatibility. *HM* integrates with existing software ecosystems by achieving Linux ABI compliance through a shim (❸) that redirects all Linux syscalls to appropriate OS services and serves as a central repository to store and translate Linux abstractions (e.g., fd) to efficiently support functions like `poll` (section 5.1). Moreover, *HM* reuses unmodified Linux device drivers via driver container (❹), which provides the necessary runtime derived directly from the mainline Linux with minor engineering effort (section 5.2). *HM* further improves drivers' performance by exploiting control and data plane separation (❺).

> **Enhanced:** *Maximize compatibility by achieving Linux API/ABI-compliant and performant driver reuse.*

***HM*'s Threat Model.** *HM* retains the architectural design principles of microkernels, thus maintaining a similar threat model to SOTA microkernels, which prevents malicious applications and OS services from accessing other's memory and ensures the confidentiality, integrity, and availability (CIA) properties of data, with the following differences.

First, since applications in emerging scenarios require centralized memory management for compatibility reasons (section 4.5), the memory manager (the only exception), including its coalesced services (only FS in phones on deployment), can inevitably access applications' address spaces. Besides, in safety-critical scenarios where memory is self-managed, *HM* does not create such a centralized memory manager.

Moreover, for the sake of performance, there are compromises on additional attack surfaces (section 4.2), different partitioning of failure domains (section 4.3), and additional data leakage possibilities on carefully selected objects (will not corrupt the kernel, section 4.4). The detailed security design will be discussed in the corresponding section.

4 Performance Design of HongMeng

4.1 Synchronous RPC-like IPC Fastpath

Microkernels use IPC to invoke OS services. A long line of research has proposed numerous optimizations to minimize IPC overhead. However, when applying them to emerging scenarios, we encountered several severe issues, either previously neglected or caused by changed assumptions. *HM* carefully addresses these issues, as summarized in Table 2.

Table 2: Comparison of IPC in *HM*.

	IPC Fastpath	Migration	HongMeng IPC
Bypass Scheduling	Yes	Yes	Yes
Reduced Switches	N/A	Registers	Reg./Address Space/Priv.
Resource Allocation	Pre-alloc	Pre-alloc	Pre-bind & Adaptive
Resource Exhaustion	Blocked	Blocked	Reserved for Reclaiming
Resource Accounting	Temporal	Temporal	Temp./Energy/Memory

Synchronous RPC or Asynchronous IPC. IPC typically assumes *symmetric endpoints* with the same execution model. Therefore, previous work suggests that asynchronous IPC can avoid serialization on multicore [30], allowing both endpoints to continue execution without blocking. However, in emerging scenarios, we observe that most IPCs are procedure calls, where the caller and callee can be clearly identified. Furthermore, OS services are mostly invoked passively rather than working continuously, and most subsequent operations of the application depend on the results of the procedure call. Therefore, synchronous Remote Procedure Call (RPC) is a more appropriate abstraction for service invocations.

HM adopts an RPC-like thread migration [33, 94] as the IPC fastpath for service invocations. When sending an IPC, the core kernel performs a direct switch (bypassing scheduling, similar to prior work [10, 30, 49, 67, 70]) and switches only the stack/instruction pointer (avoids switching other registers) as well as the protection domain. Specifically, *HM* requires OS services to register a handler function as the entry point and to prepare an execution stack pool. When an application invokes a service, the core kernel records the caller's stack/instruction pointer in an invocation stack and switches to the handler function with the prepared execution stack. On return, *HM* pops an entry from the invocation stack and switches to the caller. The IPC arguments are primarily passed through registers, with additional arguments passed through shared memory.

Performance gap. Although *HM* bypasses scheduling and avoids switching registers, it still faces non-trivial performance degradation due to privilege level/address space switching and cache/TLB pollution [9, 30, 49, 86] (accounts for 50% of total IPC cost). We further bridge this performance gap using differentiated isolation classes in section 4.2.

Resource Allocation. The *memory footprint of IPC has been largely neglected* by previous work. However, due to the extremely high IPC frequency and massive number of connections (>1k threads simultaneously) in emerging scenarios like smartphones, we find it essential to consider IPC's memory footprint in production, as it can cause severe problems such as out-of-memory (OOM) and even system hangs. Although each IPC connection in *HM* requires only an individual execution stack (rather than a full-fledged thread with all related data structures), its memory footprint is still non-trivial, given the massive amount of IPCs.

Previous work pre-allocates a thread/stack pool of a fixed size and binds it to connections. However, its size is hard to decide due to the diversity and dynamism of workloads, including the number of running threads and requirements for different OS services. A large pool would quickly drain the memory, while dynamic allocation on connection introduces runtime overhead on the critical path of IPC. We initially tried to prepare and bind stacks in each OS service for each thread on creation. However, we quickly realized that the problem still exists because some services are barely used by some threads (wasted), and there exist many IPC chains (to another OS service) that need another stack.

Therefore, *HM* strikes a sweet spot by *pre-binding stacks* in frequently-used OS services (e.g., process/memory manager and FS) for each thread while still maintaining a stack pool whose size is *adjusted adaptively* at runtime. When the remaining stacks fall below a threshold, the OS service will allocate more to reduce synchronous allocation. *HM* further reduces its memory footprint by reusing the same stack when calling the same service (e.g., ABA-like call) in an IPC chain.

Resource Exhaustion. IPC *can fail due to resource exhaustion*. Specifically, when the stack pool runs out while OOM occurs, OS services cannot allocate a new stack to process the IPC request. However, apps cannot handle such an error (not existing in a monolithic kernel). Therefore, such requests are queued (blocked) in SOTA microkernels, which may cause severe issues like circular wait and even system hangs.

An intuitive approach is to send another IPC to the memory manager to reclaim some memory synchronously. However, we find that under such a scenario (already OOM), the IPC to the memory manager is likely to fail again. Such a failure is likely to occur in emerging scenarios where workloads are non-deterministic and heavy loads occur frequently (e.g., opening multiple apps simultaneously).

HM mitigates this by reserving an individual stack pool. Once OOM occurs, the kernel will synchronously IPC to the memory manager using the pool for memory reclaim (repeatedly) until the user's IPC succeeds. Thus, applications' IPCs are guaranteed to be handled correctly.

Resource Accounting. IPC *assumes a different execution entity* when handling requests, thus attributing the consumed resource to OS services. However, since competing applications in emerging scenarios require a clear accounting of resources, the consumed resource should be precisely accounted to the caller app. Previous work achieves temporal isolation by inheriting the caller's scheduling context [70, 80]. However, emerging scenarios also require an accounting of both energy and memory consumption. Therefore, *HM* records the

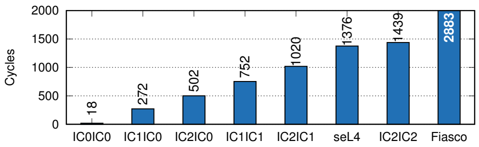


Figure 4: Round-trip IPC latency between ICx & ICy (`ICxICy`) in Raspberry Pi 4b. `IC0` includes the core kernel. `IC2` includes user apps. Zircon cannot run on Pi4b and is several times slower [49].

identity of the user app (root caller in the IPC chain), and attributes the consumed resources to it when handling IPC.

Decision: *Supplement async./sync. IPC with an RPC-like fastpath for invoking OS services while carefully addressing the resource allocation/exhaustion/accounting issues.*

4.2 Differentiated Isolation Classes

Isolation of OS Services. Placing all OS services in userspace may improve security, but it fails to meet performance requirements in emerging scenarios. We observe that *not all services require the same class of isolation*. In particular, mature, verified, and performance-critical OS services can be subjected to weaker isolation for optimal performance in practical deployments. Moreover, rapidly evolving services may frequently introduce bugs and vulnerabilities, thus requiring more robust isolation to prevent kernel corruption. OS services with large codebases and cumbersome features, such as drivers, require isolation to reduce the trusted computing base (TCB).

Therefore, *HM* adopts differentiated isolation classes (IC) to provide tailored isolation and performance for different OS services. Specifically, isolation classes *classify services and define the isolation between them*. Figure 4 shows the round-trip IPC latency between services at different isolation classes, compared to seL4 [67] and Fiasco.OC [69].

Isolation Class 0: Core TCB. IC0 applies to carefully verified, extremely performance-critical, trusted OS services, such as the ABI-compliant shim (the only IC0 service in deployment). No isolation is enforced between these services and the kernel. Therefore, IPCs are all indirect function calls.

IC0 Threat Model: IC0 is part of the core TCB, and any compromised IC0 services can arbitrarily read and modify others' memory. Therefore, placing services at IC0 should be carefully validated to avoid core kernel corruption.

Isolation Class 1: Mechanism-enforced Isolation. IC1 applies to performance-critical and validated OS services. Inspired by previous intra-kernel isolation approaches [11, 49, 59, 71, 112, 120], *HM* places these services in the kernel space and uses mechanisms to enforce isolation between services. Specifically, *HM* carefully divides the kernel address space into distinct domains and assigns each service a unique domain (IC0/core kernel also resides in a unique domain). *HM* uses ARM watchpoint [63] and Intel PKS [60] to prevent cross-domain memory access. Moreover, since IC1 services run in kernel space, they can execute privileged instructions. To prevent this, *HM* adopts binary-scanning and lightweight control-flow integrity (CFI, leveraging ARM pointer authentication (PA) [77]) to ensure services cannot execute illegal control flows that contain privileged instructions, and uses a secure monitor [49, 108] to guard the page table against code injection, which also traps any privileged instruction through *VM Exits* as a complement to CFI.

IPC between IC1 services (or to IC0) will enter a gate in the core kernel that performs a minimal context switch (switch instruction and stack pointers, w/o address space switching and scheduling) and configures the hardware to switch domains (take only a few cycles). Such a gate cannot be bypassed since domain switches require privileged instructions. Therefore, the IPC overhead is significantly reduced. As shown in Figure 4, it reduces the IPC latency between IC1 services by 50% compared to userspace services (IC2IC2).

IC1 Threat Model: IC1's threat model differs from other multi-server microkernels by assuming the correctness, soundness, and security of the applied isolation mechanism, which does expose some additional attack surfaces. For example, there are new attacks on ARM PA emerged recently [22]. Besides that, IC1 shares the same threat model, which prohibits any compromised service from reading/writing the core kernel's memory (and other OS services') and executing privileged instructions.

Isolation Class 2: Address Space Isolation. IC2 applies to non-performance-critical services or those containing third-party code (e.g., Linux drivers), enforced by address space and privilege isolation. IPC between IC2 services in *HM* (IC2IC2) is slightly slower than in seL4, mainly due to fine-grained locking, which is essential for scaling to multi-core processes under real-world loads.

IC2 Threat Model: IC2 shares exactly the same threat model as other multi-server microkernels.

Although IC1 significantly reduces the IPC overhead, it also introduces additional attack surfaces and has resource limitations (e.g., 16 domains in Intel PKS, 4 domains in ARM Watchpoint). Therefore, only performance-critical and validated OS services are placed at IC1. In addition, *HM* can quickly move all services back to IC2 if new attacks emerge. We further discuss deployment experiences on configuring isolation classes in section 4.3. Moreover, IC0/1 does not imply coupling to the kernel. The isolation classes allow for configurable isolation decisions during deployment rather than an isolation assumption during development. Different scenarios use different configurations, as shown in Figure 3.

Decision: *Not all OS services require the same class of isolation. Adopt differentiated isolation classes to relax isolation between trusted services for improved performance.*

4.3 Flexible Composition

Partitioning of OS Services. Although intuitively, OS services should be well-decoupled, e.g., FS and memory manager, we observe that *OS services are asymmetric* since some functionalities require close cooperation between specific services. For example, the FS is not the only entrance to accessing a file. POSIX supports file mapping that reads files through the memory manager, and it frequently appears on the critical path and significantly affects the user experience.

The isolation classes enforce the same isolation between same-class OS services. Therefore, without further structural support, *HM* still faces performance degradation compared to the monolithic kernel. First, the frequently invoked IPCs between tightly coupled services still cause noticeable overhead (20% in page fault handling for memory-mapped files) even in IC1 (kernel space). Moreover, double bookkeeping of shared states, such as page caches, introduces significant memory footprint and synchronization overhead. Finally, there is no global view of page caches to guide resource recycling (e.g., Least Recently Used, LRU).

To bridge the performance gap, *HM* adopts a configurable approach that allows coalescing tightly coupled OS services in performance-demanding scenarios, trading off isolation for better performance, while retaining the ability to separate them in safety-critical scenarios. When coalesced, no isolation is enforced, and IPCs between two services become function calls, while others remain as they are (well-isolated).

Coalescing also enables efficient co-management of page caches. Instead of maintaining them in both the FS and the memory manager, they can be co-managed when coalesced. It eliminates double bookkeeping and synchronization overhead and provides a global view for efficient recycling. To retain the ability to separate them, we provide a mechanism to automatically convert accesses of shared page caches into IPCs when separated. However, it will introduce non-trivial overhead. Therefore, in deployment, we implement both versions (sep./shr.) manually and enable them accordingly.

Performance. As shown in Table 3, when coalescing the FS with the memory manager, replacing the IPC reduces the latency of handling page faults caused by page cache misses by 20% (*Sep. Cache*). It can be further reduced by 30% (*Shr. Cache*) and achieves similar performance with Linux (5.10, detailed in section 6.2) by co-managing the shared page caches. Coalescing also speeds up the write throughput of *tmpfs* by 40%. Moreover, the memory footprint of coalesced services is reduced by 37% (FS+memory) in smartphones.

Security. The coalesced services are in a single failure domain, whose threat model (as a whole) remains the same as the isolation class in which it resides. Therefore, any failed or compromised service can only corrupt its coalesced services, which is also the primary compromise for performance. Hence, service coalescing should be carefully evaluated. In practice, due to the extremely high frequency of file operations in smartphones (Figure 1e), their performance targets can only be achieved by coalescing the FS with the memory manager. However, the security is still improved (isolated from other services) compared with monolithic kernels.

Deployment Experiences. Together with the differentiated isolation classes, *HM* enables flexible composition, allowing the key components to be assembled flexibly (user-space or kernel-space, separated or coalesced), enabling exploration of tradeoffs between isolation and performance according to scenarios' requirements, and the ability to scale from routers to smartphones with the same code base. The evolution of *HM* witnesses such explorations. Initially, all services were isolated at IC2. To meet the performance goal, we carefully assemble the system to retain most security properties by preserving the following rules.

Table 3: Performance improved by coalescing the FS service and the memory manager in the big core of Kirin9000 [57].

	Separated	**Coalesced**	**Linux**
Page Fault (Cycles)	7092	5290 (Sep. Cache) 3785 (Shr. Cache)	3432
Tmpfs Write (MB/s)	1492	2067	2133
Memory Footprint (MB)	190	120	N/A

Table 4: Address tokens support most operations of capabilities and allow direct access, except restricting fine-grained operation and chain revocation.

	Capabilities	**Address Tokens**
Token	CSlot id	Mapped Address
Access	Delegate to Kernel	Direct(RW)/`writev`(RO)
Ownership	Caps in CNode	Mapped Pages
Grant	Move to CNode	Map Page to VSpace
Revoke	Remove from CNode	Unmap Page
Chain Revoke	Support	No support
Security	Monitor all operations	Restrictions on mapped Obj.

First, due to the additional attack surfaces, IC1 services cannot contain any third-party code. Thus, although some drivers are also performance-critical, we kept them at IC2 and sped up via control/data plane separation (section 5.2). Second, service coalescing, especially with the memory manager, undeniably weakens isolation and security (though still improved compared with monolithic kernels). Therefore, we leave it configurable and only enable it on phones. Moreover, IC0 not only increases the core TCB but also has strict memory limitations and non-blocking requirements. Thus, in practice, *HM* only places the ABI shim (which can be opted out) in IC0. Section 6.1 details the configurations.

Decision: *OS services are asymmetric. Coalesce tightly coupled OS services and flexibly assemble the system to meet diverse requirements in various scenarios.*

4.4 Address Token-based Access Control

SOTA microkernels make all kernel objects explicit and subject to capability-based access control [30] to preserve the principle of least privileged, which is primarily implemented in a partitioned fashion that keeps a token (typically a slot ID) in userspace representing the permission to access a kernel object. However, we encountered severe performance issues when deploying it in emerging scenarios.

Clear relationship but slow access. Although capabilities are effective in describing the *external relationships* of kernel objects, i.e., the authorization chain, *accessing their internal* *contents* requires sending the token with the operations to the core kernel, which introduces non-trivial performance overhead due to privilege switches and accesses to multiple metadata tables. Kernel objects are hidden behind the capabilities and are only accessible by the core kernel. However, due to the minimality principle, the content of some kernel objects (e.g., page tables) should be frequently updated by OS services outside the core kernel, for which partitioned capabilities are no longer efficient.

Some microkernels speed up access by mapping specific objects to userspace. However, they can only be applied to few limited objects (e.g., memory objects [30, 67], part of the thread control block [3, 9, 76], and kernel interface page [76]) for security and lack the ability to synchronize data correctly, which inhibits the cooperation between the kernel and OS services. To address these issues, *HM* proposes a generalized address token-based approach that can be applied to a broader range of objects, enabling efficient co-management.

Specifically, as shown in Figure 5, each kernel object is placed on a unique physical page in *HM*. Granting a kernel object to an OS service requires mapping such a page to its address space (❶). Thus, the mapped address serves as the token to access the kernel object directly from the hardware without involving the kernel (unwillingly). Kernel objects can be granted (mapped) as read-only (RO) or read-write (RW). OS services can read RO kernel objects without kernel involvement. To update them, a new syscall, `writev`, should be used, passing the target address with the updated value, and the core kernel will verify permissions by referring to the kernel object's metadata (❷). For RW kernel objects, once granted, can be updated by OS services without kernel involvement (❸). Moreover, for objects smaller than a page with the same property (permission) and a similar life cycle, *HM* batches these objects into a single page upon allocation, allowing them to be granted and revoked collectively.

Functionality. Address tokens support most operations of capabilities, as shown in Table 4 (compared to seL4 [107], Zircon has similar functionality [37]), with two exceptions. First, address tokens cannot restrict fine-grained operations once granted, which weakens security and exposes additional attack surfaces. Besides, capabilities store the detailed relationship, allowing chain revocation, which address tokens do not support due to implicit ownership. Nevertheless, address tokens are only used by selected co-managed kernel objects. The attack surfaces are carefully mitigated (discuss below). Moreover, due to the centralized resource management, kernel objects have specific owners (will not be granted to others). Thus, chain revocation is rarely used.

Security. Once an address token is granted to an OS service, the kernel cannot monitor the subsequent operations. *HM* mitigates this by restricting the objects mapped to userspace (enforced by static analysis). Only kernel objects that exclusively contain the values of certain variables in kernel-preserved structures (pointers are not allowed to prevent the time-to-check to time-to-use attack) are mapped RW (e.g., PCache in section 4.5), ensuring they will not corrupt the kernel with incorrect or inconsistent data. The rest of the kernel's internal states (e.g., pointers and reference counters) can only be mapped RO or not granted at all to prevent it from being corrupted. *HM* further applies a sanity check when reading from RW objects. It does introduce some attack surfaces by leaking kernel-internal information, which can be mitigated by hardware encryption like ARM PA.

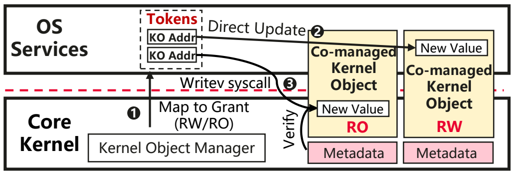


Figure 5: Address token-based access control in *HM*. ❶ Map kernel object's page to grant. ❷ Direct access to RW objects. ❸ Use `writev` to update RO objects, verified by the kernel.

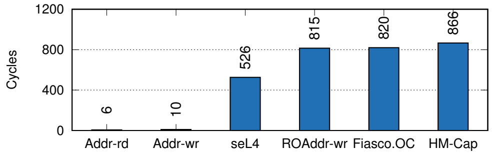


Figure 6: Latency of accessing kernel objects on Raspberry Pi 4b. *Addr-rd/wr* represent address tokens in *HM*. *ROAddr-wr* represents writing to read-only objects in *HM*.

Synchronization. There are two approaches to sharing data between OS services and the kernel leveraging address tokens. First, OS services and the kernel can exchange messages asynchronously (message-passing). For example, PCache in section 4.5 sends pre-allocated pages to the kernel for future kernel paging. *HM* uses a lock-free ring buffer to synchronize the data correctly. Besides, OS services can apply in-place updates to the objects (e.g., VSpace in section 4.5, which stores the memory layout) that the kernel may read concurrently. *HM* adopts fine-grained locking to ensure correctness. However, it may block the kernel when the service is preempted while executing critical sections. Therefore, the kernel can only use the `trylock` operation on RW-mapped objects. If it fails, *HM* will redirect to the OS services (slow path) to finish the procedure (e.g., paging in section 4.5).

Performance. Figure 6 compares the latency of accessing kernel objects after applying address tokens. The reading and writing (to RW) latencies are significantly reduced compared with capability-based approaches. However, the latency of writing RO objects is slower than seL4 on RPi4b, mainly due to the use of `AT` instruction on ARM to translate the address and check the permissions, which is slow on RPi4b (yet optimized in the advanced smartphone chips).

Usage Scenario. For security concerns (see above), address tokens are OS-internal abstractions that supplement the capabilities for efficient co-management with OS services. Specifically, besides enabling direct updates to kernel objects managed by services, it allows them to read internal states (e.g., poll list in section 5.1) without kernel involvement, sim-

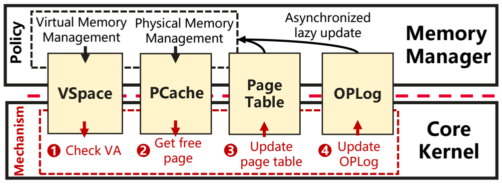


Figure 7: Policy-free kernel paging in *HM*. On page fault, the kernel checks the address ❶ and, if anonymous, ❷/❸ maps a pre-allocated page, and ❹ records an OPLog.

ilar to virtual dynamic shared objects (vDSO) in Linux [82]. It also allows handling performance-critical events (e.g., page faults in section 4.5) entirely in the kernel without violating the minimality principle by making policy-driven decisions in advance (by services) and communicating with the core kernel through co-managed objects.

Decision: *Capabilities that hide kernel objects behind interpose kernel (unwillingly) on the data plane. Supplement with address tokens for efficient co-management.*

4.5 Policy-free Kernel Paging

Centralized Management vs. Distributed Pager. Some SOTA microkernels (e.g., seL4) delegate memory management to applications with individual custom pagers. However, since competing applications in emerging scenarios require coordinated and centralized management, we found it difficult to implement certain features efficiently that require a global view of memory with decentralized pagers, such as the control group (cgroup) and memory recycling. Therefore, *HM* manages the memory through a centralized memory manager. For minimality, the memory manager is outside the core kernel, which manages the physical and virtual memory and handles page faults for all applications and OS services.

Slow userspace paging. We observe a significant performance degradation in performance-critical scenarios (e.g., app startup in Figure 2) due to the slow paging procedure of anonymous memory (e.g., stack/heap), which occurs frequently in smartphones, as shown in Figure 1b. The degradation is primarily due to the extra round-trip from the kernel to the pager. Specifically, after throwing a page fault exception, the kernel issues an IPC to the pager, which checks the address and assigns a new page, then returns to the kernel to update the page table before finally returning to the application. Such a round-trip is inevitable because page fault handling involves a policy of deciding whether and which physical page to map, which should be kept out of the kernel [27], while the exceptions are handled inside the kernel, and the page table is hidden behind a capability in the kernel.

To improve the performance of handling page faults of anonymous memory, *HM* makes policy-driven decisions in advance, and leaves a policy-free page fault handling mechanism in the core kernel. Thus, it eliminates the extra IPC round-trip on the critical path. Specifically, the memory manager provides the address range of anonymous memory along with several pre-allocated physical pages. As shown in Figure 7, if the page fault is triggered within the range (❶), the core kernel can map it directly to a pre-allocated physical page (❷ and ❸), and record an operation log (OPLog, ❹), which the memory manager will use to asynchronously update its internal states (e.g., the counter of the mapped anonymous pages). Otherwise, if the address is outside the specified range (not performance-critical) or the pre-allocated pages are exhausted, the core kernel will make an IPC to the memory manager. The involved kernel objects are co-managed by the memory manager via address tokens, including the page table, the operation log, the VSpace, which records the layout of virtual memory space for identifying anonymous memory, and the PCache, which stores the pre-allocated pages.

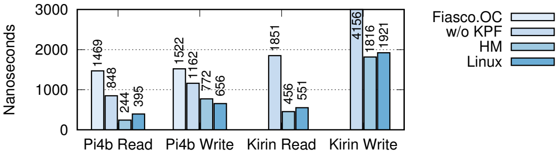


Figure 8: Page fault latency of private anonymous memory. Read is optimized with zero page. seL4 is not included since it does not support demand paging by default.

Compromises. By making policy-driven decisions in advance, the policies (whether/which to map) are still kept outside the core kernel. The only compromised ability is to change the policy after being pre-allocated to the PCache, which reduces flexibility. PCache also introduces some additional memory footprints. However, since PCache can be periodically replenished (off the critical path), its size remains relatively small, making these tradeoffs acceptable.

Performance. Figure 8 shows the reduced latency of kernel paging (KPF) in *HM*. *HM* reduces read/write latency by 72%/33% on Pi4b and 75%/55% on Kirin9000 (little core), making it even slightly shorter than Linux (6.1 on Pi4b and 5.10 on Kirin9000). seL4 is not included since it requires a custom pager and does not support demand paging by default. The round-trip (to the pager) of fault handling takes about 140ns on Pi4b (measured using sel4bench), which makes it significantly slower than Linux.

Decision: *Enable policy-free kernel paging by preempting policy-driven decisions.*

5 Compatibility Design of HongMeng

5.1 Linux ABI Compatibility

Deploying in emerging scenarios requires Linux ABI compatibility, which poses challenges in multi-server microkernels. K42 [68] achieves Linux ABI compatibility through trap reflection, which redirects syscalls back to the k42 library loaded into the application's address space. However, it introduces significant performance overhead due to the additional roundtrip to the kernel [2] and also faces implementation challenges [29, 111] due to keeping states in userspace. FreeBSD [36] and Windows (WSL1 [87]) also achieve partial Linux ABI compatibility through syscall emulation. However, since all their OS services reside in kernel space, the emulation layer can map abstractions like fd directly to their internal states and efficiently support functions like `fork` and `poll`, which is challenging in multi-server microkernels.

Syscall Redirection. *HM* achieves Linux ABI compatibility by placing an ABI-compliant shim in IC0 (kernel space), which redirects Linux syscalls into IPCs towards appropriate OS services (identified by syscall number, with native syscalls bypassing the shim), as illustrated in Figure 3a. In addition, the shim is optional. In scenarios where applications are predominantly custom, *HM* replaces the shim with POSIX-compliant libraries, as shown in Figure 3b.

Centralized States. Apart from binary compatibility, microkernels no longer have a central repository for global states, such as the file descriptor (fd) table, making functions like fd multiplexing (i.e., polling) and syscalls like `fork`[2] challenging to implement. Specifically, the fd table is usually kept in the application's address space (only contains credentials verified by OS services). Thus, fd multiplexing requires mapping all waiting fd to a notification primitive and sending it to all related services. Moreover, syscalls like `fork` have to correctly assemble such distributed states in the userspace. It introduces significant complexity and performance overhead, primarily due to passing states from parent to child and the additional page faults caused by updating these copy-on-write states [8, 29]. Therefore, SOTA microkernels, including seL4, Fiasco, and Zircon, do not support `fork`, while `fork` in K42 is known to have severe performance issues [29, 111].

Therefore, the ABI-compliant shim in *HM* also serves as a central repository for global states like the fd table, enabling efficient implementation of both fd multiplexing like `poll` and syscalls like `fork`. Specifically, the shim maintains the fd table, which maps fd to credentials (used by OS services to identify the user). Therefore, implementing `poll` only requires maintaining a poll list within the shim, co-managed with OS services via address tokens. It also avoids copying the fd table in userspace when executing `fork`.

Vectored Syscalls. Although most of the syscall translations are achieved solely in the ABI-compliant shim, there are vectored syscalls [116] (e.g., `ioctl/fcntl`) that extend system APIs and allow custom extensions (for drivers/modules) via the file abstraction. *HM* redirects and handles them in the FS service (e.g., invoking driver containers in section 5.2).

Deployment Experiences. *HM* passes all the tests in the AOSP compatibility and vendor test suite (CTS/VTS [43, 44]), which examines both the kernel functionalities and driver behavior. Although most binaries can run out of the box, we observe that some apps rely on unstable/undocumented Linux behavior and fail to run on *HM*. For example, an application that depends on a specific `epoll` return order [95] fails to run on *HM* (it also fails with different Linux versions).

Decision: *Achieve Linux binary compatibility through ABI-compliant shim.*

5.2 Driver Container

Linux undeniably has the richest device driver ecosystem. Further, some drivers are not source-available, which makes porting challenging. Therefore, reusing Linux drivers is essential for widespread deployment.

Challenging practical and performant driver reuse. Previous work, including both transplantation [3, 17, 32, 41, 118] and VM-based methods [72], face challenges in *achieving high compatibility, reasonable engineering effort, and uncompromised performance simultaneously*. In particular, transplanting the runtime environment requires re-implementing all kernel APIs (KAPIs) used by drivers. Since some drivers use a large number of KAPIs, some of which are even constantly evolving, this approach faces challenges of compatibility and affordable engineering effort. In addition, reused drivers (with large untrusted code base) should be enforced with strict address space isolation for better security and to avoid license contamination [34], which also degrades performance. Reusing drivers through a virtual machine can achieve better compatibility with less human effort. However, it introduces issues including memory double management that causes extra memory footprint (crucial in memory-constrained scenarios like smartphones) and thread double scheduling that degrades performance due to the frequent use of asynchronous notifications in drivers.

HM reuses Linux drivers (Figure 9) through a driver container, which strikes to find a sweet spot between compatibility, engineering effort, and critical-path performance.

Compatibility. Inspired by LKL [101], UML [26], and SawMill [39], the Linux Driver Container (LDC) provides all necessary Linux KAPIs by reusing the Linux code base as a userspace runtime, allowing existing Linux drivers to run without modification. The main difference with LKL/UML/SawMill is that LDC reuses the driver rather than components like the file system and network stack. Thus, drivers should be able to access the hardware devices directly rather than redirecting to host drivers. Further, the runtime relies on *HM* for resource management. Therefore, all related functionalities, like the thread scheduler, are removed.

HM creates another device manager that manages both the Linux and the native driver containers (where the native drivers reside). Besides initializing driver containers, it registers entries (❶ in Figure 9) in the virtual file system (VFS) so that driver invocations through VFS (e.g., `ioctl` ❷) can be correctly redirected to the appropriate driver container (❸).

Using the LDC, *HM* has successfully reused over 700 device drivers from Linux, including all the needed ones for

[2] While there have been arguments that `fork` should be deprecated [8], popular frameworks like AOSP/OpenHarmony still use `fork`.

smartphones and vehicles to function correctly, such as camera, display, audio, NPU/GPU, and storage. Though most drivers can directly run out of the box, several exceptions exist. Since the LDC runs in userspace (IC2), drivers that use privileged instructions (e.g., `smc`) will trigger faults. They require binary rewrites or manual porting (for those frequently using privileged instructions, e.g., GIC) in the core kernel.

Engineering Effort. The Linux runtime in the LDC is derived directly from mainline Linux, with minor modifications to redirect several functionalities to the driver container base (DC-base in Figure 9) for proper execution. Therefore, the required engineering effort is minor. Specifically, we provide a virtual architecture and redirect the kthread/memory interfaces to *HM*. To make the drivers work correctly, DC-base creates a virtual timer and a virtual IRQ chip to provide the interrupt request and reserves a linear mapped space for functions like `virt_to_phys`. Compared to the VM-based method, which introduces double memory management and double thread scheduling, the driver container avoids these issues by redirecting and managing them in *HM*.

In practice, supporting long-term support (LTS) kernel distributions is sufficient for reusing most drivers (currently, *HM* supports 4.4, 4.19, and 5.10). In addition, since the Linux interfaces associated with the DC-base are relatively stable, only minor modifications are required to upgrade the Linux runtime. Upgrading from 4.19 to 5.10 requires less than 100 changes to the DC-base, most of which are minor modifications to the procedure names, arguments, and structures.

Critical Path Performance. The LDC is placed in IC2 (userspace) to preserve security (drivers have large untrusted code bases) and avoid license contamination. However, it introduces non-trivial overhead in driver-critical scenarios, such as app startup and camera. Therefore, *HM* applies control plane and data plane separation by creating a twin driver in the native driver container that handles I/O IRQs on the performance critical path (❹ in Figure 9). The twin driver rewrites the data handling procedure and can thus be enforced with weaker isolation (placed at IC1 in kernel space), resulting in significantly better performance. The control planes, which contain cumbersome procedures like init/suspend/resume, remain in the LDC (❺).

The twinned drivers synchronize the states (usually a variable) via IPC. Since the control plane is handled entirely in the LDC, the twin driver does not modify the states (I/O errors are redirected to the LDC). On initialization, the LDC passes device information to the native one to create the twin driver. When handling non-I/O IRQs and errors, the LDC synchronizes the updated states back to the twin driver. Unlike the transparent integration solely in LDC, which results in poor performance, the twin driver requires additional engineering effort to split and redirect interrupts and synchronize states. Therefore, the twin driver is used only for performance-critical drivers like the Universal Flash Storage (UFS) driver (others are integrated transparently w/o modification).

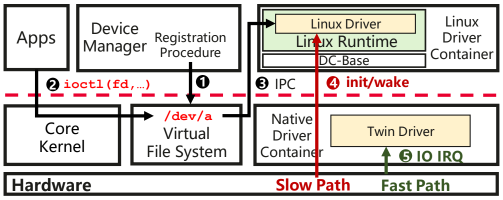
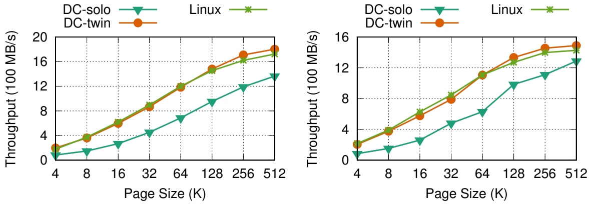


Figure 9: Drivers in *HM*. The device manager creates file nodes in the VFS ❶. VFS redirects invocations ❷ to drivers ❸. *HM* improves performance by separating the control ❹/data ❺ plane.



(a) BIO read.

(b) BIO write.

Figure 10: Block I/O throughput on Kirin9000. DC-twin applies data and control plane separation, while DC-solo does not.

Figure 10 shows the improved throughput in the UFS Block I/O benchmark. In the experiment, I/O requests are issued directly from the driver. DC-twin (applied data/control plane separation) achieves a similar throughput to Linux 5.10 and outperforms DC-solo (w/o separation) by 140% at 4K size.

Security. The LDC is almost a normal userspace (IC2) OS service in *HM*, with an additional ability to create a linear mapped space whose range is strictly restricted to its allocated memory (by only setting the present bit on the allocated pages). Thus, it shares the same threat model as IC2 OS services and userspace drivers in other microkernels. In addition, *HM* uses SMMU [5] to prevent DMA attacks, with its driver residing in an isolated native driver container.

Decision: *Reuse Linux device drivers efficiently through driver containers with control/data plane separation.*

6 HongMeng in the Wild

6.1 Implementation and Deployment

The core kernel of *HM* is implemented primarily in a confined subset of C, consisting of 90k lines of code (LoC), which includes the basic functionalities. All other OS services are decoupled and can be deployed individually, totaling over 1 million LoC. The *HM*'s build system can assemble the OS services based on detailed configurations specified for various scenarios, such as placing OS services in different isolation classes or coalescing some OS services.

HM has been deployed in tens of millions of devices in various emerging scenarios, which share the same code base but with different configurations. In safety-critical scenarios, such as smart vehicles (dashboard and entertainment system) and the trusted execution environment (TEE) of smartphones, security and strict isolation are prioritized over performance. In addition, applications are mostly customized

Table 5: LMbench results.

Benchmark Commands[1]	**Unit**	**Linux**	***HM***	**Norm.**[2]
`lat_unix -P 1`	μs	10.23	10.39	0.98
`lat_tcp -m 16`	μs	21.22	17.19	1.23
`lat_tcp -m 16K`	μs	24.54	18.9	1.29
`lat_tcp -m 1K (Same Core)`	μs	21.21	17.19	1.23
`lat_tcp -m 1K (Cross core)`	μs	37.96	25.66	1.47
`lat_udp -m 16`	μs	17.83	19.48	0.92
`lat_udp -m 16K`	μs	23.63	22.02	1.07
`lat_udp -m 1K (Same Core)`	μs	18.04	19.55	0.92
`lat_udp -m 1K (Cross core)`	μs	34.17	26.84	1.27
`bw_tcp -m 10M`	MB/s	1812	3109	1.71
`bw_unix`	MB/s	7124	8478	1.19
`bw_mem 256m bcopy`	MB/s	17696	17202	1.02
`bw_mem 512m frd`	MB/s	14514	14593	0.99
`bw_mem 256m fcp`	MB/s	17492	15867	0.91
`bw_mem 512m fwr`	MB/s	34771	35318	1.01
`bw_file_rd 512M io_only`	MB/s	8976	9396	1.04
`bw_mmap_rd 512M mmap_only`	MB/s	26073	27520	1.05
`lat_mmap 512m`	μs	3315	3628	0.91
`lat_pagefault`	μs	0.83	0.78	1.06
`lat_ctx -s 16 8`	μs	4.53	3.41	1.32
`bw_pipe`	MB/s	3808	4127	1.08
`lat_pipe`	μs	9.00	7.88	1.14
`lat_proc exec`	μs	336	1305	0.26
`lat_proc fork`	μs	323	1280	0.25
`lat_proc shell`	μs	2269	4778	0.47
`lat_clone (create thread)`	μs	28.6	54.3	0.52

[1] Argument "`-P 1`" is omitted.

[2] *Norm.* shows the normalized performance. For throughput, use *HM*/Linux, for latency, use Linux/*HM*. The more the better.

and source-available. Therefore, *HM* places all OS services in IC2 (userspace) and exposes the POSIX API to applications through libraries. Moreover, *HM* achieves fault tolerance by introducing a driver micro-reboot in the TEE. Drivers in the TEE can be considered stateless since only re-initialization is required to recover a corrupted driver. With micro-reboot, the TEE can recover from driver corruption within hundreds of milliseconds, whereas a complete system reboot is required with a monolithic kernel. Fault tolerance for a broader range of scenarios (e.g., stateful OS services in rich-OS) requires additional efforts to store states and preserve their consistency, which we leave for future work.

In performance-demanding scenarios like smartphones, *HM* places the performance-critical OS services in IC1 (kernel space), including the process manager, memory manager, FS, and native driver container, and coalesces FS with the memory manager. The Linux driver container and other non-performance-critical OS services, such as CPU frequency governor and power manager, remain in IC2 (userspace).

6.2 Performance

We present the end-to-end performance comparison between *HM* and Linux in emerging scenarios, including smartphones (using Kirin9000 SoC [57]), smart vehicles, and smart routers, which existing microkernels fail to support. The compared Linux 5.10 counterparts are already highly optimized (used in prior products) rather than vanilla versions.

LMbench. We evaluate the basic OS functionalities using LMbench [85] on Kirin9000. Table 5 shows the results related to OS architecture. Compared to Linux (5.10), the context switching `lat_ctx` (32%) and networking (avg. 21%) are faster on *HM*, mainly due to the simplified handling procedure compared to Linux [89,96]. Memory operations perform similarly to Linux. Although `fork` still performs worse than Linux in the microbenchmark, we observe that the major overhead of `fork` in the real-world load comes from copying virtual memory areas (VMAs). It can be accelerated through parallelism, which reduces its overhead from 150ms to 60ms (in typical apps, close to 30ms in Linux). `Clone` (creating thread) is also 1x slower than Linux, mainly due to the additional IPCs between multiple OS services (especially the driver container in IC2) and the core kernel.

Geekbench. Figure 11c presents the normalized single-core results of the CPU-intensive Geekbench 5.3.2 [99]. By assembling the system to prioritize performance, *HM* achieves similar performance with Linux, with minor differences due to the different CPU frequency altering strategies.

Application Cold Startup Time. App startup time is critical to the user experience, stressing multiple OS services (e.g., reading from flash memory and creating threads) with extensive IPCs. Figure 11a shows the startup time of the top 30 AOSP apps on *HM*. The framework/app versions are the same on Linux and *HM*. As analyzed in section 3.1, the major overhead of microkernel in such scenarios comes from state double bookkeeping and slow paging, which *HM* eliminates. Therefore, the startup time is even 17% shorter (geometric mean) than Linux, mainly due to the lighter loads (see below) and the custom scheduling strategies.

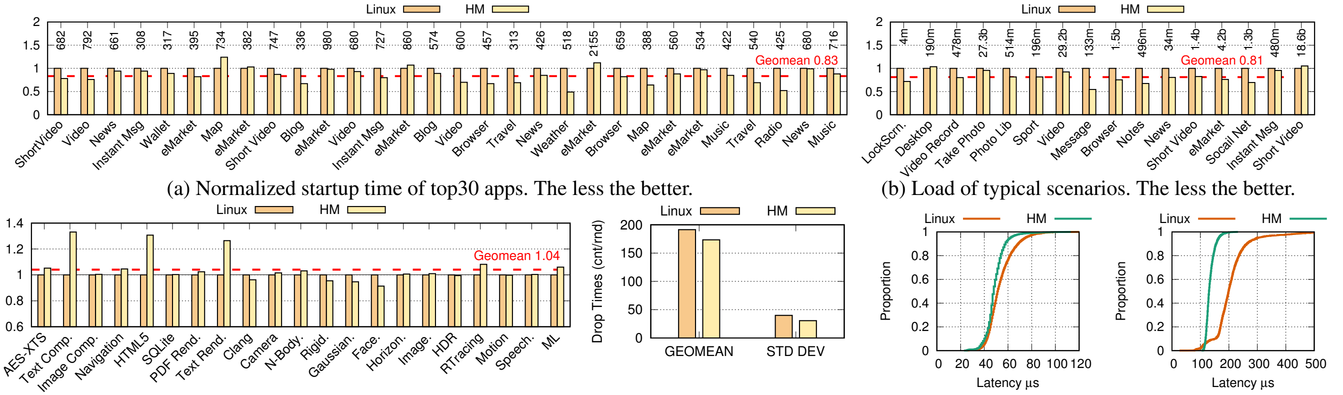
Application Loads. Figure 11b presents the loads in a period in different scenarios. The loads (number of executed instructions) are collected using `perf`, which includes the executed instructions in OS services (or in Linux kernel). The load on *HM* is 19% lighter (geometric mean) than on Linux. The proposed techniques in *HM* significantly reduce the overhead of minimality and fine-grained access control. Lighter loads also enable *HM* to achieve better performance and energy efficiency than Linux.

We further present improvements using custom strategies in *HM*, which are challenging to apply in Linux (section 2.3).

Frame Drops. Figure 11d shows the frame drop times (crucial for user experience) in 20 rounds of running the typical usage model in section 3.1. Due to the lighter load and a *custom QoS-guided scheduling* in *HM*, frame drops are 10% less and 20% stabler than Linux.

Interrupt Latencies. Figure 11e and 11f show the latency CDF of the related interrupts when playing video and audio, which are essential for user experience. *HM* reduces their latencies by 10% (video) and 65% (audio) by using a *custom experience-first strategy* that executes all the handling procedures at once, which is handled in another additional interrupt (due to lazy disable of ARM GIC [40]) in Linux.

Experiences in smart routers and smart vehicles. In smart routers, *HM* reduces OS memory footprint by 30%, allowing 30% more client connections. In smart vehicles, *HM* reduces system cold boot time from 1.5s (Linux) to 0.6s (critical for user experience, e.g., enabling 360-degree surround view) and reduces cross-domain (dashboard and entertain-



(a) Normalized startup time of top30 apps. The less the better.

(b) Load of typical scenarios. The less the better.

(c) Geekbench (single core). The more the better.

(d) Frame drops. The less the better.

(e) Video int. latency CDF.

(f) Audio int. latency CDF.

Figure 11: Performance of *HM* compared with optimized Linux 5.10 on Kirin9000. (a), (b) and (c) normalized the result for comparison. Labels in (a) show the startup time in milliseconds on *HM*. Labels in (b) show the executed instructions on *HM*.

ment) communication latency from 250μs to 100μs.

7 Lessons and Experiences

Compatible first, then nativize gradually. Compatibility is a crucial first step for commercial deployment. First, a product typically prefers a unified code base for various platforms for cost-efficiency. Second, some third-party apps/drivers are distributed in binaries. Moreover, even aiming to rebuild a new software ecosystem, many essential libraries still require Linux compatibility. Therefore, only by being compatible at first can a new OS be widely deployed and have a chance to evolve towards native interfaces for improved performance.

Specification alone is insufficient. Examine compatibility via large-scale testing. Achieving full compatibility is difficult (if possible), primarily due to Hyrum's Law [121], which reveals that all observable system behaviors will be depended on. Therefore, rather than satisfying certain specifications, we examine compatibility through large-scale testing, which is necessary to uncover hidden compatibility issues.

Deploy first, then optimize continuously. A microkernel is hard to satisfy all performance goals initially and requires full-system optimizations (e.g., framework, even hardware). Without deployment, promoting cooperation among multiple teams for such optimizations is difficult. Moreover, production deployments require time to test reliability. Therefore, deployments should commence early, starting on a small scale.

Use automated verification as much as possible. We found that complete formal verification (using interactive theorem proving) is unsustainable due to the rapid growth in code size and functionalities. Hence, we resort to semi-formal verification of critical components and use automated verification and verification-guided testing to enhance the code quality.

Amplification of hardware failures/bugs due to the scale effects. We found that some low-probability hardware faults or bugs are relatively likely to occur when deployed at scale, significantly affecting user experience and potentially becoming fatal in safety-critical scenarios. *HM* mitigates these issues by isolating critical drivers in different LDCs, restarting stateless drivers in TEE, and creating watchdogs for monitoring. *HM*, as a microkernel, also provides opportunities to address these issues through architectural design in future work.

Big kernel lock is not scalable in emerging scenarios. While it is argued that a big kernel lock is sufficiently scalable for a microkernel [97], mainly due to the short duration of most syscalls, we found that it still faces scalability issues on phones. First, phones have a high syscall frequency (61k/s, Figure 1f), causing *significant contentions*. Moreover, emerging scenarios demand some *complex functionality with long durations*. Examples include `poll`, which requires synchronizing a large number of states within the shim (IC0), and energy-aware scheduling [115], which involves frequent and costly calculations of power consumption for each scheduling decision due to the short-running nature of threads on phones.

8 Conclusion and Future Work

HongMeng is a commercialized general-purpose microkernel that retains microkernel principles while providing structural supports to address compatibility and performance challenges in emerging scenarios. It also facilitates future exploration of microkernels' benefits in production. For instance, its flexibility offers opportunities to accommodate the increasing hardware heterogeneity that Linux fails to address [104], and to achieve fault tolerance for improving availability.

Acknowledgements

We thank our shepherd, Timothy Roscoe, and the anonymous reviewers for the insightful comments. We also thank colleagues in the Huawei OS Kernel Lab and Shanghai Jiao Tong University for their helpful work and support. Haibo Chen is also partly supported by National Natural Science Foundation of China (No. 61925206 and 62132014).

References

[1] Real time Linux. https://wiki.linuxfoundation.org/realtime/start. Accessed 16 April 2024.

[2] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with K42, an Open-Source, Linux-Compatible, Scalable Operating-System Kernel. *IBM Syst. J.*, 44(2):427–440, jan 2005.

[3] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, Technical report, IBM Watson Research, 2002.

[4] Apple. XNU Project. https://github.com/apple-oss-distributions/xnu. Accessed 16 April 2024.

[5] ARM. System MMU Support. https://developer.arm.com/Architectures/System%20MMU%20Support. Accessed 16 April 2024.

[6] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[7] AUTOSAR. Adaptive Platform. https://www.autosar.org/standards/adaptive-platform. Accessed 16 April 2024.

[8] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A Fork() in the Road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 14–22, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[10] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 102–113, New York, NY, USA, 1989. Association for Computing Machinery.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995.

[12] Simon Biggs, Damon Lee, and Gernot Heiser. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] BlackBerry. BlackBerry QNX: Real-Time OS and Software for Embedded Systems. https://blackberry.qnx.com/en. Accessed 16 April 2024.

[14] BlackBerry. Meet the Power Behind the BlackBerry Tablet OS. https://www.qnx.com/company/announcements/blackberry_tablet_os.html. Accessed 16 April 2024.

[15] BlackBerry. BlackBerry OS End of Life. https://www.blackberry.com/us/en/support/devices/end-of-life, 2020.

[16] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1–19. USENIX Association, 2020.

[17] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[18] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces. In Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, pages 166–173. ACM, 2023.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou, editors, *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 5. ACM, 2011.

[20] Yonghun Choi, Seonghoon Park, and Hojung Cha. Graphics-Aware Power Governing for Mobile Devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 469–481, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Yonghun Choi, Seonghoon Park, Seunghyeok Jeon, Rhan Ha, and Hojung Cha. Optimizing Energy Consumption of Mobile Games. *IEEE Transactions on Mobile Computing*, 21(10):3744–3756, 2022.

[22] CVE. CVE-2021-30769. https://nvd.nist.gov/vuln/detail/CVE-2021-30769. Accessed 16 April 2024.

[23] CVE. CVE Records. https://cve.mitre.org/. Accessed 16 April 2024.

[24] CVEdetails. Linux Kernel 2.6 Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/version_id-410986/Linux-Linux-Kernel-2.6.html. Accessed 16 April 2024.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Jeff Dike. User-mode Linux. In *5th Annual Linux Showcase & Conference (ALS 01)*, Oakland, CA, November 2001. USENIX Association.

[27] Björn Döbel. Memory, IPC, and L4Re. https://os.inf.tu-dresden.de/~doebel/downloads/02-MemoryAndIPC.pdf, 2012.

[28] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 671–684. ACM, 2019.

[29] David Edelsohn. Providing a Linux API on the Scalable K42 Kernel. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[30] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 133–150. ACM, 2013.

[31] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 251–266. ACM, 1995.

[32] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.

[33] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to A Migrating Thread Model. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*, San Francisco, CA, January 1994. USENIX Association.

[34] Free Software Foundation. Frequently Asked Questions about the GNU Licenses. https://www.gnu.org/licenses/gpl-faq.en.html#MereAggregation. Accessed 16 April 2024.

[35] OpenAtom Foundation. OpenHarmony Project. https://gitee.com/openharmony/docs/blob/master/en/OpenHarmony-Overview.md. Accessed 16 April 2024.

[36] FreeBSD. Linux emulation in FreeBSD. https://docs.freebsd.org/en/articles/linux-emulation/. Accessed 16 April 2024.

[37] Google Fuchsia. Zircon Handles. https://fuchsia.dev/fuchsia-src/concepts/kernel/handles. Accessed 16 April 2024.

[38] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 168–178. ACM, 2008.

[39] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European*

Workshop: Beyond the PC: New Challenges for the Operating System, EW 9, page 109–114, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Thomas Gleixner and Ingo Molnar. Linux generic IRQ handling. https://www.kernel.org/doc/html/v4.18/core-api/genericirq.html, 2010.

[41] Shantanu Goel and Dan Duchamp. Linux Device Driver Emulation in Mach. In *USENIX Annual Technical Conference*, pages 65–74, 1996.

[42] Google. Android Open Source Project. https://source.android.com/. Accessed 16 April 2024.

[43] Google. AOSP Compatibility Test Suite. https://source.android.com/docs/compatibility/cts. Accessed 16 April 2024.

[44] Google. AOSP Vendor Test Suite (VTS) and infrastructure. https://source.android.com/docs/core/tests/vts. Accessed 16 April 2024.

[45] Google. App startup time. https://developer.android.com/topic/performance/vitals/launch-time. Accessed 16 April 2024.

[46] Google. Fuchsia Zircon Kernel. https://fuchsia.dev/fuchsia-src/concepts/kernel?hl=en. Accessed 16 April 2024.

[47] Google. Fuchsia's libc. https://fuchsia.dev/fuchsia-src/development/languages/c-cpp/libc?hl=en. Accessed 16 April 2024.

[48] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 119–134, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 401–417. USENIX Association, 2020.

[50] Nikolai Hampton. The working dead: The security risks of outdated Linux kernels. https://www2.computerworld.com.au/article/615338/working-dead-security-risk-dated-linux-kernels/, 2017.

[51] Nikolai Hampton and Patryk Szewczyk. A survey and method for analysing soho router firmware currency. 2015.

[52] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In Chandramohan A. Thekkath, Ramakrishna Kotla, and Lidong Zhou, editors, *Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems, ApSys 2010, New Delhi, India, August 30, 2010*, pages 19–24. ACM, 2010.

[53] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, jul 2006.

[54] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[55] Hans J Holberg and Udo Brockmeyer. ISO 26262 compliant verification of functional requirements in the model-based software development process. In *White paper. Embedded World Exhibition and Conference*, 2011.

[56] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating Device Driver Isolation. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 613–631. USENIX Association, 2022.

[57] Huawei. Kirin9000. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000. Accessed 16 April 2024.

[58] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 588–604. ACM, 2021.

[59] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, apr 2007.

[60] R Intel. Architecture instruction set extensions and future features programming reference. https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf, 2021.

[61] ISO. Road vehicles Functional safety. https://www.iso.org/obp/ui/#iso:std:iso:26262:-3:ed-1:v1:en. Accessed 16 April 2024.

[62] ISO. ISO/IEC 15408-1:2022: Information security, cybersecurity and privacy protection - Evaluation criteria for IT security. https://ccsp.alukos.com/standards/iso-iec-15408-1-2022/, 2022.

[63] Jinsoo Jang and Brent ByungHoon Kang. In-process Memory Isolation Using Hardware Watchpoint. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 32. ACM, 2019.

[64] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

[65] Antti Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

[66] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[68] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[69] l4re. L4Re Operating System Framework. https://l4re.org/. Accessed 16 April 2024.

[70] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, page 93–102, New York, NY, USA, 2012. Association for Computing Machinery.

[71] Hugo Lefeuvre, Vlad-Andrei Badoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: towards flexible OS isolation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 467–482. ACM, 2022.

[72] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI*, pages 17–30, 2004.

[73] Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/Mechanism Separation in HYDRA. In James C. Browne and Juan Rodriguez-Rosell, editors, *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*, pages 132–140. ACM, 1975.

[74] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[75] Jochen Liedtke. Improving IPC by Kernel Design. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 175–188. ACM, 1993.

[76] Jochen Liedtke. On micro-Kernel Construction. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 237–250. ACM, 1995.

[77] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[78] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 819–835. ACM, 2021.

[79] Elton Lum. Study Confirms That Microkernel Is Inherently More Secure. https://blogs.blackberry.com/en/2020/09/study-confirms-that-microkernel-is-inherently-more-secure, 2020.

[80] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[81] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 461–472. ACM, 2013.

[82] Linux manual page. VDSO: Virtual Dynamic Shared Object. https://man7.org/linux/man-pages/man7/vdso.7.html. Accessed 16 April 2024.

[83] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 115–128. ACM, 2011.

[84] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[85] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 279–294. USENIX Association, 1996.

[86] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Microsoft. Windows Subsystem for Linux Documentation. https://learn.microsoft.com/en-us/windows/wsl/. Accessed 16 April 2024.

[88] Microsoft. MS Windows NT Kernel-mode User and GDI White Paper. https://learn.microsoft.com/en-us/previous-versions/cc750820(v=technet.10), 2014.

[89] Till Miemietz, Maksym Planeta, and Viktor Laurin Reusch. New Mechanism for Fast System Calls. *arXiv preprint arXiv:2112.10106*, 2021.

[90] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 269–284. USENIX Association, 2019.

[91] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci, editors, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 157–171. ACM, 2020.

[92] Ruslan Nikolaev and Godmar Back. VirtuOS: an operating system with kernel virtualization. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 116–132. ACM, 2013.

[93] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[94] Gabriel Parmer. The case for thread migration: Predictable ipc in a customizable and reliable os. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, page 91, 2010.

[95] Roman Penyaev. epoll: make sure all elements in ready list are in FIFO order. https://patchwork.kernel.org/project/linux-fsdevel/patch/20181212110357.25656-2-rpenyaev@suse.de, 2018.

[96] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[97] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[98] Mark Pitchford. Using Linux with critical applications: Like mixing oil and water? https://www.embedded.com/using-linux-with-critical-applications-like-mixing-oil-and-water/, 2021.

[99] Primate Labs Inc. Geekbench 5 CPU Workloads. https://www.geekbench.com/doc/geekbench5-cpu-workloads.pdf, 2019.

[100] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 563–577. IEEE, 2020.

[101] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333. IEEE, 2010.

[102] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel Linux (UKL). In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 590–605, New York, NY, USA, 2023. Association for Computing Machinery.

[103] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery.

[104] Timothy Roscoe. It's Time for Operating Systems to Rediscover Hardware. USENIX Association, July 2021.

[105] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: taming device drivers. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 275–288. ACM, 2009.

[106] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 73–86. ACM, 2009.

[107] seL4. seL4 capabilities. https://docs.sel4.systems/Tutorials/capabilities.html. Accessed 16 April 2024.

[108] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 335–350. ACM, 2007.

[109] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[110] Anand Lal Shimpi. The BlackBerry PlayBook Review. https://www.anandtech.com/show/4266/blackberry-playbook-review/14, 2011.

[111] Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, Amos Waterland, David Tam, and Andrew Baumann. K42: An Infrastructure for Operating System Research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42, apr 2006.

[112] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.*, 37(5):207–222, oct 2003.

[113] Sysgo. Open Source and ASIL D Certification – possible? https://www.sysgo.com/blog/article/open-source-and-asil-d-certification-possible, 2018.

[114] Willy Tarreau. Linux 2.6.32.71 (EOL). https://lwn.net/Articles/679874/, 2016.

[115] The Linux Kernel documentation. Energy Aware Scheduling. https://docs.kernel.org/scheduler/sched-energy.html, 2019.

[116] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[117] Johannes vom Dorp and René Helmke. Home Router Security Report 2022. https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/2022-11-28_HRSR_2022.pdf, 2022.

[118] Hannes Weisbach, Björn Döbel, and Adam Lackorzynski. Generic user-level PCI drivers. In *Proceedings of the 13th Real-Time Linux Workshop.*, 2011.

[119] Embedded Linux Wiki. Embedded Linux System Size. https://elinux.org/System_Size. Accessed 16 April 2024.

[120] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, New York, NY, USA, 2005. Association for Computing Machinery.

[121] Hyrum Wright. Hyrum's Law. https://www.hyrumslaw.com/, 2012.

[122] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. TreeSLS: A Whole-System Persistent Microkernel with Tree-Structured State Checkpoint on NVM. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 1–16, New York, NY, USA, 2023. Association for Computing Machinery.

[123] Feng Zhou, Jeremy Condit, Zachary R. Anderson, Ilya Bagrak, Robert Ennals, Matthew Harren, George C. Necula, and Eric A. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation OSDI'06, November 6-8, Seattle, WA, USA*, pages 45–60. USENIX Association, 2006.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

When will my ML Job finish? Toward providing Completion Time Estimates through Predictability-Centric Scheduling

Abdullah Bin Faisal
Tufts University

Noah Martin
Tufts University

Hafiz Mohsin Bashir
Tufts University

Swaminathan Lamelas
Tufts University

Fahad R. Dogar
Tufts University

Abstract

In this paper, we make a case for providing job completion time estimates to GPU cluster users, similar to providing the delivery date of a package or arrival time of a booked ride. Our analysis reveals that providing predictability can come at the expense of performance and fairness. Existing GPU schedulers optimize for extreme points in the trade-off space, making them either extremely unpredictable or impractical.

To address this challenge, we present PCS, a new scheduling framework that aims to provide predictability while balancing other traditional objectives. The key idea behind PCS is to use Weighted-Fair-Queueing (WFQ) and find a suitable configuration of different WFQ parameters (e.g., queue weights) that meets specific goals for predictability. It uses a simulation-aided search strategy to efficiently discover WFQ configurations that lie around the Pareto front of the trade-off space between these objectives. We implement and evaluate PCS in the context of scheduling ML training workloads on GPUs. Our evaluation, on a small-scale GPU testbed and larger-scale simulations, shows that PCS can provide accurate completion time estimates while marginally compromising on performance and fairness.

1 Introduction

Humans desire predictability in their daily lives [66]: from knowing how long their home-to-office commute will be to the arrival time of an Amazon package [80] or an Uber ride [7]. Fortunately, most real world systems (e.g., transportation, e-commerce, etc) meet this need by providing their users with a (reliable) prediction (e.g., estimated delivery date). As more and more of our lives move to the cloud (e.g., Metaverse [39, 73]), it begs the question of whether the cloud can offer similar predictability. More concretely, when a user submits a "job" (e.g., train a Machine Learning (ML) model) to the cloud, can the cloud provide a reliable job completion time prediction?

Such feedback can ensure a seamless experience and ease user frustration; perhaps more emphatically than simply making the cloud faster or fairer, according to studies in human psychology [22, 43] and systems usage [50]. It can also empower users to decide between different cloud platforms and services within a cloud based on the provided estimate, or be integrated with emerging inter-cloud brokers (e.g., SkyPilot [97]). In light of this, we advocate for the need to provide reliable job completion time predictions as a *core* primitive in today's cloud, akin to real world systems we interact with.

Several aspects of the user-cloud ecosystem can impact the (lack of) predictability of a job's completion time (e.g., failures [49], shared vs. dedicated resources [50], knowledge of individual job sizes [27, 58], workload characteristics etc.). The focus of this paper is on understanding the unpredictability stemming from the *scheduling* mechanism used by the cloud (sub)systems (e.g., FIFO vs. Fair Sharing vs. other policies). We situate our work in the context of ML workloads running on multi-tenant GPU clusters (e.g., PAI [89], Philly [49], etc). This is an important scenario as scheduling delays matter and can be highly variable due to the ever growing demand for GPUs by emerging AI applications such as those powered by Large Language Models (LLMs) [4], while other sources of unpredictability are minimal (e.g., workloads are predictable [49, 54, 62, 76]). It is also a challenging scenario because unlike the public cloud setting where users pay for dedicated (and hence predictable) GPU resources, these clusters are best-effort and heavily rely on intelligent scheduling mechanisms to determine how the underlying GPU resources are to be shared between ML applications or tenants (cluster users).

Our key observation is that a scheduling policy's use of *unbounded preemption* results in its inability to provide reliable Job Completion Time Predictions (JCTpred). Preemption is a key enabler for existing GPU scheduling proposals that optimize for metrics like minimizing average and/or tail job completion times (JCT) (e.g., Tiresias) [37, 76], fairness and resource efficiency (e.g., Themis) [14, 44, 62, 79, 93, 105], and meeting deadlines (e.g., Chronus) [30, 36, 59]. While crucial for achieving their respective objectives, the extensive use of preemption leads to unpredictability (i.e., prediction error) in a job's completion time due to (repeated) preemptions from future jobs. On the other hand, non-preemptive scheduling policies, such as First-In-First-Out (FIFO), are predictable (as future arrivals do not impact current jobs) but can result in extremely poor performance and a lack of fairness due to Head-Of-Line (HOL) blocking [26, 28, 37, 70].

This observation highlights an inherent trade-off between offering predictability and optimizing for other metrics, such as minimizing JCTs. Existing scheduling solutions typically occupy extreme points on this trade-off spectrum. They are either highly unpredictable due to the use of unbounded preemption or impractical because they do not employ preemption at all.

In light of these limitations, an important question arises: Are there intermediate points on this trade-off spectrum that can provide a balance between predictability and practicality? Specifically, can these intermediate solutions be achieved by controlling the extent of preemption used? Furthermore, the cluster operator may desire to operate at potentially any one of these intermediate trade-offs depending on their *relative* preferences. The trade-off space can be vast, and some points may be inherently less desirable than others. In such scenarios, how can we enable operators to express their preferences and efficiently explore the trade-off space?

To address these questions, we propose a novel scheduling framework called Predictability-Centric Scheduling (PCS) that aims to provide reliable JCTpred (predictability) while balancing other practical goals (flexibility) such as performance and fairness. PCS exposes a high level bi-directional preference interface which allows cloud operators to express the objectives they are interested in (e.g., avg JCTs vs. avg prediction error). To facilitate cloud operators in making an informed choice based on their *relative* preferences, PCS provides a *set* of Pareto-optimal trade-offs. Each Pareto-optimal trade-off improves one objective (e.g., predictability) while marginally sacrificing on other objectives (e.g., performance and/or fairness). This is unlike other tunable schedulers [52, 64, 71] which typically return a single solution.

At its core, PCS leverages Weighted-Fair-Queuing (WFQ) as a basic building block [23]. Our use of WFQ is motivated by the fact that it uses bounded preemption and offers direct control over the extent of preemption used. WFQ maps incoming jobs to a fixed number of queues, uses FIFO to schedule jobs within a queue and assigns a guaranteed resource share (weights) to each queue. These properties bound the preemptions and reordering experienced by jobs. Furthermore, the number of queues and their assigned weights are tunable parameters of the WFQ policy. This allows direct control over i) predictability (e.g., by creating limited number of queues), ii) performance (e.g., by assigning a higher weight to queues with smaller jobs), and iii) fairness (e.g., by assigning equal weights), motivating its flexibility and ability to achieve Pareto-optimal trade-offs.

Finding Pareto-optimal WFQ configurations is challenging because the space of possible configurations is large, with some trade-offs not feasible (e.g., optimal performance and maximum predictability) or beneficial (e.g., more unpredictable and unfairer than existing schemes). To address this challenge, PCS uses a highly-parallel simulation-based search strategy with an intelligent parameterization of WFQ using heuristics, to efficiently find suitable and feasible (Pareto-optimal) WFQ configurations. For example, we use the variation in job-sizes to determine the number of queues and thresholds as opposed to trying out arbitrary combinations. We show that Pareto-optimal trade-offs can be discovered for realistic workloads in a timely manner (§5.4).

A key benefit of PCS is that it is a generic scheduling framework, which can accommodate various types of jobs (e.g., network flows, DNN training jobs), allowing it to be realized in various multi-tenant scheduling scenarios. It only requires knowledge of a job's demand function, which can either be provided by the user or reliably estimated by the system [13, 50, 62]. This requirement is typically satisfiable for ML workloads and we later discuss the broader applicability of PCS to other scheduling scenarios in §6. PCS uses these demand functions to generate a completion time prediction as well as balance considerations for performance and fairness (e.g., when dealing with sub-linear scaling jobs) to be competitive with efficiency based schedulers (e.g., AFS [44]), as we show in §5.

We implement and evaluate PCS for realistic ML training workloads on a small-scale GPU cluster as well as large scale simulations. Our evaluation shows that PCS can successfully discover Pareto-optimal WFQ configurations to meet varying trade-offs. For example, PCS can reduce the prediction error by 50-800% while being within 1.1-3.5× of performance and fairness optimal schemes (§5).

Overall, we make the following contributions:

- We show that state-of-the-art GPU scheduling policies which optimize for performance and fairness [37, 44, 62] result in unpredictability. Our analysis shows that these policies typically lie on extreme points of predictability-performance or predictability-fairness trade-offs (§2).

- We design PCS, a generic job scheduler, which uses WFQ in a unique and novel way to achieve predictability and flexibility (§3.1).

- We provide a simple but expressive bi-directional interface to be used by cloud operators, enabling them to specify different high level objectives and giving them the ability to choose between trade-offs — a property existing scheduling systems fail to provide (§3.2).

- We implement a prototype of PCS in Ray [67] and evaluate it on a testbed and in simulations for realistic ML training workloads (§4 §5).

PCS is a step in providing predictability in today's cloud systems. It opens up important questions which we discuss in §6. Finally, we build upon and benefit from a large body of prior work in scheduling systems, which we discuss in §7. The code for PCS is made available at https://github.com/TuftsNATLab/PCS.

2 A Case for Predictable Scheduling

In this section we motivate the need for predictable scheduling to be a core primitive in today's cloud, and show how it is different from deadline-based scheduling. We provide several use-cases of predictable scheduling in the context of multi-tenant GPU clusters and draw analogies between real-world systems and the cloud. While this discussion has broader applicability in various scenarios (e.g., CPU scheduling, network bandwidth scheduling), we situate it in the context of multi-tenant GPU clusters and discuss the opportunities and challenges in supporting predictable scheduling in that context.

2.1 Why provide JCT predictions (JCTpred)?

A scheduling system that provides JCTpred can have two broad benefits: (1) Alleviating User frustration. Several studies on real-world systems (e.g., online retail [78], airlines [11]) show that providing a timeline to users can help ease frustration in the face of long and variable waiting times [43, 45, 99]. JCTpred can offer a similar role in the context of multi-tenant GPU clusters where users can suffer from large and unpredictable delays, inevitably leading to a poor and frustrating experience [22, 42]. Measurements on Microsoft's GPU cluster (Philly) show that ML training jobs can face up to 100 hours of queuing and preemption related delays [49], hinting that organizational GPU clusters are heavily oversubscribed. Research shows that users are often trying to guess when their training jobs will complete and that user-driven predictions can be off by more than 100%, with some users finding it *impossible* to make any meaningful predictions [30]. With the paradigm of AutoML, jobs that spawn hundreds of DNN trials [57, 62], and LLMs (e.g., GPT4 [4]) becoming mainstream, these issues will only exacerbate [9]. Additionally, predictability expectations are higher for users submitting repetitive jobs [50] and according to one study, 60% of training jobs exhibit DNN architecture similarity [54], emphasizing the need to provide JCTpred in such scenarios.

(2) Enabling decision making. In real-world systems, if the predicted timeline is long, customers may elect to perform other tasks or seek alternatives [63]. For example, estimated delivery dates can help shoppers decide between e-commerce platforms (e.g., Amazon [2] vs Temu [6]) and even between sellers within a platform. Today's cloud users have similar choices to make and JCTpred can enable them to make these choices in a more informed way. For example, it can help users decide between different cloud systems to run their ML workloads on, each option potentially offering a different cost-JCTpred trade-off. As a forward looking avenue, JCTpred can facilitate the growing eco-system around inter-cloud brokers which orchestrate seamless access to multiple clouds with low user effort (e.g., SkyPilot [19, 46, 84, 97]). Within a cloud, JCTpred can facilitate users in selecting between different model variants/pipelines to train, based on the expected accuracy-JCTpred trade-off [12, 20, 95, 101].

Why are Deadlines not the answer? One may wonder how the predictability metric is different from deadlines (and the large body of work on deadline-based scheduling for GPUs and beyond [16, 17, 30, 56]) where a user provides a deadline along with their job and the system tries to satisfy it. The fundamental difference is that in the deadline-based context, the burden lies on the *user* to provide a timeline to the system, with the system deciding the user's fate. We posit that it should instead be the *system* that provides the user with a timeline (i.e., a JCTpred), empowering them to decide whether it is acceptable or not. Our approach is analogous to real-world systems like ride-sharing where most users request a ride, wanting it ASAP (i.e., no deadline) while the system comes up with the expected arrival time of the ride.

Even if we try to shoehorn predictability into deadlines, it will be challenging for two reasons. First, coming up with a reasonable deadline is hard because the slowdown of a job is highly dependent on: i) cluster load (which can be highly variable and bursty at short timescales) and ii) underlying job-to-resource mapping which is (dynamically) determined at run time [50] and can result in significant variation due to heterogeneity in the underlying resources (e.g., low vs. high end GPUs [14, 71, 89], RDMA vs. TCP [31, 77], etc.,). Second, unless there is an inherent difference in user requirements (and hence deadlines), users have the incentive to specify a small deadline (i.e., to act greedy), which limits any prioritization the system can enable. In both the above cases, the lack of reasonable deadlines will render the system ineffective.

Feasibility of providing JCTpred. Computing JCTpred requires the knowledge of a job's size and its demand function (i.e., how its execution time will change based on the allocated GPUs). Fortunately, several attributes of ML workloads allow us to (approximately) estimate these. (1) Intra-job predictability. DNN training and inference jobs [37, 49, 62] exhibit intra-job predictability; the time it takes to run an inference job [38] or train a DNN for a specified number of epochs is fairly deterministic [62]. By profiling [44, 71, 74, 79] or modelling [31, 62, 76, 96, 105] the job's throughput and combining it with the provided job information (e.g., total number of epochs, convergence criteria, budget), its size and demand function can be estimated. (2) Recurring jobs. ML workloads are known to contain recurring jobs [24, 54, 90]. This can make history [75] and sampling [47] based strategies highly effective in estimating job sizes.

2.2 Limitations of Existing Schedulers

Reliably predicting the completion time of a user's job requires the underlying scheduling system to be predictable [30]. In this section, we highlight and analyze why existing schedulers used by GPU clusters today are either not amenable to reliable completion time predictions or are not practical.

Unbounded Preemption: the Price of Fairness and Performance. Performance and fairness-oriented schedulers frequently utilize *unbounded preemption* to prioritize and distribute resources among jobs. Preemption collectively refers to when some or all of the resources assigned to a job are reallocated or when its position in the queue is altered because of another (future) job. Although preemption is essential for achieving the goals of these schedulers, it can lead to unpredictability in a job's completion time [50]. Under preemptive scheduling policies, the arrival of future jobs can affect the completion times of current jobs by preempting the resources (e.g., GPUs) they are using.

Preemption manifests in today's cloud systems in the following ways: (1) Prioritization. When a higher priority job arrives and needs to be scheduled, running jobs are paused or waiting jobs are pushed further back in the queue. Several schedulers use prioritization to minimize JCTs and meet deadlines [8, 17, 30, 37, 56, 59, 76]. (2) Elastic Sharing. Jobs may need to be multiplexed together to achieve fairness and efficiency [14, 32, 44, 62, 93]. As new jobs arrive, the GPU share of existing jobs is reduced, stretching their completion times [50] or the scheduler takes away GPUs from existing less-efficient jobs and assigns them to new jobs that can utilize them more efficiently [13, 44].

Takeaway: Unbounded preemption results in unpredictability, making it challenging to provide a reliable JCTpred. A scheduler which utilizes bounded preemption will be more predictable.

Fixed Trade-offs. The other option is to use non-preemptive schedulers such as First-In-First-Out (FIFO) [86] and reservation based schemes [49] which are highly predictable as they guarantee resource allocation throughout the lifetime of a job — future job arrivals do not impact current jobs in the system. However, such schemes suffer from well known performance issues such as Head-Of-Line (HOL) blocking in the case of FIFO [26, 28, 37, 44, 70] and poor utilization for reservation based schemes [49, 89, 94]. There is no clear way to tune these schedulers that lie on extreme ends, to offer different trade-offs between predictability and other objectives. This is an issue because different cluster operators may want to settle for different (intermediate) trade-offs rather than switch between these two extremes.

Takeaway: Existing schedulers offer a fixed trade-off: predictable but high/unfair JCTs (non-preemptive) or low/fair but unreliable JCTs (unbounded preemptive). A scheduler which offers different trade-offs between these competing objectives is more practical.

Motivating Example. We use a simple toy example (Fig. 1a) with four jobs (J1-J4) to demonstrate these limitations. We analyze the performance of three schedulers — FIFO, Tiresias, and Themis — on reducing JCTs, unfairness, and unpredictability. FIFO is the default scheduler used in YARN [86]. Tiresias [37] prioritizes DNN training jobs with smaller remaining service times, while Themis [62] strives to minimize peak unfairness.[1] Tiresias and Themis are representative of a large space of policies which either use size based or fair scheduling, respectively. Unpredictability is captured as $\text{Pred}_{err} = \frac{JCT_{true} - JCT_{pred}}{JCT_{pred}}\%$, while unfairness is captured as the additional time it takes for a job to complete compared to its Fair Finish-Time (FFT) [15, 62] in percentage terms. JCTpred is computed at the time of a job's submission and is defined as the time it takes for a submitted job to complete given a scheduling policy and the current cluster state (i.e., GPU allocations to existing jobs). We provide a practical way to compute it for all scheduling policies in §4.

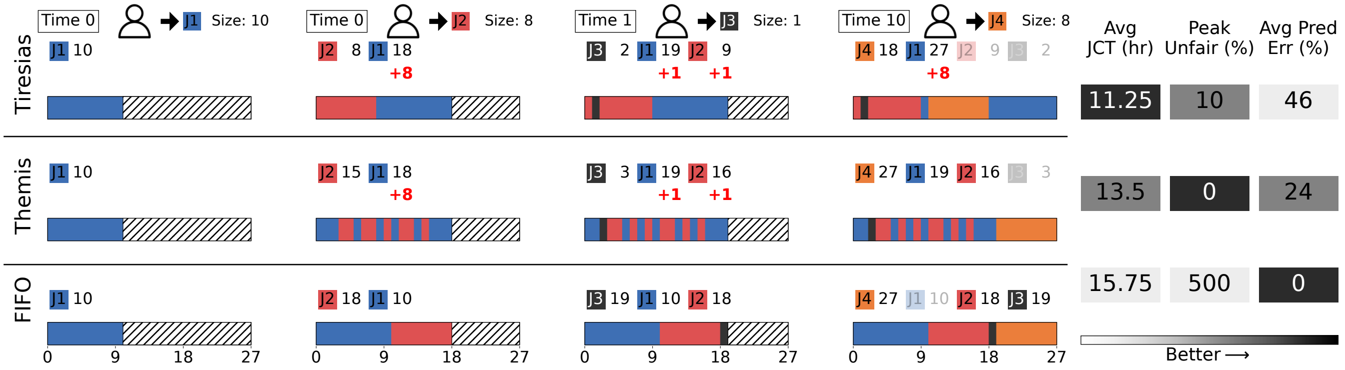
As new jobs arrive (moving left to right in Fig. 1a), both Tiresias and Themis result in a change in completion times of previous jobs. For instance, in Tiresias (top row), when J2 and J4 arrive in the system (second and fourth column), there is an eight time unit increase in J1's predicted JCT each time. While Tiresias achieves the minimum average JCT, it results in the highest average prediction error — 46% Pred_{err} in our example. Similarly, in Themis (middle row), the scheduler's multiplexing of J1 and J2 causes J1's predicted completion time to increase by eight time units (second column). While Themis ensures all jobs finish before their FFT (unfairness of 0) and also avoids HOL blocking, it has an avg Pred_{err} of 24%. The FIFO scheduler (bottom row) achieves a prediction error of 0 as it is non-preemptive, but is the most unfair strategy and has the highest average JCT. Figure 1b summarizes these outcomes.

We now discuss PCS, a generic resource scheduler that attempts to offer predictability while being flexible in balancing performance and fairness related objectives.

3 Predictability-Centric Scheduling (PCS)

Requirements. Our analysis in the previous section reveals that a scheduling policy with *no preemption* (i.e., FIFO) results in maximum predictability. However, this comes at a high cost in terms of performance (i.e., JCTs) and fairness, which makes it an *impractical* option. On the other extreme, there are scheduling policies that have *unbounded preemption* (e.g., Fair-Share, Shortest Job First, etc.). In these policies, an influx of future arrivals can arbitrarily stretch the completion

[1] We use a lease duration of 1 time unit for Themis and assume job size information is known for all schedulers



(a) Impact of future arrivals on scheduling

(b) Summary

Figure 1: Toy example with 1 GPU, demonstrating the limitation of existing strategies. (a) shows how the scheduling order changes as jobs arrive over time under the Tiresias [37], Themis [62], and FIFO [86] schedulers. Time moves from left to right with a new job arriving in each column. The expected finish times for the current jobs are displayed above the current schedule. Jobs that are finished are grayed out. (b) summarizes the results for performance, fairness, and predictability for these policies.

time of an existing job, making them unsuitable for providing predictability.

This insight distills into the following two requirements that a scheduling policy must satisfy in order to provide predictability while being practical:

- R1 Predictability Requirement: a scheduling policy must have *bounded preemption*. This is essential in order to provide reliable JCT predictions.

- R2 Flexibility Requirement: it should be able to approximate maximum predictability, optimal performance, and maximum fairness. Most importantly, it should be able to achieve Pareto-optimal trade-offs between these. This is essential for practicality.

PCS Overview. Our solution to this end is PCS, a generic scheduling framework (Fig.2), which captures these requirements using a high level preference interface (§3.2), and meets them by using the well-known Weighted-Fair-Queuing (WFQ) [23] policy in a novel way. The inherent properties of WFQ, careful selection of various WFQ parameters (number of queues, weights, etc) along with how each job is mapped to a queue and processed within it, allow us to meet our objectives (§3.1). Specifically, WFQ creates a fixed number of queues, assigns each of them a guaranteed share of the resource capacity (weights) and schedules jobs within a queue in FIFO order – this allows WFQ to satisfy our predictability requirement (R1). Similarly, the number of queues, weights, and how jobs are mapped to each queue are *tunable*, allowing it to offer the desired flexibility (R2).

A key component of PCS that enables the above functionality is the *preference solver* (§3.3), which translates the specified high level objectives into a set of Pareto-optimal WFQ configurations using a simulation-based search strategy.

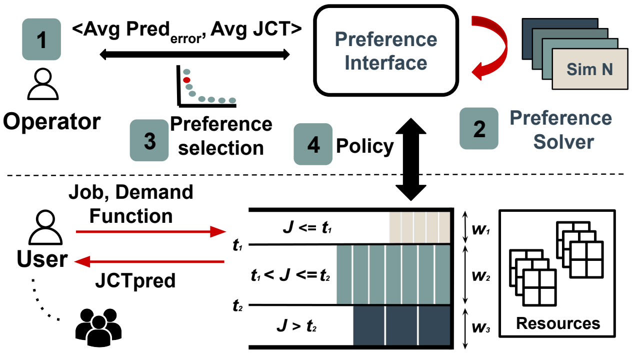


Figure 2: Key components of PCS: The preference framework can be used by operators to specify high level objectives. The preference solver uses a simulation-based search strategy to find Pareto-optimal WFQ configurations that are then shared with the operator. On the critical path, users submit their jobs along with the job's demand function and are given a JCTpred.

The simulation based search strategy is *not* on the critical path of a submitted job; it operates at coarser timescales, aligned with changes in workloads. Since ML workloads are fairly stable, expending the time to search for Pareto-optimal WFQ configurations is feasible. While the space of possible configurations is large, we use an intelligent parameterization of WFQ (e.g., coefficient of variation of job sizes within a queue) to navigate it in a feasible manner. Once a particular WFQ configuration is selected, it can be used to schedule submitted jobs as they arrive.

An important benefit of PCS is that it is a generic scheduling policy – it operates on the notion of a *job* which could be a network flow or a DNN training job etc. To deal with the varying needs of these different scenarios, in PCS, a job is defined using a demand function. The demand function is a mapping between the job's execution time and the resources allocated to it i.e., $demand(n) \mapsto T_{exec}$ and has a minimum ($demand_{min}$) and maximum ($demand_{max}$) resource allocation bound, denoting the execution time under the lowest and highest possible allocation. For ML workloads, in particular DNN training, the demand function is sophisticated, as different models can have different speedups based on the GPU type and affinity and is estimated on the users behalf, as discussed in §4. For scenarios like network (co)flow scheduling [18, 26, 28], the demand function is simpler, as we discuss in §6.

Finally, the user submits their job, optionally including its demand function. PCS then computes and returns the predicted completion time (JCTpred).

We now explain in detail, our choice of using WFQ as a building block (§3.1), followed by preference solver and interface.

3.1 WFQ under PCS

We begin by motivating why WFQ is a useful starting point and then share PCS's careful usage of WFQ in meeting our objectives. Our observation is that a lack of preemption, as in FIFO, and a non-zero guaranteed resource share for jobs is crucial for predictability. WFQ uses FIFO scheduling within each queue and across queues the resources are shared according to, strictly positive, queue weights, helping us satisfy the predictability requirement. To highlight the flexibility of WFQ, we show how it can be configured to optimize for extreme points in the trade-off space of maximum predictability, performance and fairness.

- **Maximum Predictability:** WFQ with a single queue is exactly FIFO scheduling which achieves a prediction error of 0
- **Near-optimal Performance:** Shortest Job First (SJF) is near-optimal in minimizing avg JCT for a single bottleneck [83]. WFQ can map each job to its own queue and give a higher weight to queues with smaller jobs, approximating SJF as shown by prior work [18, 88].
- **Max-Min Fairness:** If each job is mapped to its own queue and each queue gets an equal weight, WFQ can emulate Max-Min fair allocation which minimizes unfairness for a single bottleneck [33].

As our analysis in §2 reveals, a combination of these objectives is more practical. WFQ offers the necessary baseline flexibility in the queue creation, job mapping and weight assignment strategy. This motivates that we can achieve a combination of these objectives as well, which leads to PCS's preference interface §3.2.

Beyond vanilla WFQ. Our core idea is the novel use of WFQ to meet our objectives. First, PCS intelligently chooses the number of queues, weights and the job-to-queue mapping strategy to find various Pareto-optimal configurations, including extreme points, such as FIFO, SJF and Max-Min Fair Share. In PCS, jobs are mapped to different queues based on their size and a set of thresholds ($t_i^{\prime}s$), while strictly positive weights (w_i's) dictate the guaranteed resource share for each queue. For example, jobs with size $> t_k$ and $\leq t_{k+1}$ will be mapped to the k^{th} queue.

Second, within a queue, PCS deviates slightly from a strict FIFO schedule in favor of improving performance and fairness. In PCS, a job's demand function is used to cap the resources allocated to it. For example, a job at the head of its queue may not be assigned all of the guaranteed resource share of its queue (as in strict FIFO); instead, some of the resources may be allocated to the jobs behind it. This allows PCS to handle jobs that exhibits diminishing speedup w.r.t. increase in allocated resources, such as ML training jobs (§4).

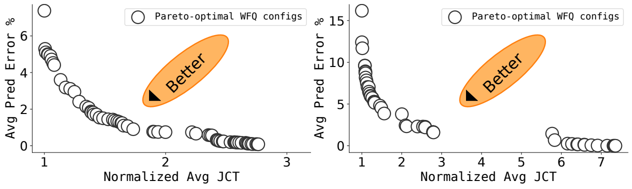
Finally, to ensure work-conservation, any residual allocation is then redistributed first within a queue in FIFO order by incrementally relaxing the cap on each job's demand function and then across queues proportional to their weights. We expose the weights, thresholds and the demand capping criteria to the preference solver which searches over the space of possible choices of these parameters in order to discover Pareto-optimal configurations (§3.3).

Pred$_{err}$ in PCS. Since PCS is work-conserving, a job may get a higher resource share compared to its guaranteed share. For example, if a job arrives when no other job is present, it will be allocated all the available resources (up to $demand_{max}$). This can lead to prediction errors (Pred$_{err}$) if in the future, other jobs arrive and occupy different queues.

In PCS, we bound these errors in a few ways. Firstly, a job's worst-case completion time is strictly bounded, irrespective of the number of future arrivals in other queues or its own queue. This is possible because each queue is assigned a strictly positive weight and uses FIFO scheduling (both properties of WFQ). By bounding the worst-case completion time of a job, the number of preemption events a job will experience during its lifetime is bounded, resulting in bounded Pred$_{err}$. Second, we exploit the fact that cloud systems are typically highly loaded [37], and by limiting the queues created we can reduce the likelihood of sudden and drastic changes in queue occupancy due to future arrivals. Furthermore, the exact load of a queue is controlled by the thresholds and weight assignment strategy. These observations guide us in discovering Pareto-optimal WFQ configurations.

3.2 Preference Interface

PCS exposes a simple yet expressive bi-directional interface that allows operators to specify high level objectives and present Pareto-optimal trade-offs (WFQ configurations) to choose from. This is unlike other tunable systems [52, 64, 71] which assume operators are aware of the trade-offs involved



(a) Trace #1

(b) Trace #6

Figure 3: Pareto front of the trade-off between Pred$_{err}$ and normalized average JCT for workload-2 (§5). *Better* indicates WFQ configurations that achieve a tight bound on average/tail Pred$_{err}$ while incurring the smallest possible increase in average JCT.

i.e., PCS *actively* tries to help the operator in making an informed choice. Our decision to use Pareto-optimal choices, as a way to support informed decision making, is grounded in fundamental literature on multi-objective decision making [34, 72], which maps neatly to the problem PCS is trying to address: predictability while being practical.

The preference interface itself, is general enough to be used in scenarios beyond predictability as well. For example it can be used to strike a balance between fairness and performance (e.g., Carbyne [35]) and between minimizing average and tail JCTs [26, 28, 35, 70]. The preference interface exposes the following API:

```
void SetPreference(
    Obj1 <Metric, Measure>,
    ...,
    ObjN <Metric, Measure>
);
List<WFQConfig> UpdateParetoFront();
void SetWFQConfig(WFQConfig config);
```

The current PCS API supports three `Metrics`: Performance (*JCT*), Fairness (*unfairness*), and Predictability (Pred$_{err}$). The `SetPreference()` method is used to specify the list of objectives; repeated entries are allowed to support exploring trade-offs across *different* measures of the *same* metric. For each objective, `avg(.)` or a particular `percentile(p)` needs to be specified as a `Measure`.

We envision the following API usage life cycle from an operators perspective: (1) Upon cluster deployment or drastic workload changes, the operator uses the `UpdateParetoFront()` method to kick-start the preference solver (§3.3). (2) The preference solver uses the updated workload information and preferences to discover the set of Pareto-optimal WFQ configurations. (3) Once complete, the operator can choose a specific WFQ configuration (`WFQConfig`) to be used by invoking the `SetWFQConfig()` method.

`UpdateParetoFront()` requires PCS to passively collect job size information and maintain a workload history. When bootstrapping, PCS starts with a default WFQ configuration, which can be any one of the extreme points in the trade-off space (e.g., FIFO) described in §3.1. When sufficient workload information is gathered, the preference solver is initiated.

We now show how the API is used to target scenarios covered in our evaluation.

Average JCTs *vs.* Average Pred$_{err}$: Minimizing average JCTs is a popular performance objective and has been a focus of several scheduling policies [37, 76]. To explore the trade-off between performance and predictability, one can specify it as `SetPreference(<JCT, avg>, <Pred`$_{err}$`, avg>)`. We use this for evaluating PCS for workload-1 and workload-2 in §5.

Average JCTs *vs.* Tail Pred$_{err}$: Prediction error can be tightly bound by specifying the tail Pred$_{err}$ (e.g., p99) as a measure of predictability. In such a case, the objectives would stay the same as in the above example, however, the measure for Pred$_{err}$ would change from `avg(.)` to `percentile(99)`. PCS uses this specification for workload-3 where low p99 Pred$_{err}$ is challenging to achieve with other policies.

Pareto Fronts. Figure 3 shows the set of Pareto-optimal WFQ configurations generated by PCS for two realistic DNN training workloads.

3.3 Preference Solver

The preference solver is responsible for finding Pareto-optimal WFQ configurations for the objectives specified. It uses a multi-objective search algorithm to navigate the space of possible WFQ configurations. The optimization parameters consist of the (1) number of queues, (2) queue weights, (3) queue thresholds, and (3) resource allocation cap. These parameters are deemed relevant as they directly control the different trade-offs involved between the objectives considered by PCS. For example, the number of queues influence the degree of preemption and hence predictability, while the resource allocation cap influences the overall efficiency and hence performance. Other common scheduling dimensions, such as explicit priorities or deadlines, are not considered as they relate to objectives beyond performance, fairness and predictability. For example, some systems may want to prioritize a longer running job. This conflicts with the goal of minimizing JCT; which is rather achieved by assigning a low priority to such jobs. Catering to such scenarios is beyond the scope of PCS.

Finding Pareto-optimal configurations is challenging due to the combinatorial nature of the configuration space. The solver intelligently parameterizes each configuration to make the search process feasible. It uses a simulation-based approach to evaluate the performance, predictability and fairness of a particular WFQ configuration. These are fed to the search algorithm, which decides the configurations to keep, try out next, and discard.

Intelligent Parameterization. To reduce the number of optimization parameters we use the following heuristics:

- **Creating Queues and Thresholds:** Large variation in job-sizes within a queue can lead to HOL blocking but creating too many queues increases preemption events and deteriorates predictability. In PCS, queues are created based on the squared coefficient of variation (C^2) in the job-sizes, as done by prior work [28]. We use a tunable parameter $0 < \mathcal{T} < C^2_{max}$ to ensure that queues are created such that C^2 of job-sizes within each queue is $\leq \mathcal{T}$, where C^2_{max} is the C^2 of the entire job size distribution. A larger (smaller) $\mathcal{T}$ results in fewer (more) queues created.

- **Systematic Weight selection:** Higher weights given to queues with smaller jobs improves performance for most workloads. On the other hand, a balanced weight assignment strategy may improve fairness instead. Based on this, we constrain the weight for the i^{th} queue to be $w_i = e^{-i \times \mathcal{W}}$. $\mathcal{W}$ is a tunable parameter which controls the relative weights for each queue. A higher (lower) value of $\mathcal{W}$ leads to a greater (lesser) disparity in weights among the queues. For heterogeneous deployments, containing several resource types (e.g., k different GPU types) we use $\mathcal{W}_1 \ldots \mathcal{W}_k$.

- **Finding Demand caps:** The resource efficiency of a job is used to decide its allocation cap and it is computed as $\zeta(n) = \frac{demand_{min}}{n \times demand(n)}$, where $demand_{min}$ is a job's execution time under its minimum possible allocation (e.g., 1GPU) and it is a non-increasing function of the allocated resources. For linear scaling jobs, $\zeta = 1$, while for jobs that scale sub-linearly, $0 < \zeta \leq 1$. Instead of a fine-grained efficiency comparison between all jobs, we introduce a tunable threshold ζ_{min} to be used for all jobs, to reduce the search parameters. Using this, a job's resource allocation is capped at k such that $\zeta(k) \geq \zeta_{min}$. Intuitively, a low (high) ζ_{min} means the scheduler is more (less) tolerant towards inefficient jobs. Our evaluation in §5 shows that this heuristic is competitive compared to the approach taken by other efficiency based schedulers (e.g., AFS [44], Themis [62]).

Using these heuristics, WFQ($\mathcal{T}, \mathcal{W}, \zeta_{min}$) becomes the succinct parameterization of each configuration. Different values for these parameters results in different trade-offs between the objectives specified by the operator. For example, setting ($\mathcal{T} = C^2_{max}, \zeta_{min} = 0$) achieves maximum predictability (i.e., strict FIFO) as only one queue is created and no allocation cap is enforced.

Simulation-based Search. We use a simulation based approach to discover Pareto-optimal WFQ configurations. Our methodology utilizes a simulator, which accepts a WFQ configuration (denoted by ($\mathcal{T}, \mathcal{W}, \zeta_{min}$)) as input. The simulator evaluates the provided configuration under a random sample ($\approx$ 1000 job arrivals) of the collected workload (i.e., size distribution and average arrival rate) and outputs the resulting JCT, FFT and JCTpred metrics. The results are then fed to the search algorithm.

The search algorithm samples the search space of possible WFQ configurations and interacts with the simulator to converge to Pareto-optimal solutions. We use SPEA2 as our choice of the search algorithm. It is based on evolutionary search and supports optimizing over multiple objectives [106]. Other multi-objective optimization algorithms can also be used as an alternate, in a plug-and-play fashion. To improve the robustness of each discovered WFQ configuration, it undergoes multiple evaluations under different random samples of the workload to increase its likelihood of being Pareto-optimal.

While we don't have any theoretical basis for the convergence and optimality properties of our approach, it works well in practice and can timely ($\approx$ 1hr) discover the Pareto front for a reasonably sized GPU cluster. Our evaluation confirms that Pareto-optimal configurations found using simulations follow the same trade-offs on the testbed experiment (§5.2). We micro-benchmark the feasibility of the simulation-based search strategy in §5.4.

4 PCS for GPU Scheduling

We now describe the realization of PCS for DNN scheduling on GPU clusters, highlighting important differences and how our abstraction of a job and demand function handles these differences.

DNN Jobs. A job is either a single DNN training job or a collection of DNN trials being run as part of a hyper-parameter tuning task (i.e., AutoML). The demand function for such workloads can be complicated. Modern DNNs require distributed training (e.g., data parallelism) on multiple GPUs. They are known to have sublinear speedup w.r.t to the (1) number, (2) type and (3) locality of GPUs allocated to them [44, 62, 71, 79]. PCS relies on existing techniques, such as throughput modelling and profiling, to estimate a job's demand function. As described in §3, the demand function describes how the job's execution time changes with different resource allocations. Since allocations have three dimensions: locality, GPU type and number of allocated GPUs, the demand function takes as input different combinations and returns the corresponding execution time. This is akin to the notion of bids in Themis [62] and throughput in Gavel [71].

Role of Demand Functions. PCS uses $demand_{min}$ as a job's size to map it to its respective queue. The demand function is also used to cap the maximum GPU allocation for DNNs that exhibit sub-linear speedup. Allocating GPUs up to the maximum demand ($demand_{max}$) for such jobs can result in poor performance. We evaluate this approach and show that it works for DNN workloads consisting of jobs that scale sub-linearly (§5.3). As described in §3.3, the allocation cap is a tunable parameter for the preference solver and can be adjusted for different trade-offs and workloads.

Implementation. We implement PCS as a central coordinator in Python and use the Ray cluster manager [67] for GPU allocation enforcement as well as for general cluster management tasks such as fault tolerance. Each job is either a single trial or consists of multiple trials as part of a hyper-parameter tuning algorithm provided by RayTune [60]. We use a custom `ray_trial_executor` to control starting, stopping and preempting individual trials based on the allocations computed by PCS. To determine the remaining service requirements of running jobs, we use various callbacks (e.g., `on_step_start`) exposed by RayTune to get the exact number of iterations completed by each job and multiply it with the profiled time per iteration.

In addition to the central coordinator, PCS consists of an agent, which uses information about running jobs to provide a prediction interface. This interface returns a JCTpred in real time to the user whenever they submit their jobs. The agent computes JCTpred by "virtually" playing out (i.e., in a simulated setting) the current snapshot of the cluster state (e.g., running jobs, available GPUs etc.), accounting for preemption overhead and demand functions of other jobs, to determine the time at which the job will end. This approach is inspired by prior work [29, 82], which use a simulator to compute a job's duration under different resource allocation strategies.

5 Evaluation

We evaluate PCS on a 16 GPU cluster with a realistic AutoML style workload to validate our observations. We also cover additional workloads at a larger scale using an event-based simulator. Our evaluation covers different application workloads (e.g., heavy-tailed vs. light-tailed, AutoML apps vs single DNNs), different scheduling schemes (e.g., Tiresias [37], Themis [62]) and different metrics (e.g, avg, p99).

Our evaluation attempts to answer the following key questions:

- **How does PCS perform in terms of Pred$_{err}$ compared to other schemes?** Our testbed results reveal that PCS configurations achieve significantly lower Pred$_{err}$ (20%) while being within 10% of high performing schemes on the performance side.

- **Does PCS work well across different workload types?** The flexibility and predictability provided by PCS holds across different workloads and across preference specifications. PCS can discover configurations that bound the tail Pred$_{err}$ to be within 100% compared to AFS [44] and Tiresias [37] which suffer from $\geq$ 300% error at the tail.

- **Are PCS configurations fair?** PCS configurations that are optimized for the performance vs Pred$_{err}$ trade-off do not necessarily suffer from unfairness because each queue is guaranteed a GPU share which helps in avoiding starvation.

- **Is the search process feasible?** Our micro-benchmark reveals that the search process can complete within O(hr), making it practical to use, and PCS configurations discovered using the simulation based search-strategy observe the same trade-off *trends* on the testbed.

	Testbed (16 GPUs)	Simulations (64 GPUs)	
Workload	Workload-1	Workload-2	Workload-3
Job Type	AutoML	DNN	DNN
DNN/job	1-20	1	1
GPUs/DNN	1	1-52	1-8

Table 1: Summary of the settings used to evaluate PCS

5.1 Experimental Setup

Testbed. Our testbed cluster consists of 16 1-GPU c240g5 machines in the Wisconsin Cloudlab cluster [3]. Each machine has one NVIDIA P100 GPU with 12GB GPU memory.

Simulation. We use an event-based simulator to cover workloads that contain jobs requiring O(100) GPUs on a homogeneous 64 GPU cluster. We have verified the fidelity of our simulator with trace results from Microsoft [49] and our testbed results with the difference being within 5%.

Pareto Search. The Pareto-optimal configurations for our workloads are discovered by the preference solver §3.3 running on a cluster of 10 c220g5 machines in the Wisconsin Cloudlab cluster [3]. It is important to note that these configurations are discovered and evaluated on different sampled subsets of the workload i.e., there exists a notion of training set vs testing set.

Workloads. Table 1 summarizes the characteristics of our candidate workloads. We now discuss these workloads in detail.

- **Workload-1:** We borrow this workload from Themis [62] (referred in their work as *Workload-1*). For our testbed evaluation we scale down the maximum number of trials per app to 20 and the maximum service time to 2 GPU-hours. The maximum number of GPUs per trial is set to 1. Each trial tunes a different hyper-parameter (learning rate and momentum) of popular vision models from the VGG family [81].

- **Workload-2:** We use traces from 6 virtual clusters from Philly [5] containing the largest number of jobs. In contrast to other workloads, jobs in these traces exhibit sub-linear scaling. We use the scaling data shared by Hwang et al. on Github [1]. More details are in the attached artifact appendix A.

- **Workload-3:** This is borrowed from Gavel [71] (referred in their work as *continuous-multiple*). It is a heavy-tailed workload, with a large number of very small jobs and few long running jobs. We run this workload at a job arrival rate of 4 jobs/hr.

A common characteristic of these workloads is that the minimum requirement of any job is 1 GPU i.e., as long as there is at-least one GPU available, a job can start. This also holds true for RayTune apps which we use in our testbed evaluation.

Scheduling Policies. We compare PCS against FIFO and recently proposed GPU scheduling systems (Themis [62], Tiresias [37], AFS [44]). All scheduling policies considered in our evaluation are "work-conserving" and elastic i.e., they redistribute unused GPUs amongst other jobs according to the policy. For example for FIFO if a job only needs k GPUs and n are available, where $n > k$, then $n-k$ are attempted to be allocated to the next-in-line jobs.

We now describe our implementation of Themis, AFS, and Tiresias that we use in our evaluation.

- **Themis** [62]. On every resource change event and lease duration expiry, in-progress jobs report their fair-finish-time and we allocate GPUs to jobs in descending order of the reported number. We do not consider the scenario where jobs could lie and thus do not require the partial allocation mechanism. The lease duration is set to 10 minutes as per the recommendations of the authors.

- **Tiresias** [37]. Since we assume complete knowledge about job sizes, here Tiresias emulates the Shortest-Remaining-Service-First (SRSF) policy.[2] As such, GPUs are first allocated to jobs with the lowest remaining service times on every resource change event.

- **AFS** [44]. This scheduler tries to minimize avg and tail JCTs while maximizing resource efficiency. On every resource change event we compute each job's allocation using the AFS-L algorithm.

PCS Configurations. We use three configurations for PCS: (1) PCS-pred, (2) PCS-JCT, and (3) PCS-balanced. Each configuration makes a different trade-off. PCS-pred has the highest JCT but the lowest Pred$_{err}$ amongst the three. For each workload and objective the set of WFQ configurations are different and are discovered using the preference specifications described in §3.2.

Comparison Criteria. We evaluate the merit of PCS on three fronts:

1. Job Completion Times (JCTs): A commonly used metric to evaluate the performance of scheduling policies.

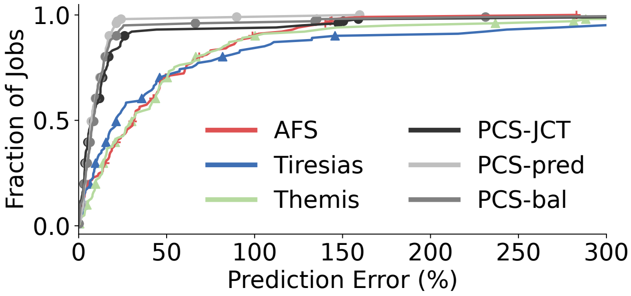
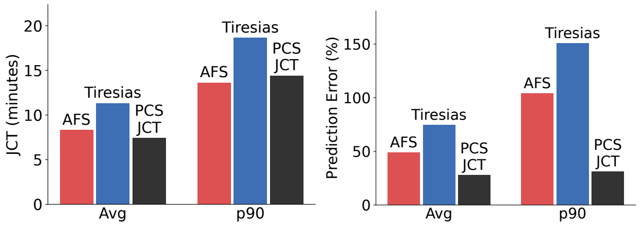
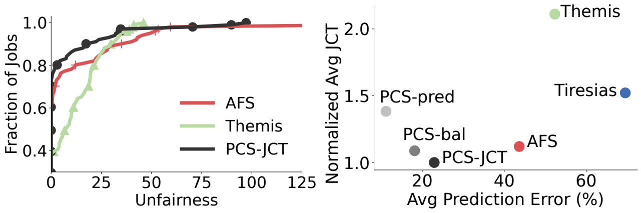


Figure 4: [TESTBED] Distribution of Pred$_{err}$ showcasing three configurations of PCS discovered by PCS — performance oriented, predictability oriented and balanced compared to other schemes.



(a) JCTs **(b)** Pred$_{err}$

Figure 5: [TESTBED] Zooming into the trade-off between performance and predictability. PCS is within 1.1× AFS at p90 JCT, with significant improvement to predictability.



(a) Distribution of unfairness **(b)** Pareto font

Figure 6: [TESTBED] (a) shows the CDF of unfairness showcasing that PCS does not significantly compromise on fairness compared to a policy that optimizes for it. (b) highlights the Pareto-optimal configurations discovered in a simulated environment observe the same trend on the testbed evaluation.

2. Unpredictability (Pred$_{err}$): A proxy to capture the error in JCTpred.

3. Unfairness: It captures the extra time taken by a job to complete, compared to its fair-finish-time (FFT) and is 0 for jobs that complete before their FFT.

We consider all important statistics such as the average and tail (e.g., p99 Pred$_{err}$, avg JCTs) for all objectives. For each objective, a lower value is better.[3]

[2] Referred to as Tiresias-G in their paper

[3] The testbed result is an average across 3 seeds while simulation results are an average across 5 seeds

5.2 Testbed Experiment

For our main experiment we compare three PCS configurations, discovered by the preference solver for workload-1, against other schemes.

A tight bound on Pred$_{err}$. Figure 4 shows the CDFs of Pred$_{err}$ achieved by different scheduling schemes and the three PCS configurations. We observe that all PCS configurations are able to achieve significantly lower Pred$_{err}$. At p90, the difference is an 80% lower error achieved by all configurations compared to other schemes. At higher percentiles, PCS-pred provides the lowest worst-case Pred$_{err}$ of 150% while other schemes have a long tail. PCS-JCT still has a lower Pred$_{err}$ up until p95.

Negligible performance sacrifice for high predictability. Figure 5 zooms into the performance versus predictability trade-off achieved by PCS-JCT compared to AFS and Tiresias which aim to minimize JCTs. We see that PCS-JCT achieves equivalent performance to AFS and Tiresias for the average JCTs. It is within 1.1× of AFS at p90, however this trade-off results in significant improvement on the predictability front, where Tiresias and AFS suffer. Pred$_{err}$ under PCS-JCT is within 20% for average and p90 Pred$_{err}$ while AFS and Tiresias have $\geq 40\%(\geq 100\%)$ prediction error at the average (p90). This signifies that PCS-JCT trades off negligible performance to significantly improve predictability. Another source of improvement we observe is that since PCS makes limited use of preemption, overheads associated with preempting running jobs are reduced compared to other schedulers. This is the reason behind PCS outperforming performance oriented schedulers (i.e., AFS and Tiresias).

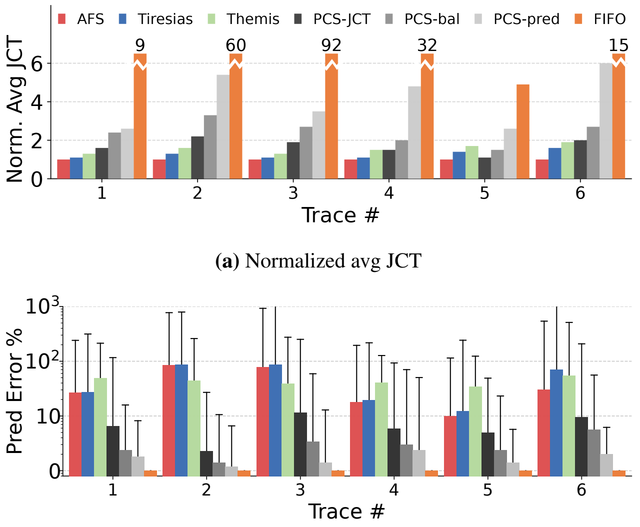
Unfairness. Figure 6a compares the unfairness for PCS-JCT compared to AFS, which optimizes for average JCT, and Themis, which minimizes unfairness. PCS achieves lowest unfairness till p95 and has the worst-case unfairness $\leq 100\%$ compared to AFS which has a worst-case unfairness $> 200\%$. Not surprisingly, Themis offers the tightest bound on the worst-case unfairness of less than 50%.

Pareto-optimality. Finally, figure 6b shows different PCS configurations that achieve different trade-off points in the space of avg JCT vs avg Pred$_{err}$. As expected, PCS-JCT has the lowest avg JCT, while PCS-pred achieves the lowest average Pred$_{err}$.

5.3 Simulation Experiments

We now consider different workloads at a larger scale in simulations and show the trade-offs achieved by suitable PCS configurations compared to performance and fairness optimal schedulers.

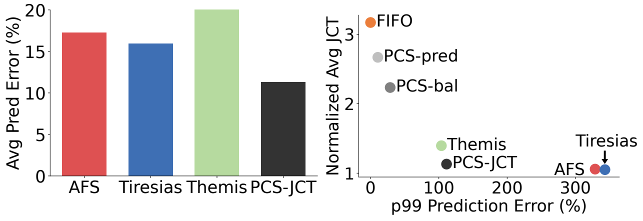
Workload-2. Figure 7 compares the performance and predictability of PCS with other schedulers for workload-2. For such workloads, AFS achieves the lowest possible avg JCT by giving more GPUs to jobs with higher efficiency. Despite its conservative approach in dealing with sub-linear scaling jobs, PCS remains within 1.5 to 4× of the optimal scheme for minimizing avg JCT, while drastically reducing the avg and tail Pred$_{err}$. For example, PCS-JCT reduces the average Pred$_{err}$ from 80% to 1% for Trace #2 and PCS-pred reduces the p99 Pred$_{err}$ from 900% to 10% for Trace #3.



(a) Normalized avg JCT

(b) Predictability

Figure 7: [SIM]: PCS for workload-2. a) Most PCS configurations are within 1.5-4× of the performance optimal policies while b) shows that they drastically reduce the average and tail Pred$_{err}$. In b), the bar height (line) represents average (p99) Pred$_{err}$ and the y-axis follows a logscale.

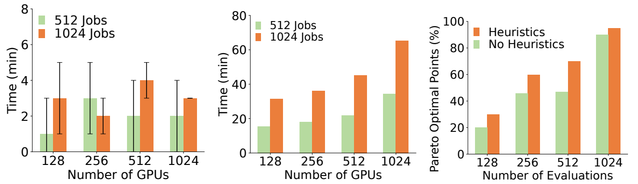


(a) Prediction Error

(b) Pareto front

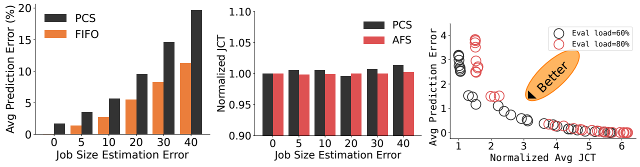
Figure 8: [SIM]: Showing that schemes that optimize for average JCTs for workload-3 also have a small average error. For such workloads, the tail Pred$_{err}$ becomes an important metric.

Workload-3. Figure 8 compares the different schedulers for workload-3. For this workload, we observe that schedulers optimized for performance, including PCS-JCT achieve reasonably low average Pred$_{err}$. This is because for workload-3, majority of the jobs are small and similar in size. For such workloads, tail Pred$_{err}$, becomes important owing to some jobs being starved under priority schedulers. With the appropriate preference specification, PCS discovers configurations



(a) Single sim time **(b)** Search time **(c)** Heuristics

Figure 9: Feasibility of the simulation-based search strategy. (a) captures the time to run a single simulation, (b) shows the time it takes to discover the entire Pareto-front. (c) highlights that intelligent parameterization helps in discovering more Pareto optimal points for a given evaluation budget.



(a) Pred_{err} **(b)** Avg JCT **(c)** Sensitivity

Figure 10: Shows the effects of error in job size and load estimation. a) compares the average Pred_{err} using PCS and FIFO [86] with varying job size estimation error. b) compares the avg JCT of PCS and AFS [44] under the same error. c) shows sensitivity of WFQ configs to load changes.

that can drastically reduce the p99 Pred_{err}. For example, PCS-JCT reduces the Pred_{err} from $\geq 300\%$ to $\approx 100\%$ while being within $1.1\times$ of performance optimal schemes (Fig. 8b).

5.4 Micro-benchmarks

Feasibility of the Search Strategy. Figure 9a shows that PCS takes $O(minutes)$ to run a single simulation for a given load (number of jobs) and cluster size (number of GPUs). PCS extensively leverages the underlying parallelism to discover the Pareto-front – requires running ≈ 1000 simulations– in approximately 60 minutes (Fig. 9b). Figure 9c shows that PCS benefits from the heuristics (discussed in §3.3) to speed up the search and improve the quality of the Pareto-front by discovering new points faster than searching on the unparameterized search space.

Error in Job Size Estimation. Figure 10 shows the impact of estimation error in job-sizes on the predictability and performance of PCS. As job-size estimation gets poorer, the impact on avg Pred_{err} follows the same trend as the Pred_{err} under FIFO (Fig. 10a). Figure 10b, compares the avg JCTs of AFS with no error in job-size estimation to PCS with varying estimation error. PCS is still within $1.05\times$ of AFS. This is because as long as the job is mapped to the correct queue, the error in estimating its size has limited impact on performance.

Sensitivity of Pareto-optimal configurations. To evaluate the sensitivity of Pareto-optimal configurations, we evaluate configurations discovered for workload-1 assuming 60% load on a system actually running at 80% load. Figure 10c shows that while the exact trends do not hold when the estimated workload is a mismatch, 75% of the configurations are within 10% of the closest Pareto-optimal point.

6 Discussion

Generalizability of PCS. In this paper, we realized PCS for ML workloads, however, it is designed as a generic job scheduling framework and the core insights (e.g., utilizing WFQ to realize multiple trade-off points, bounded preemption to provide predictability, etc) still hold across different scheduling scenarios. We tease apart different aspects of PCS's current realization and discuss their broader applicability. (1) Providing JCTpred. JCTpred can be computed if a job's demand function or simply put, its size is either known or can be estimated. There are several scheduling scenarios, beyond ML, where this requirement holds. Prior work has looked at estimating job sizes for requests in microservice deployments [51, 102], network flows [27, 58], compute tasks in data processing clusters [13, 21, 50] and I/O requests in storage clusters [40, 41]. In some scenarios, like network (co)flow scheduling, the demand function is simple: $\frac{\text{estimated (co)flow size}}{\text{allocated bandwidth}}$, while in other scenarios it may be more complicated and costly to determine. (2) Search process. The current simulation-aided search process is meant to be triggered on coarser timescales, assuming workloads are stable and predictable on shorter timescales. This is true for ML workloads as highlighted in §2 but also for some workloads beyond ML [48, 50]. If workload changes are highly dynamic, the search process may not be able to keep-up. This opens up an interesting avenue for future research to tailor the search process for such workloads.

Resource Heterogeneity. To handle resource heterogeneity (e.g., different GPU types), PCS can reuse an existing solution: Gavel [71], which makes a GPU scheduling policy heterogeneity-aware. It supports hierarchical policies with weighted-fairness across entities and FIFO scheduling within an entity. The different parameters of WFQ (e.g., number of queues, weights etc.) map elegantly to these primitives. Once an operator chooses a WFQ configuration, PCS can convert it into an optimization problem that Gavel can solve for.

Sophisticated prediction techniques. Using more complicated prediction techniques is orthogonal work. We posit that future arrivals, the main source of unpredictability, may be difficult to take into account in the prediction decision given that various attributes about them are unknown. For instance, a future job's demand function and its arrival time cannot be determined before it actually arrives. Our emphasis is on making scheduling *predictable* and rely on a simple prediction strategy instead.

Other use-cases of JCTpred. In addition to the use-cases discussed in §2, JCTpred can be used to co-design AutoML app schedulers (e.g., Hyperband [57]) with the underlying system; based on the predicted completion time, the app scheduler can decide to prioritize certain DNNs/trials over another. This can be framed as a bi-level optimization problem where the end goal is to find the most promising DNN hyper-parameters in the quickest time. This will require widening the prediction interface to allow users the option of cancelling and making shadow reservations. Beyond ML workloads, JCTpred can facilitate user applications in i) replica selection strategies (e.g., MittOS [40]), and ii) optimizing the right parallelism and placement for network-bound data processing tasks [29, 69].

Deciding between Pareto-optimal choices. Exposing trade-offs as Pareto-optimal choices can help operators to make informed choices by narrowing down the possibilities. We still, however, rely on the operator's ability to decide between them. One potential strategy is to elicit user preferences via surveys and averaging them to come up with a cluster wide trade-off point. Allowing individual users to pick different preferences on a per job basis, however, can result in cross-user conflicts which may be difficult to resolve. We leave picking preferences on a per-job basis as future work.

7 Related work

Scheduling Systems. A large body of work emphasizes on intelligent GPU scheduling for DNN workloads, considering metrics such as minimizing average job completion times [37,44,55,87,98], maximizing fairness [14,62,93], cluster efficiency [44,55,94] and average DNN accuracy [76,100]. They use preemption based techniques to achieve their objectives; we show in this paper that preemption is detrimental to predictability.

PCS can benefit from system-level techniques, such as elastic scaling [44,74], efficient GPU preemption [85,92–94], DNN throughput profiling [61, 79], job/AutoML app size estimation [62, 76], and sharing-safety [103] used in these systems. However, in contrast to them, PCS focuses on predictability by limited use of preemption and offering flexibility to cluster operators in choosing various trade-off points between predictability and other traditional objectives. Gavel [71] also translates different scheduling policies to optimization objectives but does not cover predictability and only finds a point solution for each objective while PCS allows operators to choose from a range of Pareto-optimal choices.

Multi-queue Scheduling. A broad category of schedulers use the idea of queue-based scheduling [10, 18, 25, 26, 28, 37, 68, 70] in different contexts to achieve performance related goals. We borrow ideas from these techniques. For example, like 2D [28], we also create queues based job size variation within a queue. Similarly, our limited use of multiplexing is inspired by the FIFO-LM scheduler [26]. However, these techniques opt for a fixed strategy in creating queues, mapping jobs to queues and assigning weights (e.g., Baraat [26] and Tiresias [37] only use 2 queues) and will be limited to offering a fixed trade-off between objectives.

Adaptive Schedulers. There are multiple recent examples of empirical, adaptive cluster management. For example, Self-Tune [52] applies reinforcement learning techniques to automatically update the cluster management policy based on periodic cluster status updates. Decima [65] uses simulations to learn optimal scheduling algorithms for data processing. SWP [104] uses a simulation guided approach to find optimal bandwidth scheduling decisions. These works show the efficacy of using simulated environments to learn system decisions. Our strategy is inspired by them.

Predictable Scheduling. Predictable scheduling and delay guarantees has been studied in broader contexts. Weirman et al [91] classify different scheduling policies based on the variation in the slowdown experienced by jobs. Other studies [22, 43] look at the benefits of providing delay information to users and understand how much delay is tolerable. CFQ [15] defines predictability as a job's FFT, similar to Themis. However, FFT is prone to variation itself as new jobs arrive [50].

8 Conclusion

In this paper, we called for providing predictability as a first order consideration in GPU scheduling systems. Our inspiration comes from real-world systems that provide their users with predictions (e.g., estimated delivery dates). Our solution, PCS, provides predictability while balancing other considerations like performance and fairness. It comprises of a bi-directional preference interface to empower cloud operators in making informed trade-offs between multiple objectives. To realize these trade-offs, PCS uses WFQ in unique way coupled with a simulation-based strategy to discover Pareto-optimal WFQ configurations. Our results show the flexibility of PCS in achieving a wide range of operator objectives, offering a first step towards predictable scheduling in a practical way.

Acknowledgements

We thank our shepherd, Jonathan Mace, the anonymous OSDI reviewers, Varun Gupta, all the members of the NAT lab and D.O.C.C lab for their invaluable feedback and suggestions that helped improve this work. This research was partially supported by NSF grants CNS-1815046 and CNS-2106797.

References

[1] AFS-Simulator. https://github.com/chhwang/schedsim.

[2] Amazon. http://www.amazon.com.

[3] CloudLab. https://www.cloudlab.us.

[4] GPT. https://openai.com/blog/chatgpt.

[5] Philly traces. https://github.com/msr-fiddle/philly-traces.

[6] Temu. http://www.temu.com.

[7] Uber. https://www.uber.com.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[9] G. Appenzeller, M. Bornstein, and M. Casado. Navigating the high cost of ai compute. April 2023.

[10] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *Prox. Usenix NSDI*, 2015.

[11] M. Baranishyn, B. Cudmore, and T. Fletcher. Customer service in the face of flight delays. *Journal of Vacation Marketing*, 2010.

[12] P. Behnam, J. Tong, A. Khare, Y. Chen, Y. Pan, P. Gadikar, A. Bambhaniya, T. Krishna, and A. Tumanov. Hardware-software co-design for real-time latency-accuracy navigation in tinyml applications. *IEEE Micro*, 2023.

[13] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[14] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[15] C. Chen, W. Wang, S. Zhang, and B. Li. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017.

[16] Z. Chen, W. Quan, M. Wen, J. Fang, J. Yu, C. Zhang, and L. Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[17] D. Cheng, J. Rao, C. Jiang, and X. Zhou. Resource and deadline-aware job scheduling in dynamic hadoop clusters. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 956–965. IEEE, 2015.

[18] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[19] R. Cordingly and W. Lloyd. Enabling serverless sky computing. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2023.

[20] D. Crankshaw, G. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2020.

[21] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2014.

[22] B. G. Dellaert and B. E. Kahn. How tolerable is delay?: Consumers' evaluations of internet web sites after waiting. *Journal of interactive marketing*, 1999.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

[24] B. Derakhshan, A. Rezaei Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2020.

[25] F. R. Dogar, L. Aslam, Z. A. Uzmi, S. Abbasi, and Y. Kim. Cam01-3: Connection preemption in multiclass networks. In *IEEE Globecom 2006*, pages 1–6. IEEE, 2006.

[26] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[27] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla. Is advance knowledge of flow sizes a plausible assumption. In *Proc. USENIX NSDI*, 2019.

[28] A. B. Faisal, H. M. Bashir, I. A. Qazi, Z. Uzmi, and F. R. Dogar. Workload adaptive flow scheduling. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. ACM, 2018.

[29] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[30] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[31] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *Proc. USENIX ATC*, 2021.

[32] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. USENIX Association, March 2011.

[33] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[34] I. Giagkiozis and P. J. Fleming. Pareto front estimation for decision making. *Evolutionary Computation*, 2014.

[35] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[36] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 2023.

[37] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[38] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[39] A. Haeberlen, L. Phan, and M. McGuire. Metaverse as a service: Megascale social 3d on the cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2023.

[40] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proc SOSP*, 2017.

[41] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[42] M. Hertzum and K. Hornbæk. Frustration: Still a common user experience. *ACM Trans. Comput.-Hum. Interact.*, jan 2023. Just Accepted.

[43] M. K. Hui and L. Zhou. How does waiting duration information influence customers' reactions to waiting for services? *Journal of Applied Social Psychology*, 1996.

[44] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. USENIX Association, 2021.

[45] R. Ibrahim. Sharing delay information in service systems: a literature survey. *Queueing Systems*, 2018.

[46] A. Isenko, R. Mayer, and H. Jacobsen. How can we train deep learning models across clouds and continents? an experimental study. *arXiv preprint arXiv:2306.03163*, 2023.

[47] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng. A case for task sampling based learning for cluster job scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[48] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *SIGCOMM Comput. Commun. Rev.*, 2015.

[49] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.

[50] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[51] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[52] A. Karthikeyan, N. Natarajan, G. Somashekar, L. Zhao, R. Bhagwan, R. Fonseca, T. Racheva, and Y. Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[53] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[54] F. Lai, Y. Dai, H. V. Madhyastha, and M. Chowdhury. ModelKeeper: Accelerating DNN training via automated training warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[55] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu. Allox: Compute allocation in hybrid clusters. In *Proc. EuroSys*, 2020.

[56] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu. Dcloud: deadline-aware resource allocation for cloud computing jobs. *IEEE transactions on parallel and distributed systems*, 27(8):2248–2260, 2015.

[57] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2017.

[58] W. Li, X. He, Y. Liu, K. Li, K. Chen, Z. Ge, Z. Guan, H. Qi, S. Zhang, and G. Liu. Flow scheduling with imprecise knowledge. In *Proc. USENIX NSDI*, 2024.

[59] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[60] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[61] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[62] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[63] D. H. Maister et al. *The psychology of waiting lines*. Citeseer, 1984.

[64] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19. Association for Computing Machinery, 2019.

[65] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 2019.

[66] S. M. Miller. Predictability and human stress: Toward a clarification of evidence and theory. Advances in Experimental Social Psychology. Academic Press, 1981.

[67] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[68] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[69] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. Network scheduling aware task placement in datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16.

[70] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, 2009.

[71] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. USENIX Association, 2020.

[72] P. Ngatchou, A. Zarei, and A. El-Sharkawi. Pareto multi objective optimization. In *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, pages 84–91, 2005.

[73] H. Ning, H. Wang, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand. A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges. *IEEE Internet of Things Journal*, 2023.

[74] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2020.

[75] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018.

[76] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[77] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, 2019.

[78] A. Perkiomaki. How estimated delivery dates (edds) enhance user experience and drive transactions for ecommerce brands. September 2023.

[79] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021.

[80] N. Salari, S. Liu, and Z. M. Shen. Real-time delivery time forecasting and promising in online retailing: When will your package arrive? *Manufacturing & Service Operations Management*, 24(3):1421–1436, 2022.

[81] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[82] R. Singhal and A. Verma. Predicting job completion time in heterogeneous mapreduce environments. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[83] D. R. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*.

[84] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021.

[85] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[86] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. Association for Computing Machinery, 2013.

[87] S. Wang, O. J. Gonzalez, X. Zhou, T. Williams, B. D. Friedman, M. Havemann, and T. Woo. An efficient and non-intrusive gpu scheduling framework for deep learning training systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[88] S. Wang, Y. Wang, and K. Lin. Integrating priority with share in the priority-based weighted fair queuing scheduler for real-time networks. *Real-Time Systems*, 2002.

[89] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. USENIX Association, 2022.

[90] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[91] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to higher moments of conditional response time. *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[92] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[93] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[94] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[95] Y. Xu, A. Khare, G. Matlin, M. Ramadoss, R. Kamaleswaran, C. Zhang, and A. Tumanov. Unfoldml: Cost-aware and uncertainty-based dynamic 2d prediction for multi-stage classification. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2022.

[96] G. Yang, C. Shin, J. Lee, Y. Yoo, and C. Yoo. Prediction of the resource consumption of distributed deep learning systems. *Proc. ACM Measurement and Analysis of Computing Systems*, 2022.

[97] Z. Yang, Z. Wu, M. Luo, L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. Sifei, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[98] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[99] Q. Yu, G. Allon, and A. Bassamboo. How do delay announcements shape customer behavior? an empirical study. *Management Science*, 63(1):1–20, 2017.

[100] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[101] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. USENIX Association, 2020.

[102] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023.

[103] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang. Hived: Sharing a gpu cluster for deep learning with guarantees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2020.

[104] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson. Swp: Microsecond network slos without priorities. *arXiv preprint arXiv:2103.01314*, 2021.

[105] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[106] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 1999.

A Artifact Appendix

Abstract

We have open sourced our implementation of PCS at https://github.com/TuftsNATLab/PCS/tree/osdi24-artifact. The repository contains jupyter notebooks to recreate figures from the paper as well as scripts to simulate schedulers used in the paper including PCS, FIFO, Tiresias, Themis, and AFS. There are also instructions for running the testbed on Cloudlab.

Artifact Checklist

- **Algorithms:** Both the simulator and testbed implement PCS as well as FIFO, Tiresias, Themis, and AFS.
- **Hardware:** Experiments on a physical cluster require 16 type c240g5 nodes on CloudLab. The nodes should be running Ubuntu 22.04.
- **Setup Instructions:** Setup instructions are available in TESTBED.md and the system prerequisites and setup sections of README.md. TESTBED.md provides instructions on setting up PCS on a CloudLab cluster as noted above as well as setting it up locally, provided the system has CUDA compatible GPUs.
- **Runtime** The testbed experiments take approximately a day (multiple hours per configuration) and the simulations take < 1 hr.

Description

Hardware Dependencies

We ran experiments on 16 c240g5 type nodes on CloudLab. We tested our system on Ubuntu 22.04 with Python 3.10.12 that should be accessilbe with `python3`. For more details, see TESTBED.md and the system prerequisites section of README.md.

Software Dependencies and Hardware Configuration

Software dependencies and hardware configuration can be installed using a script provided in the artifact. For details, see TESTBED.md.

Datasets

Experiments use either job workloads to generate traces, or directly use traces. Workloads consist of a distribution of arrival times, service times, min/max GPUs, and number of jobs per application. These are in the workloads and traces folders respectively. Traces from 6 virtual clusters (vc id's: 0e4a51, b436b2, 6214e9, 6c71a0, 2869ce, and ee9e8c) from Philly [49] are used in some simulator experiments. PCS configurations (PCS-JCT, PCS-bal, PCS-pred) used for each experiment can be found in the PCS_configs folder.

The testbed experiment train a VGG16 model using the CIFAR-10 dataset [53] which is automatically downloaded when the testbed experiment is started.

Experiment workflow

There are two kinds of experiments in the repo - simulation and testbed. For each of these experiments there are additional jupyter notebooks for plotting the results and creating the graphs used in this paper. The data needed to generate graphs used in the paper can be created with shell scripts described in the README of the respository.

Simulation experiments are run from a workload that is generated by a known distribution of job characteristics. We sample from these distributions and generate a workload that matches the provided cluster load, number of GPUs, and number of apps. The workload is then run through the simulator using a selected scheduling strategy.

The testbed is run on CloudLab using a ray cluster that has been modified to implement PCS.

Running Additional Simulations

We provide in the artifact the ability to choose and evaluate different PCS configurations, beyond the ones covered in the paper, for a set of workloads and traces. The user can also modify different experiment parameters (e.g., number of GPUs, number of apps, load). For more details, see reproduce.py and sim.py.

Optimizing Resource Allocation in Hyperscale Datacenters: Scalability, Usability, and Experiences

Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir,
Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang

Meta Platforms

Abstract

Meta's private cloud uses millions of servers to host tens of thousands of services that power multiple products for billions of users. This complex environment has various optimization problems involving resource allocation, including hardware placement, server allocation, ML training & inference placement, traffic routing, database & container migration for load balancing, grouping serverless functions for locality, etc.

The main challenges for a reusable resource-allocation framework are its usability and scalability. Usability is impeded by practitioners struggling to translate real-life policies into precise mathematical formulas required by formal optimization methods, while scalability is hampered by NP-hard problems that cannot be solved efficiently by commercial solvers.

These challenges are addressed by Rebalancer, Meta's resource-allocation framework. It has been applied to dozens of large-scale use cases over the past seven years, demonstrating its usability, scalability, and generality. At the core of Rebalancer is an expression graph that enables its optimization algorithm to run more efficiently than past algorithms. Moreover, Rebalancer offers a high-level specification language to lower the barrier for adoption by systems practitioners.

1 Introduction

In Meta's private cloud, millions of servers are deployed to host tens of thousands of services, powering dozens of products that serve billions of users. In such a complex environment, we routinely encounter a wide variety of resource allocation problems. The following are some real examples:

- *Hardware placement* [30]: Decide when and where to add or remove server racks in a datacenter while balancing competing goals such as staff work schedule, power budget, spread across fault domains, and colocation for proximity, e.g., ML training servers requiring high-bandwidth network.
- *Service placement* [32]: Decide on the allocation of servers to services while spreading each service across fault domains and optimizing the matching between services and server generations, as different services exhibit varying performance across server generations.
- *Service sharding* [25]: For sharded services like databases, determine how to migrate data shards both within and across datacenter regions in response to real-time load changes, while ensuring spread across fault domains and preventing too many concurrent changes that could destabilize the system.
- *Traffic routing* [5]: Route traffic from billions of users to geographically distributed datacenters while optimizing network latency and datacenter load.
- *Locality groups* [35]: Intelligently partition serverless functions into groups, with each server executing functions exclusively within its designated group. The objective is to enhance locality, maximize hits in the JIT code cache, and balance CPU and memory usage across servers.

All these problems have a common pattern where we want to *assign* a set of *objects* to a set of *bins* in a way that optimizes specific *objectives* while meeting certain *constraints*.

Mixed-Integer Programming (MIP) is a well-known technique that can be used to solve such assignment problems. In this approach, assignment variables, denoted as v_{ij}, take the value of 1 if object i is assigned to bin j, and 0 otherwise. A MIP solver determines optimal values for these variables, optimizing the specified objectives while adhering to the given constraints.

While MIP is conceptually straightforward and has been explored in systems research [11, 21, 41], reports of its usage in large-scale production systems are limited. For instance, in the context of load balancing for sharded services, Google's Slicer [2] relies on hand-crafted heuristics, and Azure Service Fabric [21] unsuccessfully experimented with MIP before eventually adopting simulated annealing. Meta's own sharding system, Shard Manager [25], initially used hand-crafted heuristics for several years, but it became too complex to add new features. Eventually, it adopted the framework described in this paper.

The limited adoption of MIP in solving large-scale systems problems is primarily due to its two major limitations: usability and scalability.

Usability. Despite MIP's conceptual simplicity, most systems practitioners lack the training to translate real systems' complex, and sometimes ad hoc, policies into MIP's precise mathematical formulas. To address this usability gap, DCM [38] enables developers to express constraints using familiar SQL statements, while Wrasse [34] employs a domain-specific language for the same purpose. However, industry adoption of these approaches is yet to be reported, and thus, their generality remains unverified. Moreover, they do not sufficiently address the scalability challenge described below.

Scalability. Recall that assignment variables v_{ij} represent whether object i is assigned to bin j. Therefore, an assignment problem has $|\mathbb{O}| \times |\mathbb{B}|$ variables. Here, $\mathbb{O}$ is the set of objects, $\mathbb{B}$ is the set of bins, and $|\mathbb{O}|$ and $|\mathbb{B}|$ are their sizes. Although one can define assignment variables differently to reduce their number, the overall input size of a MIP problem formulation is still $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$; we omit the details here. In our large-scale private cloud, assignment problems involve up to several million objects and 100,000 bins. However, even approximate MIP solvers would struggle with $\mathcal{O}(10^{11})$ assignment variables, not to mention that most assignment problems are NP-hard.

1.1 Overview of Rebalancer

To address the usability and scalability challenges, we have developed Rebalancer, a generic assignment problem solver. Over the past seven years, it has been applied to dozens of diverse use cases, demonstrating usability, scalability, and generality.

There are three core issues in designing a solver: *model specification*, *model representation*, and *model solving*. To address the scalability challenge, Rebalancer represents the model as a directed acyclic graph (DAG), reducing the model size from $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$ to $\mathcal{O}(|\mathbb{O}| + |\mathbb{B}|)$. Moreover, the graph representation is a fundamental reason why Rebalancer's optimized local search can solve the model more efficiently than past local search algorithms [1]. To address the usability challenge, Rebalancer supports declarative model specification through intuitive APIs, automatically translating high-level specifications into the graph representation for efficient processing. We discuss each of these topics below.

Model specification. To address the usability challenge, Rebalancer employs a three-step approach to incrementally elevate the level of abstraction for ease of use. First, it introduces essential modeling constructs, such as *dimensions* (representing objects and bins' attributes like CPU and memory) and bin *scopes* (representing server, rack, datacenter, etc). Next, it provides an *expression API* to expose commonly used expressions (e.g., Max and Sum) for transformations on these constructs, as well as recursively on other expressions. Finally, leveraging these expressions, it exposes a high-level *spec API* implementing dozens of common objectives and constraints.

The high-level spec API enables developers to effortlessly construct assignment problems (see Figure 1). For example, CapacitySpec allows developers to specify constraints such as memory usage on a server with 64GB memory, and GroupCountSpec can be used to specify that each server rack can host at most one replica of a database shard, ensuring spread across fault domains.

If no existing high-level spec meets a developer's needs, they can always utilize the expression API to define a new spec, which can then be exposed for other developers to reuse in the future. In practice, across dozens of use cases supported by Rebalancer, 85% of their constraints and objectives are implemented by directly reusing existing specs, without resorting to the expression API. Moreover, we demonstrate that it is relatively easy to define a new spec using the expression API.

Model representation. After a developer leverages the specs to define an assignment problem, Rebalancer translates it into an *expression graph* $\mathcal{G}$. Each node in $\mathcal{G}$ represents an expression constructed from its child expressions. Due to careful choices in model constructs and common expressions, the size of $\mathcal{G}$ is scalable, $\mathcal{O}(|\text{objects}| + |\text{bins}|)$, as opposed to $\mathcal{O}(|\text{objects}| \times |\text{bins}|)$ in the MIP problem formulation.

Model solving. Since the class of assignment problems are in general NP-hard, our goal is to find a high-quality solution within a reasonable amount of time. For small problems, Rebalancer translates the expression graph into a MIP model and solves it with commercial solvers. However, large problems at Meta are either too large for commercial solvers or have tight deadlines, e.g., due to real-time load balancing requirements. To address these limitations, Rebalancer implements an optimized local search algorithm. The graph representation is the fundamental reason why this algorithm is more efficient than past local search algorithms [1, 29, 33, 34].

Contributions. This paper makes the following contributions.

- **First of a kind.** To our knowledge, Rebalancer is the first framework that solves a wide range of assignment problems and has been extensively validated through production usage in hyperscale infrastructure.

- **Model specification.** Seven years of hands-on experience with dozens of use cases has allowed us to iteratively improve and arrive at the current modeling constructs and high-level specification API. Although other usability-enhancing abstractions have been proposed before, their generality has not been validated through widespread production usage.

- **Model representation.** Due to careful choices in model constructs and expressions, the size of the expression graph is scalable, $\mathcal{O}(|\text{objects}| + |\text{bins}|)$, as opposed to $\mathcal{O}(|\text{objects}| \times |\text{bins}|)$ in the MIP problem formulation.

- **Model solving.** The expression graph is also an important distinction that enables us to design a highly scalable algorithm for model solving, utilizing optimized local search on top of the graph.

In subsequent sections, we will describe model specification, model representation, and model solving, in that order.

2 Model Specification

Rebalancer allows developers to easily specify an assignment problem as a composition of a predefined set of high-level *specs*. Figure 1 shows such an example.

2.1 Modeling Constructs

To ensure reusability of the specs, Rebalancer defines a set of modeling constructs that can flexibly represent user requirements:

- **Dimensions.** A *dimension* is a mapping of each object and bin to a number. For example, the memory dimension of a server (a bin) specifies the server's memory capacity, while the memory dimension of a task (an object) specifies the amount of memory needed to run the task. Dimensions can also represent complex relationships. For example, we can define a prohibitedObjects dimension, where an object takes a value of 1 or 0, depending on its assignability to a bin.
- **Bin hierarchy.** Rebalancer uses *scopes* to represent the hierarchical structure of bins. For example, the *datacenter* and *rack* scopes represent servers in a datacenter or rack. A scope divides bins into sets called *scope items*. For example, under the *rack* scope, the scope items $rack_1$ and $rack_2$ represent the set of servers in those specific racks.
- **Object hierarchy.** Similar to scopes and scope items for bins, an *object partition* is an aggregation of objects, which may not be necessarily disjoint. Each set in the partition is referred to as a *group*. For example, in the context of cluster management, all tasks are *partitioned* into jobs and a job is a *group* of tasks that run the same executable.

As a concrete example of using these constructs, Figure 1 defines two dimensions, CPU and storage, to model resources; a rack scope as a fault domain; and a job partition where each group comprises tasks that run the same executable.

2.2 Definition of Utilization

Next, we describe an important concept called *utilization* of bins or scope items. It encompasses, but is more general than the intuitive concept of a server's CPU or memory utilization. Formally, given an object-to-bin assignment and a dimension D, the *utilization* of a bin b_j with respect to D, denoted $\mathsf{util}(b_j, D)$, is defined as the sum of dimension values of all objects assigned to the bin. That is,

$$\mathsf{util}(b_j, D) = \sum_{o_i \in \mathbb{O}} D(o_i) \cdot v_{ij}, \tag{1}$$

where $D(o_i)$ is the dimension value of object o_i, and v_{ij} takes value 1 if o_i is assigned to b_j and 0 otherwise.

```
// Do not exceed CPU and storage capacity.
addConstraint(CapacitySpec(
    scope="server", dimension="CPU"))
addConstraint(CapacitySpec(
    scope="server", dimension="storage"))

// A rack hosts no more than one task per job.
addConstraint(GroupCountSpec(
    scope="rack", dimension="ObjectCount",
    partition="job", limit=1))

// Balance CPU and storage usage across servers.
addObjective(BalanceSpec(
    scope="server", dimension="CPU"))
addObjective(BalanceSpec(
    scope="server", dimension="storage"))
```

Figure 1: Using Rebalancer's high-level specs to specify the objectives and constraints for assigning tasks (objects) to servers (bins).

For example, a bin's utilization with respect to the ObjectCount dimension is simply the number of objects assigned to it. Note that utilization can also be defined for a scope item with respect to a group of objects. For example, $\mathsf{util}(\mathsf{rack}_r, \mathsf{job}_j, \mathsf{ObjectCount})$ counts the number of job_j's tasks deployed on servers in rack_r.

Flavors of utilization. The basic definition of utilization in Eqn 1 is inadequate for certain intricate scenarios. For instance, in the process of migrating a data shard from a source server to a destination server, it may be necessary to first load the shard on the destination, ensuring its healthy operation, before removing it from the source. Throughout this transition period, which might be prolonged when involving substantial data copying, the shard consumes resources on both the source and destination servers. Another scenario involves modeling system stability requirements, such as restricting the number of objects moved in and out of a bin.

To accommodate these complexities, we introduce additional utilization variants. Utilization of bin b_j is the sum of contributions from a set of objects. In Eqn 1, this set consists of objects *currently* assigned to bin b_j, referred to as AFTER. Additionally, we define sets like INITIAL, representing the objects initially assigned to b_j, and $\mathsf{STAYED} = \mathsf{INITIAL} \cap \mathsf{AFTER}$, denoting the initial objects that remained in bin b_j. Extending the notation to include the *temporal* set of objects contributing to utilization as $\mathsf{util}(b_j, D, \mathsf{TIME})$, we refer to it as $\mathsf{TIME}_{\mathsf{util}}$, where TIME can be AFTER, STAYED, or INITIAL. Concretely, expressions like $\mathsf{INITIAL}_{\mathsf{util}}$, $\mathsf{AFTER}_{\mathsf{util}}$, and $\mathsf{STAYED}_{\mathsf{util}}$ capture the utilization by the sets of INITIAL, AFTER, and STAYED objects, respectively.

Through set operations on these base definitions of util, we can create derived definitions such as:

- $\mathsf{NEW}_{\mathsf{util}} = \mathsf{AFTER}_{\mathsf{util}} - \mathsf{STAYED}_{\mathsf{util}}$ which captures the utilization of *new* objects that moved into bin b_j.
- $\mathsf{OLD}_{\mathsf{util}} = \mathsf{INITIAL}_{\mathsf{util}} - \mathsf{STAYED}_{\mathsf{util}}$ which captures the uti-

lization of *old* objects that moved out of bin b_j.

- $\mathsf{ANY_{util}} = \mathsf{INITIAL_{util}} + \mathsf{AFTER_{util}} - \mathsf{STAYED_{util}}$ which captures the utilization of objects that were in bin b_j at any point in time. Note that subtracting the STAYED term avoids double counting for objects that stayed in b_j.

These utilization variants help capture complex scenarios. For instance, $\mathsf{ANY_{util}}$ can model double occupancy, while $\mathsf{NEW_{util}}$ and $\mathsf{OLD_{util}}$ can model system stability.

Internally, Rebalancer translates these utilization variants into their mathematical forms. $\mathsf{AFTER_{util}}$ is simply Eqn 1 using v_{ij} determined by the current assignment. $\mathsf{INITIAL_{util}}$ is a constant value that can be pre-computed from Eqn 1 using the initial assignment. To implement $\mathsf{STAYED_{util}} = \mathsf{util}(b_j, D, \mathsf{STAYED})$, a new dimension named D_j^{init} is introduced for each bin b_j. This dimension takes the value $D(o_i)$ for all objects initially assigned to b_j and zero otherwise. We can again use Eqn 1 with the fact that $\mathsf{util}(b_j, D, \mathsf{STAYED}) = \mathsf{util}(b_j, D_j^{\mathsf{init}}, \mathsf{AFTER})$.

2.3 Common Specs

Over the past seven years, through the process of supporting dozens of large-scale use cases, we have iteratively developed the common specs shown in Table 1. On average, an assignment problem uses seven specs, with a maximum of 14.

Table 1 highlights the reuse of many specs, with six used only once, suggesting they are developed when needed initially. The high-level spec API prioritizes ease of use, while the low-level expression API offers extensibility for new spec development. It provides different flavors of util expressions, mathematical operators (Max, Sum) for aggregation, and transformation operators (Step, Ceil, Log, and Power). Besides being user-friendly, this API enables modeling of non-linear properties, providing a more convenient alternative to crafting a MIP problem formulation from scratch. Overall, the expression API facilitates the implementation of simple specs in dozens of lines of code, and even the most complex specs can be implemented in a few hundred lines of code.

Consider, for example, the introduction of UtilIncreaseCostSpec to prioritize moves to servers with CPU utilization less than a specified threshold T_0. When all servers have CPU utilization over T_0, it favors the one with the least utilization. Using the expressions API, this is modeled in just 65 lines of code by adding the penalty expression $\mathsf{Power}(\mathsf{excessUtil}_i,\ 2)$ to the objective for every server *i*. Here, $\mathsf{excessUtil}_i = \mathsf{Max}(0,\ \mathsf{util}(\mathsf{server}_i, \mathsf{CPU}, \mathsf{AFTER}) - T_0)$.

3 Case Studies of Model Specification

In this section, we describe how to model several real world assignment problems using Rebalancer's spec language.

3.1 Hardware placement

To provide context, we first outline our infrastructure hierarchy: datacenter region→datacenter→suite→main switch board (MSB)→server row→server rack→server. Globally, there are tens of datacenter regions and each region has multiple datacenters within a few miles' radius. Each datacenter consists of four large rooms called suites. Each suite has three MSBs, each supplying power to 10K to 20K servers laid out as rows of server racks. Each rack hosts tens of servers.

A datacenter undergoes continuous evolution with the addition of new server racks and the removal of existing racks for maintenance or decommissioning. The hardware-placement problem involves computing an optimized weekly schedule for these operations, considering the staff's work schedule, ensuring hardware spread across fault domains, and adhering to capacity constraints on power, network, etc.

We model racks as *objects* and *(week, position)* pairs as *bins*, where a *position* is a physical location in the datacenter. We introduce *scopes*, such as *MSB* and *week*, where each *scope item* is a collection of bins associated with the same MSB and week respectively. Similarly, an *object partition* of racks can be defined, where each group consists of racks of the same type. In the following, we outline a small subset of objectives and constraints for hardware placement.

- **New racks.** Initially, all new racks belong to a special bin called *unassigned*. Applying ToFreeSpec on that bin ensures that new racks are assigned to certain *(week, position)*.

- **Power and network constraints.** This is achieved by using CapacitySpec at different *scopes* of the infrastructure hierarchy such as position, MSB, and suite.

- **AI Zone.** AI server racks must be placed in an *AI zone*, which is a special section of the datacenter connected by a high-bandwidth network. To enforce the placement of AI racks in the AI zone, we introduce a new dimension AiRack, which takes the value 1 for AI racks and 0 otherwise. Similarly, this dimension has a limit of 1 for AI zone positions and 0 otherwise. We then apply CapacitySpec with the AiRack dimension over the *position* scope.

- **Place certain racks in the same week.** This is achieved by putting those racks into a *group* and applying ColocateGroupSpec to the group over the *week* scope.

Overall, as rack changes occur incrementally over time and we compute a solution for each datacenter region separately, the hardware-placement problem is relatively small in size, involving hundreds of objects and thousands of bins. For these small problems, Rebalancer translates the expression graph into a MIP problem formulation and employs a MIP solver, instead of a local-search solver, to ensure high-quality results.

3.2 Service Placement

Hardware capacity in datacenters are allocated to teams responsible for different products in the form of quotas called *reservations*. Whole servers or fractions of a server's resources are assigned to reservations while adhering to all kinds of constraints. While reservations can be either global

Spec name	Description	Usage count	Lines of code
CapacitySpec	Enforce that the utilization of a scope item is within specified limits.	19	340
GroupCountSpec	Restrict utilization by objects of a group placed in the same scope item, commonly used to enforce the spread of objects across scope items. For instance, a job can have at most one task in a rack.	17	480
AvoidMovingSpec	Do not move any of the specified list of objects.	16	120
BalanceSpec	Balance utilization across scope items.	12	250
MinimizeMovementSpec	Minimize the number of objects that move into or out of a scope item.	12	90
MovesInProgressSpec	Objects specified as moving from one bin to another by the previous solver run must finish the move.	10	65
NonAcceptingSpec	Specify scope items that are not accepting incoming objects.	9	100
MinimizeBinsSpec	Minimize the number of bins utilized.	8	105
AssignmentAffinitiesSpec	Indicate that specific objects prefer specific bins.	6	90
ToFreeSpec	Free up certain bins. For example, move services out of servers that will be decommissioned.	5	70
ColocateGroupsSpec	Place objects of the same group in the same scope item, e.g., placing a job's tasks in the same rack.	5	100
GroupMoveLimitSpec	Limit how many objects of the same group (e.g., a database's replicas) can move concurrently.	4	95
AvoidAssignmentsSpec	Prevent assignments of certain objects to scope items. For example, in hardware placement, an AI zone in a datacenter only accepts AI server racks.	4	60
GroupDiversitySpec	Every scope item must get objects from at least (or at most) K different groups.	4	80
SingleGroupFailureBufferSpec	Provide additional buffer objects when a group of objects fails together. Used in service placement to ensure that services have enough servers even when a fraction of a datacenter fails.	2	145
DrainCapacitySpec	Allow draining objects from a faulty bin to other bins while respecting capacity constraints.	1	80
MoveGroupSpec	Move objects in the same group together across bins.	1	75
MinimizeNthLargestUtilization	Minimizes the utilization of scope items with the n-th largest utilization	1	85
MaximizeAllocationSpec	Maximize utilization on a set of scope items	1	65
UtilIncreaseCostSpec	Prefer moving objects to underutilized scope items.	1	65
Logical Or/And Specs	Perform a logical OR/AND of certain specs.	1	55

Table 1: List of 21 most frequently used specs out of a total of 28 specs currently supported by Rebalancer. Remaining specs are specific to their respective usecases and their descriptions involve defining concepts beyond the scope of this paper.

or regional, our discussion focuses on regional ones for simplicity. A regional reservation can comprise servers from any datacenter within the same region but not across regions. Our cluster management system treats each reservation as a dynamic virtual cluster and deploys the owner team's jobs on it.

In this service-placement problem [32], we model servers as *objects* and reservations as *bins*. Moreover, we group servers by *MSB*, *rack*, and hardware *type*, which become object partitions. Below, we describe some used objectives and constraints.

- **Capacity sufficiency.** If a reservation specifies a demand of X units for a server type Y, we fulfill it by utilizing CapacitySpec with the count dimension for each server type. The variability in performance among services on various server types can be represented as a dimension. Thus, we can optimize for assigning servers of a specific type to services that can extract optimal performance from them.
- **Spread.** GroupCountSpec ensures that servers allocated to a reservation are spread across *MSB* and *rack* partitions.
- **Stability.** As new reservation requests emerge or existing ones are updated, we run the solver to update both old and new reservations. Recomputing solutions for old reservations is necessary, as it enables the relocation of servers from old reservations to new ones, facilitating global optimization. However, moving many servers out of an old reservation, even if those servers are replaced with new ones, would cause churns to services running on those servers. We use MinimizeMovementSpec to minimize churns.
- **Fault tolerance.** For each reservation, we allocate additional buffer capacity to ensure that in case any single MSB in a datacenter region goes offline due to failure or maintenance, there is still sufficient capacity in the reservation. This requirement is modeled using SingleGroupFailureBufferSpec.

A large service-placement problem involves up to 700K servers (objects) in a datacenter region and 6K reservations (bins). We solve one such problem per region every hour. The solve frequency and associated downstream actions necessitate that Rebalancer must finish solving the problem within 10 minutes. Initially, Rebalancer converted the expression graph to a MIP problem and solved it with a MIP solver. However, recently we switched to using the faster local-search solver due to both the growing problem size and the desire to reduce the solve time to fulfill capacity change requests faster. This experience demonstrates one advantage of Rebalancer—it can take the same problem specification and flexibly decide which solver to use based on the problem size and time limit.

3.3 Service Sharding

Sharded services, such as databases, are prevalent at Meta and account for 68% of the total RPC traffic [25, 36]. They often host about 100 shards per Linux process for improved efficiency, and shards are dynamically migrated across these Linux processes to balance the load. For simplicity, we refer to each such Linux process as a "*server*", assuming one process per server. To ensure redundancy, each shard has multiple replicas, which are grouped together using *object partitions*. This problem assigns *shards* (objects) to *servers* (bins) while meeting various requirements, some of which are described below.

- **Capacity limit.** We use CapacitySpec to ensure that servers are not overloaded. Given that cross-server shard moves are not instantaneous, we use $\mathsf{ANY}_{\mathsf{util}}$ to account for double occupancy (§2.2).
- **Limit churns.** To cap the number of moves per server or per shard, we use CapacitySpec with *ObjectCount* as the dimension and $\mathsf{NEW}_{\mathsf{util}}$ and $\mathsf{OLD}_{\mathsf{util}}$ as the utilization (§2.2).
- **Region preference.** Certain shards prefer servers in specific datacenter regions because the users accessing those shards are close to those regions. This preference is modeled using AssignmentAffinitiesSpec with the *region* scope.
- **Load balancing.** To ensure that the load is balanced across servers, we use BalanceSpec with the *bin* and *region* scopes for regional and global load balancing, respectively.
- **Fault Tolerance.** To ensure that a shard's replicas are spread across various fault domains, such as rack and MSB, we use GroupCountSpec with *replica* as the partition and *rack* or *MSB* as the scope.

The largest sharding problems involve 1.8M objects and 27K bins and have a solve time limit of five minutes. Rebalancer's local search algorithm has scaled well to produce high-quality solutions for such large problems.

3.4 Message Queue Placement

Many people use Meta's messaging products. On the server side, a message queue is created for each user to store messages intended for delivery. Placing the message queue in a datacenter region close to the user reduces latency. This problem is to assign user queues (objects) to datacenter regions (bins). However, treating each user as an individual object is inefficient, so we aggregate users objects with common properties into a bundle and treat each bundle as an object. The bundles are computed offline based on properties such as proximity and connectivity of users within a bundle. The following are some supported requirements:

- **Minimize latency.** Every user bundle has a numerical affinity to each datacenter region. The affinity is equal to the negative of the average network latency to a region for users in the bundle. Utilizing AssignmentAffinitiesSpec as an objective with these affinities minimizes the total latency.
- **Colocate related user bundles.** If two bundles' users frequently communicate with each other, colocating them in the same region would reduce both latency and cross-region traffic. ColocateGroupSpec achieves this purpose, where each *group* is a set of related bundles.
- **Buffer capacity for disaster recovery.** One service level objective is that any single datacenter region can go offline without causing disruption to users. For this purpose, a traffic matrix with elements t_{ij} specifies that in the event of region i failure, a fraction t_{ij} of region i's traffic will be redistributed to region j. Rebalancer must ensure that each region has enough spare capacity to absorb the incoming redistributed traffic. This is achieved by using DrainCapacitySpec to enforce that the peak utilization of a bin with the worst case spillover traffic must be within its capacity limit.

The message-queue problem typically involves tens of thousands of objects and tens of bins, and is solved only once a week. Because of the small scale and lenient solving deadline, Rebalancer translates it into a MIP problem. The traffic-routing problem described in §1 shares some commonality with this problem, but it needs to update the global edge-to-datacenter traffic matrix every few minutes. Hence, it utilizes local search to achieve a low average runtime of 5 seconds.

3.5 Kubernetes Scheduler

When evaluating the flexibility of Rebalancer's specs using the use-case examples above, a question naturally arises: do the specs inherently cover these examples because they are designed to support them? To showcase Rebalancer's flexibility, we implement Kubernetes' scheduling policies using Rebalancer's existing specs. While Rebalancer handles load balancing of containers across machines in production, it does not handle the Kubernetes-like initial container placement in our fleet. This function was implemented in Meta's cluster manager using heuristics similar to those in Kubernetes years before introducing Rebalancer and is still in active use.

Specifically, to prepare for a direct comparison with DCM [38] in performance evaluation (§6), we implemented Kubernetes' scheduling policies listed in Figure 2 of the DCM paper. To improve usability, DCM uses SQL statements to express allocation policies and internally translates these SQL statements into a constraint satisfaction problem, which is then solved using the Google OR-tools CP-SAT solver.

In Kubernetes, container *pods* are scheduled on *nodes* (machines). We represent pods as objects and nodes as bins. Similar to DCM, our implementation schedules a batch of pods together. All unscheduled pods of the current batch are placed in a special *unassigned* bin, and we impose a ToFreeSpec constraint on that bin, forcing Rebalancer to place them on certain nodes. Resource limits, such as CPU and memory, are enforced using CapacitySpec. Affinity of pods for specific nodes is specified using CapacitySpec with a custom dimension representing affinity. Inter-pod anti-affinity for replica groups is modeled using GroupCountSpec. Fixing certain pods to nodes is achieved with AvoidMovingSpec, and so forth.

Overall, we are able to model Kubernetes' scheduling constraints using Rebalancer's existing specs in about 500 lines of code, which is comparable to 550 lines of SQL-based specification in DCM. This demonstrates that Rebalancer is flexible and its usability is comparable to DCM. We will compare the performance of DCM and Rebalancer in §6.

3.6 Other Use Cases

In addition to the examples described above, Rebalancer supports dozens of use cases at Meta, including Linux container rebalancing across servers [39]; routing traffic from globally distributed edge datacenters to main datacenters [5]; grouping serverless functions to improve locality [35]; balancing online ML training workloads across regions while considering the priority of ML workloads; minimizing the number of replicas for ML inference models or databases deployed across geo-distributed datacenters, while adhering to latency SLO and meeting varying user request rates; various sharding systems that have requirements different from the one in §3.3; assigning work tickets to engineers; and so forth.

4 Model Representation

Once an optimization problem is specified using specs, Rebalancer materializes them into an expression graph. Recall that Rebalancer's expression API supports operators such as Max and Sum for aggregation, and Step, Ceil, Log, and Power for transformation. For example, $\mathsf{Step}(x)$ evaluates to 1 if x is positive and 0 otherwise. We translate the specs into a recursive composition of expressions that reuses common expressions to obtain a compact expression graph.

4.1 Translating Specs into Expressions

We use several examples to illustrate how to translate specs into expressions. *CapacitySpec* enforces that the utilization of a resource is within specified limits. For instance, with a CapacitySpec applied to the scope server and dimension CPU, Rebalancer creates $|\mathbb{B}|$ constraints (one per server) in the form of $\mathsf{util}(\mathsf{server}_i, \mathsf{CPU}, \mathsf{AFTER}) \leq L_i$, where L_i represents the CPU limit of server_i. To model double occupancy, CapacitySpec would use ANY instead of AFTER.

MinimizeMovementSpec minimizes the movement of objects into or out of a scope item. Rebalancer adds the expression $\mathsf{util}(S_{\mathrm{out}}, \mathsf{count}, \mathsf{NEW}) + \sum_j \mathsf{util}(S_j, \mathsf{count}, \mathsf{NEW})$ to the objective, where S_{out} represents the set of bins that do not belong to any scope items, such as the *unassigned* bin in the hardware-placement example (§3.1). Note that each moving object is only counted once. Specifically, the first term captures the objects that move out of all the scope items, while the second term captures objects that move within the scope items.

GroupDiversitySpec ensures diversity in the set of objects assigned to a bin. For example, the servers (objects) assigned to a service's global reservation (bin) must come from at least k datacenter regions so that in the event of a region failure, there is at least some capacity available for the service. In this case, assuming objects are partitioned into groups G_i based on their region, Rebalancer adds $|\mathbb{B}|$ constraints (one per service) in the form of $\sum_i \mathsf{Step}(\mathsf{util}(b_j, G_i, \mathsf{count})) \geq k$. Note that the inner Step expression evaluates to 1 if bin b_j contains objects from group G_i and 0 otherwise.

4.2 Reducing Model Size

In the previous section, we discussed how to translate specs into mathematical formulas using expressions such as util. However, the direct representation of util as shown in Equation 1 is inefficient as it would lead to a problem representation size of $\Theta(|\mathbb{O}| \times |\mathbb{B}|)$. To address this scalability challenge, Rebalancer implements several efficient custom expressions that help reduce the problem's model representation to $\Theta(|\mathbb{O}| + |\mathbb{B}|)$. Below, we describe one such expression called Lookup, which is used to efficiently implement util.

Object Lookup. The insight behind Lookup is that, in most cases, an object's dimension value remains unaffected by the bin to which it is assigned. For instance, a task consumes the same amount of memory irrespective of the server on which it runs. This allows representations of utilizations for different bins and scope items to share and reuse these dimension values, reducing the problem input size by a factor of $\Theta(|\mathbb{B}|)$, resulting in an overall input size of $\Theta(|\mathbb{O}| + |\mathbb{B}|)$.

Specifically, for each such static dimension D, we establish an *object-vector*, denoted as V_D, representing a mapping from objects to their dimension values. Given an *object-vector* V_D and a scope item S_i, a Lookup represents an efficient aggregation operation over the (object, bin) pairs for bins in S_i. For example, $util(S_i, D) = \mathsf{Lookup}(S_i, V_D)$ simply aggregates the utilization across all bins in S_i with respect to D through lookup. Note that the memory usage of Lookup itself is of constant size since it only keeps references to S_i and V_D, while the representations of S_i and V_D, with sizes $\mathcal{O}(|\mathbb{B}|)$ and $\mathcal{O}(|\mathbb{O}|)$, respectively, are shared and reused across all expressions. This leads to an overall problem size of $\Theta(|\mathbb{O}| + |\mathbb{B}|)$.

Expression graph. Since constraints in Rebalancer are inequalities in the form of $f(\cdot) \leq 0$, they can be combined using the Max expression. For example, although a CapacitySpec results in $|\mathbb{B}|$ constraints (one per bin), it can be simplified into a single constraint by rewriting it as

$$\mathsf{Max}_i(\mathsf{Lookup}(b_i, V_D) - L_i) \leq 0.$$

As each constraint and objective can be written as a recursive composition of expressions, we can encapsulate an assignment problem's all constraints and objectives in a DAG $\mathcal{G}$. The nodes of $\mathcal{G}$ correspond to expressions, and each objective or constraint in the problem is a subgraph of $\mathcal{G}$; see Figure 2 for an example. For a node $v \in \mathcal{G}$, we use $\mathsf{children}(v)$ to denote the set of all nodes w such that $v \to w$ is an outgoing edge from v. The $\mathsf{type}(v)$ of a node (e.g., Max) is its mathematical operator, and $\mathsf{children}(v)$ represent its inputs. The DAG $\mathcal{G}$ is obtained from a recursive composition of these operators.

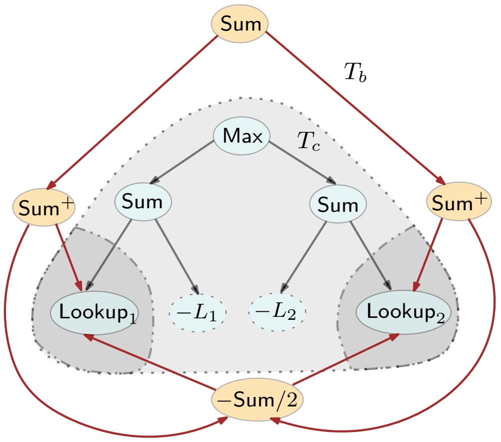


Figure 2: This *expression graph* represents a simplified version of the task-allocation problem shown in Figure 1. It aims to balance the CPU utilization across two servers, server$_1$ and server$_2$. It uses BalanceSpec on the CPU dimension as the objective (subgraph T_b) and CapacitySpec on the CPU dimension as the constraint (tree T_c). Nodes in the intersection of T_b and T_c namely Lookup$_1$ and Lookup$_2$ are reused. Constant nodes are shown in dashed circles. L_i is the CPU limit of server$_i$, Lookup$_i$ represents the lookup for the CPU dimension on server$_i$, meaning server$_i$'s CPU utilization. Sum$^+$ is a shorthand for $\max(0, \text{Sum})$

5 Model Solving

After representing an assignment problem as an expression graph $\mathcal{G}$, the next step is to solve the problem. For small problems, Rebalancer translates it into a MIP problem and solves it with a MIP solver. For large problems, Rebalancer implements its own optimized local search. The existence of the expression graph is a fundamental reason why Rebalancer's local search is more efficient than existing local search algorithms.

5.1 Using MIP Solver

To translate an expression graph into a MIP problem, each expression implements a recursive mipTranslate operation. This operation, based on the expression's type, converts it into a linear combination of binary assignment variables v_{ij} that indicates whether object i is assigned to bin j. Invoking mipTranslate on the root nodes of an expression graph yields a MIP model, which can subsequently be solved using a commercial solver. Note that the MIP model's input size is $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$, which is not scalable. Therefore, we only use the MIP solver for relatively small problems.

5.2 Graph-Assisted Local Search

In contrast to the MIP solver's all-or-nothing approach to finding the *optimal* solution, local search [1] incrementally generates a set of object *moves* that improve upon the initial assignment but without guaranteeing optimality. Each move

Algorithm 1 Local search using expression graph $\mathcal{G}$

```
while exit-conditions are not met do
    $\mathcal{L} \leftarrow$ generate_candidate_moves($\mathcal{G}$)
    for local change $\delta$ in $\mathcal{L}$ do
        obj$_\delta \leftarrow$ evaluate_moves($\mathcal{G}, \delta$)
        if obj$_\delta > 0$ then
            discard $\delta$    ▷ violates constraint or worsens objective
        end if
    end for
    $\delta^* \leftarrow \min_{\delta \in \mathcal{L}}$ obj$_\delta$    ▷ best local change
    apply_moves($\mathcal{G}, \delta^*$)
end while
```

(o_i, b_s, b_d) corresponds to reassigning object o_i from its source bin b_s to its destination bin b_d.

Although local search has been applied to assignment problems before [19, 33], the uniqueness of our approach, as highlighted in Algorithm 1, lies in its exploration of the expression graph for all its main steps: (1) generating candidate moves, (2) evaluating them, and (3) applying the best moves. In the rest of this section, we describe the main ideas that enable our algorithm to scale to millions of objects and bins.

5.2.1 Generating Candidate Moves

Because each object can potentially be moved from its current bin to any other bin, there are a total of $|\mathbb{O}| \times (|\mathbb{B}| - 1)$ candidate moves to consider. Obviously, it would be too expensive to evaluate all of them. There are two natural ways to reduce the candidate set: (1) restricting the search space to one bin at a time and finding the best moves involving that bin, or (2) restricting the search space to one object at a time and finding the best moves for that object. Rebalancer takes approach (1) because the number of bins is usually much smaller than the number of objects. With this settled, we still need to decide in which order to evaluate bins and, given a bin, how to propose candidate moves. We discuss these topics below.

Bin selection. Our insight here is to first evaluate *hot bins* that potentially can have the biggest impact on the overall objectives by moving objects into or out of these bins. The structure of the expression graph already captures what objectives are affected by which bins and by what amount. Intuitively, for example, if there is a directed path from a node v to a Lookup on bins b_1 and b_2, moving objects in and out of these bins will improve node v's value. The idea is to process the leaf nodes of $\mathcal{G}$ (such as Lookup) in a *greedy order* of their contribution to the objective. This ordering of leaf nodes gives us a sequence of sets of bins $S_v, S_w, \ldots, S_z$ and we can use these sets to infer the *hottest bin*.

Move strategies. After identifying a hot bin, Rebalancer explores different *move strategies* to move objects into and out of it. For example, the SINGLE move strategy considers moving every object in b_s to every other bin b_d exhaustively and accepts the best move. There are also variants of SINGLE, such as SINGLE_GREEDY, which accepts the first improving move, and SINGLE_RANDOM, where b_d belongs to a small sample of randomly chosen bins. A commonly used effective strategy is to first use SINGLE_RANDOM for some period of time when opportunities for improvement are abundant and later switch to using SINGLE_GREEDY when opportunities for improvement become scarce.

In addition to variants of the SINGLE strategy, Rebalancer also supports more complex strategies such as swapping objects between two bins, and using the Kernighan–Lin algorithm to identify the move-destination bin; see details in the Appendix. Finally, Rebalancer also supports custom strategies that exploit domain knowledge. For example, Shard Manager [25] uses Rebalancer to move shards (objects) across servers (bins) to balance the load. If a hot server has many small shards and a few large shards, going through the shards sequentially may spend most of the time evaluating moving small shards that have little impact on the objective. Instead, Rebalancer evaluates large shards earlier, which not only accelerates the search but also reduces the number of shard moves.

5.2.2 Evaluating and Applying Moves

The remaining components of Rebalancer's algorithm are *evaluating* and *applying* moves. When given a candidate move (o_i, b_s, b_d), a naive way to evaluate its impact is to apply the move to the initial assignment to obtain the new assignment. Then, we compute the new assignment's objectives from scratch through a full graph traversal. However, since we already have the value of every graph node under the initial assignment, few nodes might be affected by applying the candidate move. We can significantly speed up the computation by only recomputing the values for these affected nodes.

Bottom-up change propagation. To only recompute the changed nodes, we preprocess the leaf nodes in the expression graph to build a map from objects to the leaf nodes that reference them. Similarly, we build a map from bins to the leaf nodes that they affect. Given a move candidate, we use the two maps to identify a set of leaves affected by the change. We then traverse from those leaves to the roots, and the reached nodes along the way are the set of nodes whose values need to be recomputed.

Minimal computation during a node update. When recomputing the value of a changed node, iterating over all its child nodes is often unnecessary, as only a small fraction of them likely have changed. Depending on the type of the node, we can store additional information to speed up the recomputation. Below, we provide an example for the Max node, while similar optimizations exist for other node types.

For the Max node, we separately compute the maximum value of its changed child nodes (denoted as z_1) and the maximum value of its unchanged child nodes (denoted as z_2). Then, the new value of the node is simply $\max(z_1, z_2)$. To compute z_2 efficiently, we maintain a sorted list of child nodes ordered by their decreasing node value. We iterate over this list and stop at the first child node that is unchanged. This child node's value is z_2. Note that the runtime of this algorithm is proportional to the number of changed child nodes c instead of the total number ℓ of child nodes. This algorithm incurs the overhead of $\mathcal{O}(c \log \ell)$ to update the sorted child list, but it only occurs when a move is accepted and applied. In practice, the number of evaluated but rejected candidate moves often dominates.

Parallelizing Move Evaluations Since move evaluations do not affect the current state, we can use multiple threads to evaluate candidate moves in parallel before we pick the best candidate to *apply*. This improves the number of evaluations per second (evals/s) by an order of magnitude enabling local search to make faster progress. Although the exact number depends upon the problem instance, we are able to obtain roughly 150k evals/s for most large instances. For example, for a large sharding problem, parallel move evaluations resulted in 170k evals/s and 12k applied moves within the time limit of 300s. In contrast, sequential move evaluations result in 25k evals/s and 2k applied moves in the same time limit. In this case, local search was able to progress six times faster due to parallelization of move evaluations.

5.3 Identifying Equivalent Objects

In addition to focusing on hot bins to reduce the search space, for certain commonly used objectives and constraints, we can identify objects that are equivalent from a modeling perspective, thereby effectively reducing the number of objects. This optimization can be applied to both local search and MIP solvers. For example, in the problem of placing tasks on servers, all tasks that belong to the same job are equivalent since they all affect the constraints and objectives in the same way. For a set of equivalent objects, we only need to explore moves with at most one of those equivalent objects, which reduces the search space. In Rebalancer, we employ a recursive algorithm that exploits the expression graph to compute sets of equivalent objects. Various additional details about our solver algorithm can be found in the Appendix.

6 Evaluation

In this section, we evaluate Rebalancer's scalability and solution quality, and compare it with alternative approaches, such as DCM [38] and MIP partitioning techniques similar to POP [31]. Additionally, we assess the efficacy of local search techniques such as hot bin ordering.

Figure 3 shows the statistics of the problems solved by Rebalancer in production during a typical week. These tens of millions of solves span diverse use cases outlined in §3. The largest cases involve service sharding (1.8M shards, 27k servers) and service placement (700k servers, 6k reservations). We will use these application scenarios in evaluation.

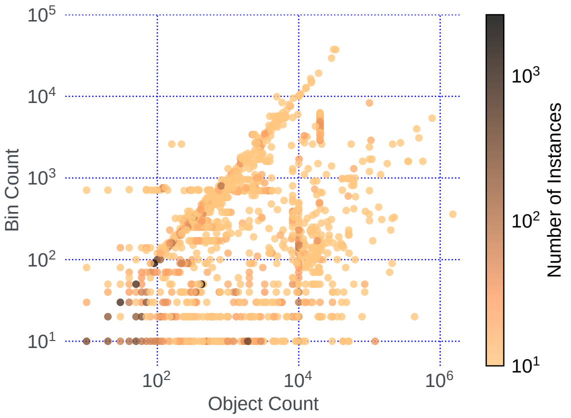


Figure 3: Samples of production problems solved by Rebalancer. The P99 (99th percentile) solve time is 16s with a problem size of 65k objects and 5k bins.

6.1 Comparison with DCM

In the context of cluster management, DCM [38] uses SQL to express policies for placing *pods* on *nodes*. It translates these SQL statements into a constraint satisfaction problem, which is then solved using the Google OR-tools CP-SAT solver. DCM's scalability bottleneck is the CP-SAT solver, which has computational intractability similar to MIP solvers.

We replicated DCM's implementation of Kubernetes scheduler in Rebalancer (§3.5). To stress-test our implementation, we impose inter-pod anti-affinities for all replica groups, a condition known to increase scheduling difficulty. Similar to evaluations in the DCM paper, we use the Azure dataset [27]. All experiments are conducted on a machine with 40 CPU cores and 256 GB of RAM.

We will quote numbers from the DCM paper instead of conducting measurements on our machines because DCM's open source code cannot run on our production machines due to the security setup of those machines. We also cannot run Rebalancer on third-party machines due to its dependencies on tools only available in our production environment. Despite the inability to directly compare their absolute performance on the same machine, their scalability trends are evident from their respective evaluations using the same dataset and implementing the same Kubernetes scheduling algorithm.

We also compare local search's SINGLE_GREEDY move type, which evaluates placing an unscheduled pod on every node, with the SINGLE_RANDOM move type, which randomly selects a fraction f of nodes as targets. If unsuccessful, it repeats the process with the remaining unexplored nodes.

Scalability. We evaluate Rebalancer under two scales: the *DCM-scale*, scheduling 50 pods per batch on clusters with 1k, 5k, and 10k nodes (similar to that in the DCM paper); and the *hyperscale*, scheduling 500 to 5k pods per batch on clusters with 10k, 50k, and 100k nodes. Using larger pod batches in the hyperscale setup is motivated by the fact that the arrival rate of pods typically increases with the cluster size. As shown in Figure 4, the P99 (99th percentile) per-pod scheduling latency of SINGLE_GREEDY is less than 35 ms for instances up to 10k nodes. However, beyond 10k nodes, SINGLE_GREEDY becomes progressively slower, while SINGLE_RANDOM continues to scale well. We have found a sample size of $f = 10\%$ (capped at 1k bins) to offer a good trade-off between solution quality and runtime.

Overall, Figure 4 demonstrates Rebalancer's significant scalability advantage over DCM. Due to inherent scalability limitations in DCM's CP-SAT solver, the largest problem tackled in the DCM paper involved scheduling 50 pods in a 10k node cluster, with close to 30 ms scheduling latency per pod. In contrast, even with a cluster size of 100k nodes (10 times that of the DCM experiment) and a batch size of 5k pods (100 times that of the DCM experiment), Rebalancer with SINGLE_RANDOM achieves a per-pod latency of 14ms.

Solution quality. We compare Rebalancer's local search with an optimal MIP solver in an experiment that places 10k pods from the Azure dataset on 500 nodes, with the objective of maximizing the number of placed pods. As shown in Table 2, neither MIP nor local search can place all pods due to their aggregate resource demand surpassing the capacity of 500 nodes. While SINGLE_RANDOM places 1.4% fewer pods than MIP when nodes are full, its runtime is 5.2 times faster.

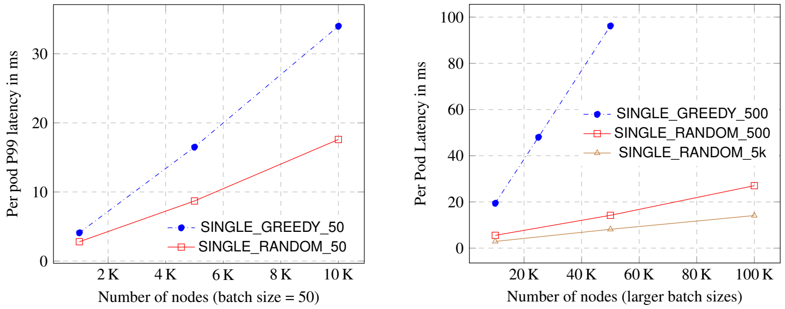
Test case	Optimal MIP solver	SINGLE GREEDY	SINGLE RANDOM
Azure dataset	94.6% (31ms)	92.8% (8ms)	93.2% (6ms)
Pathological N=10	100% (209ms)	97.1% (0.6ms)	97.8% (0.9ms)
Pathological N=100	N/A (timed out after 600s)	97.4% (26.5ms)	97.7% (22.4ms)

Table 2: Percentage of placed pods and the scheduling latency.

To evaluate local search in a scenario where it is unlikely to perform well, we design a pathological case parameterized by N, with $31N$ pods and $50N$ nodes, each with 32GB memory. Pod memory requirements follow an exponential distribution over 5 groups: the first $16N$ pods with 2GB memory, the next $8N$ with 4GB, the next $4N$ with 8GB, the next $2N$ with 16GB, and the last N pods with 32GB. The only way to schedule all pods is when the total used memory on every node is precisely 32GB. Due to the nature of local search, finding this uniquely optimal solution is unlikely. As shown in Table 2, local search can place more than 97% of the pods, while being over 200 times faster than MIP. Additionally, as depicted in Figure 5, Rebalancer finds a solution where the memory usage across nodes is well balanced.

6.2 Comparison with Partitioned MIP

POP [31] improves scalability by partitioning a large MIP problem into smaller ones and solving them independently, coordinating solutions as needed. We have implemented a



(a) DCM-scale problems. (b) Hyperscale problems.

Figure 4: Per-pod scheduling latency of our Kubernetes implementation.

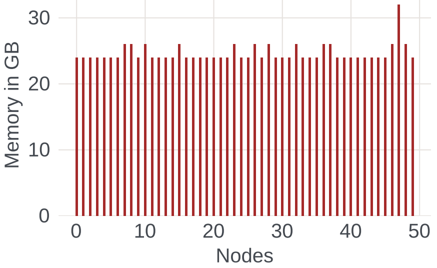


Figure 5: Memory usage across 50 nodes for the pathological case.

POP-like partitioned-MIP solver in Rebalancer in addition to its existing local-search and MIP solvers.

In Table 3, we compare local search with partitioned MIP on the service placement problem (§3.2) using our production data. Here, local search uses a combination of move types, including SINGLE_RANDOM, SINGLE_GREEDY, and SWAP. Local search is up to four times faster, meeting all the service-level requirements such as spread and capacity sufficiency, while its solution quality is less than 0.6% worse than that of partitioned MIP. Due to local search's scalability, even though we used partitioned MIP in production for a period of time, we eventually switched to local search (§7).

Problem Size (objects × bins)	Local Search	Partitioned MIP	Relative Gap
700k × 5.7k	184s	376s	0.43%
710k × 4.8k	214s	350s	0.56%
568k × 4.7k	151s	455s	0.21%
645k × 4.5k	146s	557s	0.11%

Table 3: Problem sizes and runtimes of service placement. The "*relative gap*" denotes the difference in objective values between local search and partitioned MIP.

6.3 Local Search's Individual Techniques

Next, we evaluate local search's individual techniques.

Expression graph $\mathcal{G}$ scales linearly. We validate this using three production instances of the service sharding problem, with 71k, 152k, 289k objects each and roughly 1.5-2k bins. The expression graph's memory usage grows almost linearly from 1.7GB to 3.2GB and 6.2GB.

Move evaluations are fast. As can be seen from the pod scheduling example (Figure 4), evaluating all possible 50k moves for a pod takes only 100ms. This indicates that Rebalancer can evaluate up to 500k moves per second. The techniques detailed in §5, such as bottom-up traversal of changed nodes, help achieve this speed; without them, move evaluations would be an order of magnitude slower.

Hot bin ordering is effective. A *hot bin* is one that contributes the most to the objective, and Rebalancer processes bins in the order of their hotness. As shown in Figure 6, processing bins in the order of their hotness reduces the objective value more quickly. This results in higher solution quality when the search time is capped for large problems.

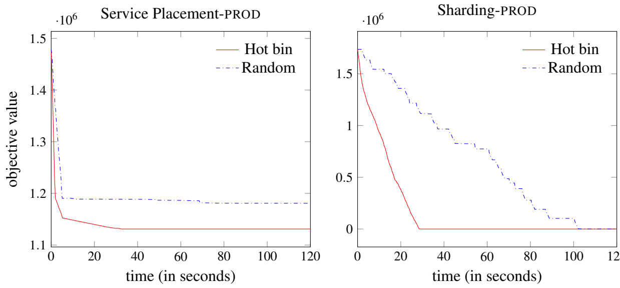


Figure 6: Comparing *hot bin* selection and *random bin* selection using service placement and service sharding.

7 Experiences and Limitations

In this section, we share some learnings from using Rebalancer and discuss its limitations.

7.1 Alternative Approaches for Optimization

We summarize three categories of approaches to solving optimization problems and make our recommendations.

Approach 1: *ad hoc heuristics*. This approach directly implements heuristics to support resource-allocation policies as code. It is the most widely used approach as it is easy to start with. However, as policies grow in complexity over time, adding new ones becomes increasingly difficult, and the solution quality tends to decrease due to the intricate balance required between different policies.

Approach 2: *formal problem specification solved by a formal solver*. Systems utilizing MIP, such as Flux [10], belong to this category, but there are other formal methods as well, such as network flow optimization. Rebalancer with a MIP solver in the backend also fits into this category. To address the challenge that most systems practitioners are unfamiliar with formal problem specification, a higher level of abstractions can be introduced, for example, DCM [38] using SQL and Rebalancer using a high-level specification language. However, a formal method's solver often lacks scalability, posing a hard barrier to its application at hyperscale.

Approach 3: *formal problem specification solved by a systematic-heuristics solver*. This category includes systems that formulate problems using MIP but solve them using simulated annealing or local search. Rebalancer using a local-search solver in the backend falls under this category. The usability issue can be solved by raising the level of abstraction, similar to that for Approach 2. While good scalability is a strength of this method, its weakness lies in producing suboptimal solutions compared to formal solvers.

Recommendation: Based on our experiences, we make recommendations as follows. First, systems requiring short and predictable latency in resource allocation decisions should use ad hoc heuristics, even though they are hard to maintain and evolve. Schedulers for high-throughput, short-lived batch jobs fall under this category. Second, systems needing high-quality solutions for small to medium-sized problems should leverage formal solvers for their optimal solutions. Lastly, for hyperscale systems not requiring real-time decisions, we recommend approach 3 over approach 1, because it is much easier to add or evolve resource-allocation policies with approach 3. One advantage of Rebalancer is its ability to support both Approaches 2 and 3 using the same high-level specification, and seamlessly switch the backend solver as needed, depending on the problem scale and solve time constraint.

7.2 Experiences with Alternative Approaches

Among the three alternatives described in the previous section, our choice for service placement (3.2) evolved from Approach 2 to Approach 3, while our choice for service sharding (3.3) evolved from Approach 1 to Approach 3. We discuss these experiences below.

Service placement. The service-placement problem [32] was initially modeled using Rebalancer's high-level specification language and solved with a MIP solver. This is because initially the problem size was still manageable for MIP, the 20-minute solve time service-level objective (SLO) was sufficient, and we were (overly) worried about the solution quality.

However, as more services and machines were added to the fleet, MIP's solve time became problematic. Meanwhile, the solve time SLO was reduced from 20 minutes to 10 minutes due to the need for faster reactions to capacity change requests. We continued optimizing the MIP solver, for example, by grouping equivalent objects to reduce the number of decision variables. These incremental optimizations bought us some time, but MIP still constantly fell behind on scalability. Other issues included the MIP solver having not only unpredictable execution times but also occasionally running into infeasibility due to numerical precision issues. Debugging and fixing these elusive problems in production under time pressure were recurring pain points for engineers.

To scale MIP, we implemented a POP-like [31] partitioned MIP solver (§6.2). Initially, it performed well in both solve time and reliability. Managing hundreds of smaller subproblems meant that a few failing MIP solves would not have a fleet-wide impact. However, new requirements, such as stacking more services on bigger machines, increased the problem scale by an order of magnitude. At this point, building a partitioned MIP model, which in the worst case uses $|\text{objects}| \times |\text{bins}|$ assignment variables, was no longer practical.

This finally forced us to explore local search. With some tuning, the local search solver achieved both good solution quality and fast solve times. As shown in Table 3, the solution-quality difference between local search and partitioned MIP is less than 0.6%. The lesson for us is that, despite having the local search technology, our unwavering faith in MIP's optimality led us on a lengthy detour to reach our current state.

Service sharding. While service placement started with Approach 2 and converged to Approach 3, the service sharding system Shard Manager, went through the opposite direction, switching from Approach 1 to Approach 3.

When Shard Manager started with ad hoc heuristics more than a decade ago, Rebalancer did not exist at that time. As Shard Manager became widely adopted by many applications, its load-balancing algorithm became overly complicated. For example, it supported multiple-dimension balancing across CPU, memory, and storage, enforced rate limiting, and considered regional and global locality. Unsurprisingly, this complexity led to frequent issues where some servers were overloaded while others were underutilized. Shard Manager struggled to balance the load because the heuristics implementing sometimes conflicting policy requirements were not robust.

The team started rewriting Shard Manager with yet another heuristic implementation. It represented the topology (region, datacenter, power domain, rack, server) as a tree and enforced resource constraints across all levels of the topology. However, achieving good load balancing proved challenging even with a clean implementation. Iterative tuning of the heuristics required constant code changes that might not lead to positive outcomes and often became dead code later.

Eventually, the heuristic-based new prototype was abandoned, and the team switched to exploring Rebalancer with local search. The usability benefits were immediate, as it was much easier to experiment with different load-balancing algorithms by changing a few lines of high-level specification in Rebalancer. The main challenges were solution quality and scalability when solving problems with millions of objects and tens of thousands of bins within the five-minute solve time SLO. These requirements are well met, and in production, 90% of the solves actually finish within 10 seconds.

7.3 Experiments with Simulated Annealing

Recall that Rebalancer's internal architecture decouples problem representation from problem solving. There is a common abstraction all solvers (e.g. local search, MIP) inherit from which can be extended to support experimentation with new kinds of solvers. In the past, we experimented with a solver based on the simulated annealing algorithm. One common problem faced by local search is finding a locally optimal solution that does not lead to a globally optimal solution. This happens when there is no sequence of moves which strictly improve the objective at each step. In order to get out of a local optimum, the sequence of moves would have to temporarily decrease the objective quality. This is not contemplated in local search, but is one of the features of simulated annealing.

However, we found our implementation of simulated annealing to not be practical at all for production-scale problems. It did not beat the performance of local search in any experiments, neither in terms of quality nor run time. At our scale, the number of possible object moves at any point is large (e.g. $\mathcal{O}(10^{11})$ combinations for a problem with several million objects and 100k bins), and the vast majority of these moves are not helpful. We found that it is important to exploit the structure of the objective function and aggressively prioritize which moves to evaluate. This is primarily done by the hot bins optimization in local search. Our implementation of simulated annealing did not exploit this structure, and blindly evaluated moves with equal probability, regardless of the shape of the objective. For this reason, we believe that further research into heuristics to reduce the search space would be needed to make simulated annealing practical. This is challenging because the heuristic would not only have to select *good* moves, but also the right *bad* moves which decrease the quality but are likely to lead to an improved quality later on.

Recall that to ensure allocation stability, it is often desired to limit or minimize the number of object moves needed to improve the assignment. We found this challenging to enforce in an algorithm such as simulated annealing, which greedily makes moves that barely improve or even decrease the quality of the objective. In contrast, local search is able to maximize the objective improvement at each individual step, finding a shorter and more optimized sequence of moves.

7.4 Evolution of Rebalancer as a Library

Originally, Rebalancer was designed as a standalone executable that took an input file describing the model in a custom format. This initial design quickly became hard to maintain as it required the service invoking Rebalancer to carefully craft the input. At that point, we decided to create a strongly typed API for programmatically defining models. As a result, with the help of the compiler and runtime sanity checks, the interface-related maintenance overhead drastically reduced. The implementation of this interface was an inflection point for the adoption of the project, as it became intuitive enough to be used by many teams at Meta.

There still remained a question of whether to make Rebalancer a service or a library. We decided to make Rebalancer a library for two main reasons. First, services invoking Rebalancer need to collect and feed Rebalancer with potentially a large amount of input data, which can be done more efficiently via a library API. Second, providing a multi-tenant service is difficult, as different Rebalancer use cases can heavily contend with each other due to their high memory and CPU usage. Currently, if a particular use case still prefers to operate Rebalancer as a single-tenant service, they could do so by wrapping the library with an RPC interface, but we have not seen that in practice.

Therefore, today Rebalancer is implemented as a library that gets compiled into each project that depends on it. The resulting binary has predictable behavior, which does not change unless it is recompiled with a newer version of the code. Different usecases have their custom logic to collect the input and setup an assignment problem using Rebalancer's specification language (See Figure 1 for an example). The usecases then specify the choice of solver (MIP or Local Search) and invoke the core solver which builds the expression graph, solves the problem and returns a solution.

7.5 Handling Multiple Objectives

Each objective specifies a weight and a priority. Rebalancer combines all objectives with the same priority using a weighted sum. Finding appropriate weights for competing objectives (e.g., objA, objB) is done by first normalizing them so that their values are comparable and then selecting multiplicative weights based on their relative importance in the problem domain. Some use cases provide strict priorities for objectives, and Rebalancer ensures that it does not regress on higher-priority objectives when solving for lower-priority objectives.

Recall that local search only makes moves that strictly improve the objective value and is generally more stable. However, when using MIP, if there are two solutions with exactly the same objective value, Rebalancer may choose one of them arbitrarily, causing instability across multiple solves. In such cases, we typically add a MinimizeMovement goal that disincentivizes moves which do not strictly improve the objective.

7.6 Debugging Solver Behavior

The majority of engineering time in setting up a model goes into debugging the behavior of the solver, which includes ranking the objectives by their importance, identifying conflicting constraints, etc. Without proper tools, this process requires a deep understanding of the internals of the solver, which only engineers working on the core of Rebalancer have. Over time we identified the common questions that helped modelers understand the solver's behavior and we built a specialized UI tool for answering them: Rebalancer-explorer. At a high level, Rebalancer-explorer can be compared to debugging tools such as Whyline [24] for post-facto analysis.

Under the hood, Rebalancer-explorer loads the expression graph representation in memory and reuses many of the algorithms mentioned earlier making it extremely fast to evaluate formulas on demand. At a high level, the UI consists of:

- *Table view* to inspect objects, bins, their dimensions and their membership in partitions and scopes.

- *Tree view* to inspect the underlying expression (sub)-graph corresponding to various user provided specs.
- *Solution comparison view* to inspect the goal and constraint values for different solutions. By default, it shows a comparison of initial and final solutions.
- *Timeline navigation view* to inspect the sequence of moves performed by local search at each step and the associated objective improvements.

One example scenario is to answer questions such as *why did solver not move an object to/from a given bin*. We can use the solution comparison view to compare the goal/constraint values between the solver-generated solution, and an ad hoc solution where a specific object is moved to/from the given bin. In this case, the UI gives real-time feedback of why that move is not good – perhaps it breaks a constraint, or makes the goal value worse.

7.7 Limitations of Rebalancer

Some problems can be modeled with MIP but cannot be modeled with Rebalancer because they do not fit the abstraction of assigning objects to bins. One such example is to assign network traffic flow to links, where Rebalancer cannot model a sequence of dependencies in the link topology. However, Rebalancer 's low-level expression API and expression graph are generic enough to support these problems while providing a significant boost for usability. Consequently, we extended the expression API and expression graph to create a more flexible framework, enabling the modeling of MIP problems beyond assignment problems.

8 Related work

There is a rich body of work in the systems research community that uses optimization problem formulations for different resource allocation settings (e.g. [2–9, 11, 13, 14, 16–18, 20, 22, 25, 28, 32, 34, 37–42]). Among these, Rebalancer shares design goals with Wrasse [34] and DCM [38] domain-specific language (DSLs) for resource allocation problems.

Comparison with existing DSLs. Wrasse [34] also uses an object-bin abstraction but their specification language is limited to a small set of properties such as resource capacity but for example does not support many other important properties listed in Table 1 such as spread, balance, affinities etc. Moreover, their GPU-based solver is tightly coupled with these properties making it hard to extend. On the other hand, DCM [38] allows a user to specify constraints and goals using SQL-like queries which are then fed into off-the-shelf solvers. As discussed before, DCM's ability to scale is limited by the intractability of underlying constraint solvers.

Other relevant systems. These systems fall in one of two categories. They either use some hand-crafted heuristics or use a commercial MIP solver. Examples which use heuristics include cluster management [4, 7, 43], application sharding [2, 25], and container reallocation [33]. The paper [33] uses variation of local search to move containers across physical machines but its scalability is limited by the fact that it uses $|\mathbb{O}| \times |\mathbb{B}|$ decision variables. Examples where commercial MIP solvers are used include capacity reservation [32] and in cluster managers [11, 41]. Rebalancer, through its specification language and its ability to translate to MIP models can help set up and solve (using Xpress and Gurobi) a version of many of the above problems at a comparable scale.

There also has been some recent progress on solving hyperscale allocation problems using MIP models. POP [31] presented a technique to decompose large scale assignment problems into small ones and combine their solutions to solve the bigger problem. However, POP requires that resources (objects) are *fungible* and clients (bins) should not prefer one object over others by a large amount. These do not always hold for assignments problems we encounter, for instance, in the case of service placement, it is common for services to request specific server types. This is the reason why our POP-like partitioned MIP solver required some additional techniques.

Work on the machine reassignment problem. A set of relevant work are the papers [12, 19, 26] from the 2012 ROADEF/EURO Challenge [29]. This challenge was designed to solve a *machine reassignment problem*, which is about reassigning processes to machines to satisfy some goals and constraints such as load balancing (see [29] for details). These papers design heuristics to solve such problems and there are some broad similarities with the ideas used in Rebalancer's local search. However, it is unclear if their techniques generalize beyond the relatively small scale and type of problems that were used in the contest.

9 Conclusion

In this paper, we discuss the design and implementation of Rebalancer, a generic framework that solves a diverse set of hyperscale assignment problems. Rebalancer decouples problem specification from optimization by defining a compact graph representation, simplifies problem specification with a high-level language, and designs a scalable optimization algorithm. Finally, we shared our experiences and lessons learned from evolving solutions for service placement and service sharding.

10 Acknowledgements

We acknowledge the contributions made by all past and present members of Algorithmic Optimization, RAS and Shard Manager teams at Meta towards evolution of Rebalancer over the last seven years. In particular, we would like to thank Aravind Narayanan, Raj Rajendran, Sahil Deshpande, Apurva Samudra and Pavan Kumar for their contributions to Rebalancer. We also thank all the reviewers and our shepherd Philip Levis for their insightful comments.

References

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 739–753, 2016.

[3] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. *ACM SIGARCH Computer Architecture News*, 40(1):61–74, 2012.

[4] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

[5] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles(SOSP'19)*, pages 430–446, 2019.

[6] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[7] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni Matteo Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, pages 177–192, 2019.

[8] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX Annual Technical Conference*, pages 1–14, 2019.

[10] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[11] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the thirteenth EuroSys conference*, pages 1–13, 2018.

[12] Haris Gavranović, Mirsad Buljubašić, and Emir Demirović. Variable neighborhood search for google machine reassignment problem. *Electronic Notes in Discrete Mathematics*, 39:209–216, 2012.

[13] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378, 2013.

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[15] Gurobi. Gurobi Optimizer [online]. 2023. URL: https://www.gurobi.com/resources/chapter-2-resource-assignment-problem.

[16] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: Vm allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 845–861, 2020.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[19] W Jaśkowski, Marcin Szubert, and Piotr Gawron. A hybrid mip-based large neighborhood search heuristic for solving the machine reassignment problem. *Annals of Operations Research*, 242:33–62, 2016.

[20] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[21] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Karthik Kambatla, Vamsee Yarlagadda, Íñigo Goiri, and Ananth Grama. Ubis: Utilization-aware cluster scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 358–367. IEEE, 2018.

[23] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[24] Amy J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, 2004.

[25] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles(SOSP'21)*, pages 553–569, 2021.

[26] Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Comparing solution methods for the machine reassignment problem. In *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 782–797. Springer, 2012.

[27] Microsoft. Azure public dataset. https://github.com/Azure/AzurePublicDataset, 2017.

[28] Pulkit A Misra, Íñigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *USENIX Annual Technical Conference*, pages 799–811, 2017.

[29] H Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: the roadef/euro challenge 2012, 2016.

[30] Aravind Narayanan, Elisa Shibley, and Mayank Pundir. https://engineering.fb.com/2020/09/08/data-center-engineering/fault-tolerance-through-optimal-workload-placement/, 2020.

[31] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[32] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. RAS: continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, pages 505–520, 2021.

[33] Bo Qiao, Fangkai Yang, Chuan Luo, Yanan Wang, Johnny Li, Qingwei Lin, Hongyu Zhang, Mohit Datta, Andrew Zhou, Thomas Moscibroda, et al. Intelligent container reallocation at microsoft 365. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'21)*, pages 1438–1443, 2021.

[34] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–12, 2012.

[35] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew McFague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 231–246, 2023.

[36] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios

Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[37] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[38] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 827–844, 2020.

[39] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matt Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 787–803, 2020.

[40] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[41] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[42] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[44] Juan Pablo Vielma. Mixed integer linear programming formulation techniques. *SIAM Review*, 57(1):3–57, 2015. doi:10.1137/130915303.

[45] FICO Xpress. Xpress Optimizer [online]. 2023. URL: https://examples.xpress.fico.com/example.pl?id=assignmentgr.

Appendix A Additional Details on Rebalancer's Local Search

The input to the local search algorithm that powers Rebalancer is an initial assignment $\mathcal{A}_0$ and the compact representation of goals and constraints as the expression graph $\mathcal{G}$. Furthermore, in the rest of this section, we assume that the following preprocessing steps have been done on $\mathcal{G}$.

- *Compute and store current values of each node*: This can be done by a simple recursive algorithm on all the roots $r_i \in \mathcal{G}$, where the value of a node Z_v is computed using the values of its children.
- *Compute and store the current upper and lower bounds for each node:* This can again be done by a simple recursive algorithm. For a node v, we use Z_v^{ub} and Z_v^{lb} to denote its upper and lower bounds, respectively.
- *Compute and store the potential of each node:* The *potential* of a node v is the difference between its current value and its lower bound (i.e., $Z_v - Z_v^{\text{lb}}$). Additionally, for each node $v \in \mathcal{G}$, we maintain a sorted order of its $\mathsf{children}(v)$, sorted by their potentials. We say that a subgraph rooted at v is *optimal* if v's potential is zero.
- *Compute and store the* affected bins *of each node:* Recall that a leaf node $\ell \in \mathcal{G}$ (such as ObjectLookup) is parameterized by a set of bins S_ℓ and changes to contents of S_ℓ will potentially change the value at node ℓ. We refer to the set S_ℓ as *affected bins* which can be recursively computed for all $v \in \mathcal{G}$ as $S_v = \{\bigcup S_w \mid w \in \mathsf{children}(v)\}$. (See also Figure 2).

A.1 Restricting the search space

One important step in any local search algorithm is to generate a neighboring set of candidate solutions. To get to a candidate solution, Rebalancer employs the notion of a *local change*, denoted δ, which is a set of ordered tuples (o_i, b_s, b_d) and where each tuple denotes the change in some o_i's assignment from some b_s to some other b_d (or alternatively, the "movement" of o_i from b_s to b_d). We refer to each tuple in δ as a *move*. So, δ is simply a set of moves. We will use the term *applying the local change* to describe the process of updating $\mathcal{A}$ with the moves in δ and denote it as $\mathcal{A} \oplus \delta$.

It is easy to see that given any two assignments $\mathcal{A}$ and $\mathcal{A}'$, there exists a set of moves δ such that $\mathcal{A}' = \mathcal{A} \oplus \delta$. So, now, the question remains as to how we can systematically generate candidate sets of moves. First, observe that even if we restrict ourselves to *single moves*—i.e., local changes where we explore moving a single object from some source to some destination bin—there are $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$ choices. As discussed before, this is unacceptable for large problems, and so we must find another way to restrict the search space.

There are two natural ways to do this, *i)* restrict the search space to *one bin at a time* and find the best moves involving that bin, *ii)* restrict the search space to *one object at a time* and find the best moves for that object. Note that in both these cases the order in which we explore the bins (resp. objects) is extremely important or we may spend too much time exploring moves that yield little improvement. In Rebalancer, we restrict the search space at the *bin* scope. This decision is primarily motivated by the fact that the number of bins is usually much smaller than the number of objects, so restricting the search by bins makes faster local progress. Now, even with the choice of exploring one bin at a time, there still remains two main questions: *i)* What is the order of bins to consider, *ii)* Given a choice of bin, how to generate the set of local changes. We will answer the second question below and return to the first one in the next section.

A.1.1 Move Types

Given a bin b_s, we can generate local changes that move objects to and from b_s. To methodically generate them, we use the notion of a *move type* that describes the semantics of these changes. For example, a SINGLE move type considers moving every object in b_s to every other bin b_d. There can be other variants of SINGLE such as SINGLE_GREEDY which accepts the first improving single move, and SINGLE_RANDOM where b_d belongs to a small sample of randomly chosen bins. Although in our experience just using *single move types* often suffices, it is worth noting that the notion of a move type is highly customizable and can exploit problem-specific domain knowledge. In the following, we describe this in a greater detail.

Overall, each move type generates a set of local changes $\mathcal{L}$, evaluates each of the resulting candidate solutions (i.e., for each $\delta \in \mathcal{L}$, evaluates $\mathcal{A}' = \mathcal{A} \oplus \delta$), and if it exists, returns the δ that improves the objective the most. The logic to generate $\mathcal{L}$ varies based on the move type and the following are some commonly used ones.

- **SINGLE:** Given a source bin b_s, it tries moving every object in b_s to every other bin b_d. That is, $\mathcal{L}$ is the set of all $\delta_{i,d}$, where $\delta_{i,d} = \{(o_i, b_s, b_d)\}$ is a move set with exactly one move.
- **SINGLE_GREEDY:** Similar to SINGLE, but instead of evaluating moving every object in b_s to every other b_d, it considers the objects in some order and only considers the moves with the subsequent object if no improving move was found with the previous one.
- **SWAP:** For a source bin b_s and all other destination bins b_d, it tries swapping every object in b_s with every object in b_d .
- **KL_SEARCH:** inspired by Kernighan–Lin algorithm [23]. Given a source bin b_s, and for every possible destination bin b_d, construct the KL-move set δ_k iteratively as follows. Let $\delta_0 = \emptyset$ be an initially empty move set. The move sets at the end of i-th ieration δ_i is best of $\delta_{i-1} \cup m_i$ where m_i is a single move from b_s to b_d or from b_d to b_s. The iteration stops once moves involving all objects in b_s and b_d have been tried. The KL-move set δ_k is the best of all δ_i.

In fact, there are more complex move types in Rebalancer, but we do not go into its details here due to space constraints.

A.1.2 Identifying equivalent objects

In addition to restricting the search space to explorations from a bin, depending on the set of objectives and constraints, it might be possible to identify objects that are equivalent from a modeling perspective. For example, in the TASKS-ON-SERVERS example, all tasks that belong to the same job are equivalent, since they all affect the constraints and objectives in the same way. Observe that if we identify the sets of equivalent objects, then we can cut down the search space even further by only exploring moves with at most one object from an equivalence class. In Rebalancer, we employ a recursive algorithm that exploits the expression graph $\mathcal{G}$ to compute sets of equivalent objects.

Consider again the TASKS-ON-SERVERS example. There we would ideally want a solver to automatically identify that all tasks that belong to the same job are *equivalent*, since they all affect the objectives and constraints identically. In fact, such a feature can be quite powerful in further reducing the search space, since it allows us to discard moves that are equivalent while exploration (two moves are equivalent if they both move equivalent objects from a source bin to destination bin).

In Rebalancer, the intuition described above is formalized using the notion of *equivalent objects*. Formally, let $\mathcal{A}$ be any feasible solution of the given problem, and for two objects o_i and o_j, let $\mathcal{A}'$ be the assignment obtained by swapping their bins, i.e., $\mathcal{A}' = \mathcal{A} \setminus \{(o_i, \mathcal{A}(o_i)), (o_j, \mathcal{A}(o_j))\} \cup \{(o_i, \mathcal{A}(o_j)), (o_j, \mathcal{A}(o_i))\}$. Then, o_i and o_j are deemed *equivalent* if all constraints and objective expressions evaluate to the same value for both $\mathcal{A}$ and $\mathcal{A}'$. Alternatively, one can also, slightly informally, think of o_i and o_j as equivalent if, for every bin b and for a problem expressed in native form (i.e., using assignment variables), the modified problem that results from replacing every variable of the form $v_{i,c}$ with variable $v_{j,c}$ is mathematically equivalent to the original problem.

Ideally, we want compute an optimal set of equivalent objects (i.e., a set $P = \{I_1, \cdots, I_k\}$ of minimum size and where each object is part of one of the I_js and each I_j is a set of equivalent objects), however this is computationally hard. Hence, we use a greedy recursive algorithm which once again exploits the expression graph $\mathcal{G}$. The main component in our algorithm is for every node in $\mathcal{G}$ to maintain some information about what sets of objects are equivalent with respect to it. For example, in the case of ObjectLookup, two objects are equivalent w.r.t. it if they have the same value in the corre-

Algorithm 2 **Finding sets of equivalent objects**

```
$P \leftarrow \{\mathbb{O}\}$                    ▷ initially, all objects are considered equivalent
repeat for each node $v \in \mathcal{G}$
    if children($v$) is empty then,
        return set of equivalent objects w.r.t. $v$
    end if                           ▷ every leaf stores sets of objects eq. w.r.t. it
    for each child $w \in$ children($v$) do
        $P_w \leftarrow$ set of equivalent objects w.r.t. $\mathcal{G}_w$
        for each set $I \in P_w$, where $I \notin P$ do
            $\mathcal{E}_1, \ldots, \mathcal{E}_k \leftarrow$ sets in $P$ that intersect $I$
            Create $2k$ sets $O_1, \ldots, O_k$ and $N_1, \ldots, N_k$,
              where $O_i = \mathcal{E}_i \setminus I$ and $N_i = \mathcal{E}_i \cap I$
            $P \leftarrow \{O_1, \ldots, O_k, N_1, \ldots, N_k, \mathcal{E}_{k+1}, \ldots, \mathcal{E}_{|P|}\}$
        end for
    end for
until all nodes in $\mathcal{G}$ are explored
return $P$
```

sponding V. Once we have this information for every node in $\mathcal{G}$, we can then have an algorithm that starts by initially considering all objects as equivalent, and then recursively splits this set while ensuring that no two objects that are deemed non-equivalent by a node is part of the same set. Algorithm 2 describes how to do this. While it is possible to prove that this algorithm does produce a set of equivalent sets (although not necessarily of minimum size), a formal proof is beyond the scope of this paper.

A.2 Computing the hottest bin

While move types help in generating local changes, the more important question is: *what order of bins should one look for moves from?* To answer this, we introduce the notion of *hottest bin*. A bin is considered *hottest* when, given an objective and a current assignment, moving objects to or from this bin reduces the objective value the most. At a high-level, during each iteration we find the hottest (a.k.a. most broken) bin and attempt to fix it by making local changes as dictated by the *move types*, and continue the search until no progress can be made. Algorithm 3 describes the high-level local search algorithm used in Rebalancer. (Timeout handling has been omitted for simplicity.) Observe that there are three performance sensitive components in our algorithm, *i)* finding the hottest bin (line 7), *ii)* given a local change, evaluating the objective value (line 12), and *iii)* applying a local change (line 17). In the rest of this section, we describe each of these components in more detail.

Algorithm 3 Local Search Algorithm

Input: Objects $\mathbb{O}$, bins $\mathbb{B}$, expression graph $\mathcal{G}$, initial assignment $\mathcal{A}_0$

```
 1: $\mathcal{A} \leftarrow \mathcal{A}_0$                              ▷ set current assignment
 2: $anyProgress \leftarrow$ true
 3: while $anyProgress$ do
 4:     $anyProgress \leftarrow$ false
 5:     $explored \leftarrow \emptyset$
 6:     while $explored \neq \mathbb{B}$ do
 7:         $b_{hot} \leftarrow$ find_hottest_bin($\mathcal{G}$) $\notin explored$
 8:         $currProgress \leftarrow$ false
 9:         for $moveType$ in MoveTypes do
10:             $\mathcal{L} \leftarrow$ local changes using $b_{hot}$ and $moveType$
11:             for local change $\delta \in \mathcal{L}$ do
12:                 obj$_\delta \leftarrow$ evaluate_changes($\mathcal{G}, \delta$)
13:                 remove $\delta$ from $\mathcal{L}$ if it violates any con-
                    straint or worsens objective, i.e., obj$_\delta > 0$
14:             end for
15:             if $\mathcal{L}$ is not empty then
16:                 $\delta^* \leftarrow \min_{\delta \in \mathcal{L}}$ obj$_\delta$         ▷ best local change
17:                 apply_changes($\mathcal{G}, \delta^*$)
18:                 $\mathcal{A} \leftarrow \mathcal{A} \oplus \delta^*$                  ▷ update assignment
19:                 $currProgress \leftarrow$ true
20:                 $anyProgress \leftarrow$ true
21:                 break
22:             end if
23:         end for
24:         if $currProgress$ is false, then add $b_{hot}$ to ex-
            plored
25:     end while
26: end while
```

Intuitively, the hottest bin is one that can potentially improve the most from local moves, but it is not obvious how to find such a bin. Prior work explored the concept of *bin potential* which, for a bin b, is the difference in the current objective value and the value of the objective after removing all objects in b [19]. Although a reasonable metric, it only works for objectives that can be improved by moving objects out of a bin, but not when objects need to be moved in, such as what is required when enforcing minimum capacity. Moreover, finding a candidate bin can be expensive as it requires computing the contribution of every bin and taking the maximum.

The bin ranking algorithm used in Rebalancer exploits the expression graph $\mathcal{G}$ and works regardless of whether the objectives improve by moving objects in or out of them. Moreover, it does not need to compute the contribution of every bin and terminates as soon as the hottest bin has been established. Recall that each node $v \in \mathcal{G}$ directly (leaf nodes such as Lookup) or indirectly (internal nodes such as Max) affects a set S_v of bins. The idea is to process the leaf nodes of $\mathcal{G}$ in a *greedy order* of their contribution to the objective. This ordering of leaf nodes gives us a sequence of sets of bins $S_v, S_w, \ldots, S_z$ and we can use these sets to infer the *hottest bin*. Indeed, if each leaf node affected exactly one bin, the hottest bin would be the one corresponding to the first leaf in this ordering. However, leaf nodes such as ObjectLookup may affect many bins, so we need a way to break ties.

We do this by maintaining an initially empty list of *hottest candidates* in a data structure H called *incremental priority*

Algorithm 4 find_hottest_bin

```
1:  Incremental priority queue $H \leftarrow \emptyset$
2:  $iter \leftarrow$ objective root
3:  **if** valid cache exists, **then** restore $H$ and $iter$ from cache
4:  **while** $iter \neq end$ **do**
5:      $v \leftarrow$ node at $iter$
6:      **if** $H$ has a unique best element **then**
7:          **return** $\mathsf{top}(H)$ as the hottest bin
8:      **end if**
9:      **if** $v$ is a leaf or *does not need expansion* **then**
10:         $S_v \leftarrow$ affected bins at $v$
11:         $\mathsf{update}(H, S_v)$                ▷ update queue $H$
12:     **end if**
13:     advance $iter$ to the next node in pre-order
14: **end while**
15: **return** $\mathsf{top}(H)$ breaking ties at random
```

queue. The items in H are bins whose priority is defined using a *series of sets* of descending priority. Given two bins $b, b' \in H$ and the series of sets associated with each, b has a higher priority than b' if it appears in a set before b' does. If both b, b' appear for the first time in the same set, then the second set breaks the tie, and if the second is also the same, then the third is used, and so on. The series of sets can be fed into the data structure one set S at a time with $\mathsf{update}(H, S)$. One implementation of this data structure is to maintain a map of bins with the indices of the sets in which they appear. This map is sorted by a custom *compare* function which orders the bins in the right way. For example, if we had three sets $\{1\colon \{b_p, b_q\}, 2\colon \{b_p, b_q, b_r\}, 3\colon \{b_p, b_r\}\}$, the sorted map will be $b_p\colon \{1,2,3\}, b_q\colon \{1,2\}, b_r\colon \{2,3\}$. In this case, once all the sets have been processed, b_p will be the hottest bin.

Algorithm 4 describes how we compute the hottest bin by traversing the expression graph $\mathcal{G}$ in *pre-order*. We start with the objective root and process children recursively in the order of decreasing potentials and exit as soon as the hottest bin is found (line 7). We also perform some natural optimizations to reduce the number of nodes traversed. For example, we do not recursively expand nodes that have achieved their bound values, and also do not expand if every node in the subgraph rooted at it affects the same set of bins (line 9). Observe that the algorithm saves the progress each time and if possible resumes from a valid cached state. If we invalidate the cache after applying each local progress, then the ordering of hot bins is *dynamic*, otherwise it is *static*. Indeed dynamic ordering often leads to a better solution quality (at the cost of recomputing the ordering every time), but there are cases when a static ordering is sufficient.

A.3 Evaluating and applying candidate solutions

The remaining two important components of our local search algorithm are *evaluating* and *applying* a set of moves δ. A naive way of evaluating or applying a change δ would be to just recompute the modified assignment $\mathcal{A}' = \mathcal{A} \oplus \delta$ and recompute the value of all nodes by a full recursive traversal. Indeed this is quite inefficient as it requires traversing and recomputing values for all nodes of the graph $\mathcal{G}$ when the number of nodes affected by the change δ is likely quite small. Since every node of the graph already stores its value w.r.t. to the current assignment, if we can find a way to identify the child nodes whose values need to be recomputed and combine them with values that were not modified, we can significantly speed up evaluate and apply operations. Observe that *applying* of moves updates the internal state of graph $\mathcal{G}$ (namely node values, bounds, ordering of children by their potential), whereas *evaluating moves* does not modify the graph. This distinction is important as it allows us to achieve even faster running times for *evaluate* operations. Below, we describe some ideas that make this possible.

Bottom-up propagation of changes.

Once the expression graph $\mathcal{G}$ is built, we can also preprocess the leaf nodes to build a map $\mathcal{M}_o$ from objects to the leaf nodes that reference them. Similarly, we build a map $\mathcal{M}_b$ from bins to the leaf nodes that they affect. Recall that each local change δ is a set of moves (o, b_s, b_d) that denotes moving object o from b_s to b_d. Given this, we can iterate over the set of moves in δ, and use the maps $\mathcal{M}_o, \mathcal{M}_b$ to compute a set of leaves L affected by the change δ. Starting from the leaves in L, we traverse the graph bottom-up (from leaves to the roots) using the *incoming edges* at every node. The set of nodes reachable in this way is precisely the minimal set of nodes whose values need to be recomputed. (See also Figure 2.)

Minimal computation at a node v.

While recomputing value of a changed node v, iterating over all the nodes in $\mathsf{children}(v)$ can be unnecessarily expensive especially when only a small number of child nodes may have been updated. Depending on the type of the expression node, we can store some additional information that makes these updates significantly faster. Here we give an example for the Max node; similar optimizations exist for other node types. For a Max node, we maintain a *sorted list of children* by their value. We first iterate over all the *updated child nodes* and take the maximum of their new values; suppose that this value is z_1. Next, we iterate over the list of sorted children nodes and stop at the first node that was not updated. Let the value of that node be z_2. Now, it is not hard to establish that the updated value of this Max node is $\max(z_1, z_2)$. Observe that as a result of this process, updating the value of this node took $O(|Z|)$ time, where $Z \subseteq \mathsf{children}$ is the set of updated children, as opposed to $\mathcal{O}(|\mathsf{children}(v)|)$. However, this comes at the cost of a more expensive apply operation where we will need to update the list of sorted children. This trade-off is acceptable because the number of evaluations is typically several orders of magnitude larger than the number of apply operations.

A.4 Local search exit conditions

Recall that our local search algorithm terminates when it can no longer make any progress. In some cases, depending on the values of the objectives, it might be possible to determine if any future moves can result in an improvement, and if not end the search early. Below, we briefly describe the notions of *global* and *local* optimality that are used for this purpose.

Global Optimality

Recall that we had recursively computed the lower bounds for each node in $\mathcal{G}$. If the current value of the objective root note has reached its estimated lower bound, we say that the current assignment is *globally optimal* and we can terminate our local search algorithm.

Local Optimailty

Observe that in Algorithm 3, we maintain a list of *explored* bins, which are the set of bins for which no improving move was found. We leverage this information to compute a new *constrained lower bound* for each expression. To do this, assume that all the bins in *explored* are frozen (can neither move objects in or out) and compute the new value of each expression. For example, consider a Lookup node v such that all its affected bins S_v are explored. Since future moves cannot improve the value of this node, we can establish that its *constrained lower bound* is its current value. Once the bound of all leaf nodes are updated, we can recursively compute the updated constrained bounds of all the expressions. If we establish that the value of the objective root node is the same as its constrained lower bound, then we say that the current assignment is *locally optimal*, and if this happens, then we can terminate the inner loop of our local search algorithm (lines 6-25 of Algorithm 3).

A.5 Numerical stability of incremental apply

Please refer to the discussion in the main text around incremental application of small changes in local search. Observe that another challenge that we need to tackle is numerical stability of *apply* operations. This is more important for nodes of the graph $\mathcal{G}$ such as *Sum* that can accumulate numerical errors by performing arithmetic operations on values of children nodes. For example, let ε be the numerical tolerance for satisfying a constraint. That is, two values are considered equal if they are within ε of each other. Now suppose applying every change incurred an error of say $\varepsilon' = 10^{-3}\varepsilon$, we would accumulate an error of $K\varepsilon'$ after applying K moves. Say if $K = 10^4$, would incur an error of 10ε which is enough to invalidate a valid constraint. Although some amount of precision loss is unavoidable, apply operation for expressions in Rebalancer are designed to minimize precision loss as much as possible. In some cases, as shown below, there is a trade off between running time and precision loss and we need to use a slightly advanced data structure to achieve both.

For example, consider the Sum, where we have two alternatives for recomputing its value for a given change δ. Note that any floating point arithmetic often results in some loss of precision.

- *Case 1 : Slow with small precision loss and numerical stability.* Compute the value dynamically after every change by summing the values of children. This takes $O(|\mathsf{children}|)$ time. Even though there is some precision loss in this process, the resulting value is the same as the values are added in the same order.

- *Case 2: Fast with high precision loss and numerical instability.* Let z_0 be the current value of the total sum, and z_p, z_n respectively be the sum of prior and new values of the *changed children*. Then the updated value of this node after applying the change is $z_0 + z_n - z_p$. Note that this takes $O(|\delta|)$ time but we would likely accumulate some additional precision loss by subtracting two approximate numbers z_n and z_p. Moreover, this problem is encountered per update, and it adds up across many incremental updates. Finally, this also results in numerical instability because the result depends on the order of applying updates.

We can address this problem by building and maintaining *segment tree* structure over the children values that supports computing sum of values in a given range as well as value updates in logarithmic time. Therefore applying the change δ takes $O(|\delta| \log |\mathsf{children}|)$ time but incurs smaller precision loss and better numerical stability.

Appendix B Additional Details on Rebalancer's MIP Based Solver

In the previous section, we described a local search based algorithm for solving assignment problems. However, for problems that are not too big or where the solution quality is extremely important, we can use commercial Mixed Integer Programming (MIP) solvers such as Gurobi [15] and Xpress [45]. As we will soon describe, here again we use the flexibility of the expression graph $\mathcal{G}$ to translate all or part of the problem to a MIP model, which in turn enables us to combine the strengths of MIP solvers with our local search algorithm for applications that need them.

B.1 On-the-fly translation to a MIP model

Recall that the standard MIP model for an assignment problem consists of *binary* assignment variables v_{ij} for every object bin pair (o_i, b_j). In Rebalancer, we reuse the notion of *equivalent objects* briefly described in Section A.1 to succinctly combine assignment variables of objects that belong to the same equivalence class. To see how, first, let $\mathbb{O}_d$ be the collection of equivalent sets of objects; each element $\mathbb{O}_d^i \in \mathbb{O}_d$ is a set of equivalent objects. Next, we introduce the notion of *dynamic bins*, denoted $\mathbb{B}_d$. A bin is dynamic if objects can move in or out of it. Note that depending on the constraints, some bins may not be able to do either (i.e., they are frozen). Given this, the assignment variables in our MIP model are defined as follows.

1. For each eq. object set $\mathbb{O}_d^i$ and dynamic bin $b_j \in \mathbb{B}_d$, create an *integer* variable v_{ij} that represents the number of objects of type i that are assigned to b_j. Set the lower bound of v_{ij} as 0 and upper bound as $|\mathbb{O}_d^i|$.
2. For each $\mathbb{O}_d^i$, add an *object integrality constraint*: $\sum_j v_{ij} = |\mathbb{O}_d^i|$ which ensures that all the equivalent objects in a set are assigned.

Note how the above reduces the number of variables from $\mathcal{O}(|\mathbb{O}| \cdot |\mathbb{B}|)$ to $\mathcal{O}(|\mathbb{O}_d| \cdot |\mathbb{B}_d|)$. In our experience, this optimization can sometimes be the difference between being able to solve the problem using a MIP solver and otherwise.

Given the assignment variables above, similar to *evaluate*, and *apply* operations, every Rebalancer expression implements a mipTranslate operation, which knows how to correctly translate the expression based on its type into a linear combination of assignment variables. Below we show a couple of examples (see [44] for details on translating many other expressions including some non-linear types).

- For a leaf node v of type Lookup, we can implement the mipTranslate similar to the native representation of util defined in Equation 1.
- For a node v of type Max, we can use standard techniques of translating a max function to MIP model. For simplicity, consider the easy case when v is minimized by some objective. In that case, we add a new variable z to the model, for all $w \in \mathsf{children}(v)$, add constraints $z \geq \mathsf{mipTranslate}(w)$, and return the expression z.

With mipTranslate operation of each node in place, we can build the MIP model M for the entire problem by recursively calling mipTranslate on the objective root obj_r and all the constraint roots, ctr_r^i. Once we have the MIP translation, the objective in the MIP model is to minimize $\mathsf{mipTranslate}(\mathsf{obj}_0)$ and the constraints are $\mathsf{mipTranslate}(\mathsf{ctr}_r^i) \leq 0$ for all ctr_r^i and the ones that are added during the mipTranslate calls on a node (like in the case of Max node described above).

B.2 Solving the model

We can use the aforementioned translation algorithm parameterized by dynamic bin set $\mathbb{B}_d$ to generate the MIP model and solve using commercial solvers. If the problem is small enough, we can solve the full problem with $\mathbb{B}_d = \mathbb{B}$. Otherwise, we can use the hottest bin ranking from Algorithm 4 to select an appropriately sized subset $\mathbb{B}_d \subset \mathbb{B}$ of dynamic bins and solve only part of the problem. This technique can in turn be useful if we want to combine our local search algorithm with MIP solvers.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

μSlope: High Compression and Fast Search on Semi-Structured Logs

Rui Wang[†] Devin Gibson[†⋆] Kirk Rodrigues[†] Yu Luo[†‡⋆]
Yun Zhang[‡] Kaibo Wang[‡] Yupeng Fu[‡] Ting Chen[‡] Ding Yuan[†⋆]

YScope[†] *Uber* [‡] *University of Toronto*[⋆]

Abstract

Internet-scale services can produce a large amount of logs. Such logs are increasingly appearing in semi-structured formats such as JSON. At Uber, the amount of semi-structured log data can exceed 10PB/day. It is prohibitively expensive to store and analyze them. As a result, logs are only kept searchable for a few days.

This paper proposes μSlope, a system that losslessly compresses semi-structured log data, and allows search without full decompression. It concisely represents the schema structures, and only keeps this representation stored once per dataset instead of interspersing it with each record. It further "structurizes" the semi-structured data by grouping the records with the same schema structure into the same table, so that each table is also well structured. Our evaluation shows that μSlope achieves 21.9:1 to 186.8:1 compression ratio, which is at least a few times higher than any existing semi-structured data management systems (SSDMS); The compression ratio is 2.34x as high as Zstandard and the search speed is 5.77x of the other SSDMSes.

1 Introduction

In the past two decades we have witnessed an explosive growth of log data from Internet-scale systems. Conventionally, logs were only in the form of unstructured, free text (e.g., output from `printf()`). However, they increasingly appear in semi-structured format, such as JSON [12] or Protocol Buffers® [9]. [1] These data models have a tree-structure, where each node (except for the root) is a key-value pair. The value may include non-primitive data types such as nested values.

Different records may have different schema structures. This is why they are referred to as *semi*-structured. For example, JSON and YAML [15] formats are schema-less, meaning that records may have arbitrarily different schemas. This flexibility allows programmers to easily log common data types in high-level programming languages, such as C struct, class, hash table, array, etc. Although Protocol Buffer and other formats require users to declare a schema to be used on all records, they allow a field to be optional. As a result, different records may still have different structures.

The dynamic schema structure imposes challenges for data management. For example, naively extending conventional relational databases would require creating one column for each *possible* key. This results in a sparse table (i.e., each row may have many NULL values), and it is challenging to handle polymorphic typing (the value of the same key might have different types).

Some existing semi-structured data management systems (SSDMS) use custom-designed data structures to store the schema structure of each record [2, 8, 20, 26, 29, 30]. These systems were initially designed for user-generated data which is relatively small (and they primarily focused on fast search speed). For example, most of the published works on SSDMSes use Twitter® datasets (i.e., Tweets in the Twitter data feed are in JSON) as their primary evaluation target [20, 26, 29, 30]. Twitter reports that in 2023 there are a total of 500 million Tweets per day [13], which results in 140GB of Tweet data per day (assuming 280 characters for each Tweet).

In comparison, logs with machine-generated data can be orders of magnitude larger. The size of JSON logs at Uber from all services exceed 10 Petabytes on a busy day, or 60PB/week, which is more than 70,000x the size of tweets. Managing data at this scale with existing SSDMSes is prohibitively expensive. These systems need to store one schema structure for each record, resulting in a large amount of duplication when a large number of records share the same schema. As a result, even if some systems can efficiently compress the data content [20, 30], the overall storage size is still large as it is dominated by the schema structures [20].

At Uber we have repeatedly suffered from scalability issues as Uber grew. Initially we used Elasticsearch® [21] to manage our JSON logs. It builds indexes for every word in a JSON document to support full text search, hence the index size is at the same order of magnitude as the original data and needs to be stored on SSD for fast search. The resource cost, together

[1] While we do not have global data on the prevalence of the semi-structured logs versus unstructured ones, in Uber, the size of semi-structured logs is about 10x of the unstructured. Part of the reason is that, even if some third-party or legacy applications emit unstructured logs, our log aggregation tools would wrap them in JSON. This is also a common practice outside of Uber. For example, Amazon CloudWatch® [1] also wraps unstructured log outputs in JSON. Grafana Loki® [5] tags unstructured logs essentially turning them into semi-structured logs for the purposes of search, and so on.

with other issues (§2), forced us to migrate away from it around 2019. Since then we have been using ClickHouse® [4], a columnar RDBMS with features for handling JSON data. Its overall compression ratio on our setup is less than 4:1, and it also requires SSD for fast search. Therefore the resource cost is still prohibitively expensive.

CLP [27] and LogGrep [31] are able to compress unstructured logs and allow users to search compressed data without full decompression. While effective on unstructured logs [2] they are limited on semi-structured logs. Fundamentally, their storage structure is not designed for the semi-structured format. For example, CLP parses an unstructured log into three components: timestamp, log type, and variables. The entire log dataset can be stored in a table consisting of these three columns: each log message is stored in one row, and all of its variable values are stored as a list in the variables column. Even if we extend CLP's parser to recognize the key/value structure (so the schema structure can be treated as the log type), the values of all the fields are intertwined in the same variables column of a single table. This hurts compression, but more importantly, querying any field requires tedious decompression and scan of the entire variables column. Indeed, real-world users who use CLP to manage JSON logs encounter poor search performance exactly for these reasons [10]. In addition, these tools only support a wildcard substring query model (like `grep`), but do not provide boolean algebra support to filter on multiple fields.

We propose μSlope[3], an SSDMS for semi-structured log data. μSlope focuses on storage efficiency by losslessly compressing the logs. And we show that fast search can be achieved with a careful system design without having to use an index (which would increase storage overhead). μSlope does not require any user annotation as it automatically handles the dynamic schema structures.

μSlope incorporates three novel designs. The first is its concise representation of the schema structures. Unlike existing SSDMSes that store separate schema information for each record, μSlope proposes (1) a *merged parse tree* to store the schema structures generalized for patterns specific to logs, and (2) a *schema map* to concisely represent each schema with a set of leaf nodes in the tree. Each unique schema structure is stored only once in the schema map, instead of per record, and it is decoupled from the storage of the value contents.

μSlope then partitions the records into *different* tables to store their values, where the records in the same table have the same schema. Therefore each table is now *perfectly* structured: all records have the same number of keys and each value is of the same type. This allows us to apply well-studied optimizations designed for relational tables. For example, we can store and compress each table in a columnar manner that maximizes the compression ratio and search speed [16].

Finally, μSlope uses a query processing algorithm that leverages the schema information and encoded tables. μSlope first builds an abstract syntax tree (AST) from the query, and systematically refines this AST by looking up the merged parse tree and schema map. This leads to early termination of queries that do not match any schema structures, and allows μSlope to decompress and scan data only when necessary.

μSlope's design was guided by a characterization study of Uber's semi-structured logs and queries (§3). We found that while records do have dynamic schema structures, there is enormous repetition as most records share a small number of common schemas. In addition, nearly one third of the queries that users performed can be terminated without table scanning because they do not match any of the schema structures.

We evaluate μSlope on a total of 21 semi-structured log datasets, and compare it with a number of widely used SSDMSes. Our result shows that μSlope achieves a compression ratio of 68.1:1 on average. This is at least 2.6x better any existing SSDMSes'. The compression ratio is even 2.34x of Zstandard's and 1.70x of LZMA's at the default compression level, even though they do not support search on compressed data. μSlope's search speed is 2.47x of ClickHouse's, 8.09x of PostgreSQL's, and 6.74x of MongoDB®'s.

This paper makes two contributions. First, it proposes μSlope, a resource efficient archival SSDMS that compresses semi-structured log data and enables fast search without full decompression. Second, we present an in-depth analysis of the characteristics of the semi-structured log data at Uber.

μSlope also has a few limitations. It is designed for semi-structured text logs, which are highly repetitive, instead of being a general-purpose SSDMS. If every record uses a different schema structure it won't work well on μSlope. Furthermore, μSlope uses an index-less design to optimize for storage efficiency; its search speed, however, may not be as fast as index-based search tools like Elasticsearch.

2 Background and Related Work

In this section we present the background information and related work. We explain (1) the semi-structured data model, (2) how prior SSDMSes manage the semi-structured data, and (3) commonly used compression algorithms.

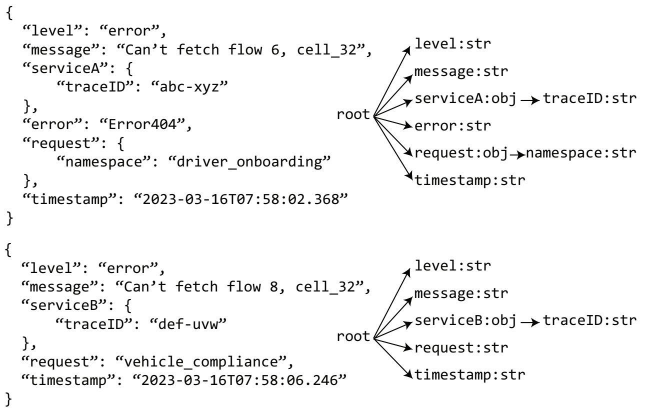
2.1 Semi-structured Data Model

Semi-structured data have a tree structure, and its data model can be defined as follows:

$$\begin{aligned}
T_{root} &= T_{object} \\
T_{object} &= \{key_1 : T_{value1}, \ldots, key_n : T_{value_n}\} \\
T_{value} &= T_{object} | T_{array} | T_{primitive} \\
T_{array} &= [T_{value}, \ldots, T_{value}] \\
T_{primitive} &= string | number | boolean | null \\
key &= string
\end{aligned}$$

[2] CLP is deployed across Uber's various data and ML platforms to manage the unstructured logs.

[3] μSlope: Semi-structured LOg Processing Engine like a micro(μ)-scope.



```
{
  "level": "error",
  "message": "Can't fetch flow 6, cell_32",
  "serviceA": {
      "traceID": "abc-xyz"
  },
  "error": "Error404",
  "request": {
      "namespace": "driver_onboarding"
  },
  "timestamp": "2023-03-16T07:58:02.368"
}
```

```
{
  "level": "error",
  "message": "Can't fetch flow 8, cell_32",
  "serviceB": {
      "traceID": "def-uvw"
  },
  "request": "vehicle_compliance",
  "timestamp": "2023-03-16T07:58:06.246"
}
```

Figure 1: Two example log records and their schema trees.

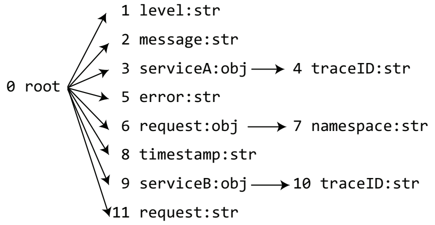


Figure 2: The merged schema tree of the two records.

Each semi-structured *log record*, or record for short, corresponds to a T_{root}. Figure 1 shows two sample records. A log dataset contains a number of such records. The *schema* of each record can be represented as a tree [30]. Figure 1 shows the two schema trees. Each node records a *field*, which consists of the name of the key and the type of its value.

Only the leaf nodes can have primitive value types in the schema tree. Any non-leaf nodes would have non-primitive value types, that is, either object or array.

If two records have the same schema tree, we say they have the same schema. The two nodes in their respective trees are considered to be the same if and only if both the key and the value type are the same. In a schema-less data format like JSON, a key could have "polymorphic" values, i.e., different value types in different records. For example, the "request" field in the two schemas in Figure 1 have different value types.

The schema trees of multiple records can be merged into a single tree [30]. We call it the *merged schema tree*, or MST. Specifically, given node N_1 and N_2 from two schema trees, we can merge them if and only if: (1) N_1 and N_2 have the same key name; (2) the value type are the same; and (3) all the predecessor nodes of N_1 and N_2 in their respective schema trees can be merged. Figure 2 shows the MST of the two schema trees in Figure 1.

2.2 Existing SSDMSes

Different semi-structured records may have different schemas, therefore we cannot naively store them in the RDMS table.

```
\x16\x00\x00\x00          // size (32-bit): 22 bytes
\x02                      // 0x02 = value type String
hello\x00                 // key name
\x06\x00\x00\x00world\x00  // size of value (6 bytes), value
\x00                      // 0x00 = 'end of object'
```

Figure 3: The BSON representation of {"hello": "world"}.

As a result, existing SSDMSes either use natively designed storage format or extend RDBMS in sophisticated ways.

Native Support for Semi-structured Data. These SSDMSes use custom data structures to represent the schema structure *of each record*. MongoDB uses a concise binary format called BSON (Binary JSON) [2]. Figure 3 shows the BSON representation of a simple JSON record {"hello": "world"} [3]. It stores the schema structure, including the type and size of each key/value pair, interspersed with the record content. The BSON records can only be stored in a row-oriented manner, which limits both the compression and search speed because it has to scan the entire record even when the user only searches for a specific key. Fast search can only be achieved via creating external indexes.

PostgreSQL®'s jsonb [8] and Oracle®'s OSON [26] are two other examples of custom schema structures. The former is similar to BSON, while the latter stores more metadata information, such as the number and offsets of nested keys.

Steed [30] proposes both row and column oriented storage methods. It operates on the merged schema tree (MST). Instead of storing the key name with each record, Steed only needs to store the node ID of the key in the MST. However, it still keeps one schema structure for each record in its row-oriented method. Keeping track of the schema structure is even more complex in its columnar method, as it splits the datasets into N independent columns where N is the number of keys, each column stores all the values of that key. Because the key/value pairs of a single record are now split, assembling the original record's schema structure requires even more complex data structures (2 additional columns for each key) and algorithms (finite state machine).

Scuba [17] is an in-memory SSDMS that keeps records in a compressed row-oriented format. Records are stored one after another in a table. Therefore it needs to scan all records in a table (and all the fields) during search. It stores the string values in a dictionary, and the dictionary is used as an index during search. However, unlike CLP, the entire string is stored as a single entry instead of being parsed into timestamp, log type, and variable values. It also has some restrictions on the data model, most notably it prohibits nested keys. Note that Scuba is designed for general-purpose data storage instead of narrowly focusing on logs.

Extending RDBMS. Sinew [29] uses a hybrid design that materializes a subset of the keys as separate columns, and stores the remainder using a binary format similar to BSON in a single RDBMS column. JSON Tiles [20] extends the idea of Sinew, reordering the records to group those with similar schemas into disjunct tiles. However, it still needs to keep a binary schema structure for each record. Although JSON Tiles applies compression to those separate columns, the compression ratio is around 1 because the large size of the per-record schema structure.

Some RDBMSes map semi-structured data into relational tables. For instance, Argo [19] proposed two methods. The first stores fields in a single table, while the second splits it to three where each is used for a primitive value type. However, the first method will result in sparse tables (many fields with NULL values). Both methods take up too much storage space due to repetitive key/values. Liu *et al.* proposed to store the entire JSON record in a single RDBMS column [25].

Elasticsearch and ClickHouse: Experiences at Uber. Elasticsearch [21] is a JSON document store that supports search. It assumes the type of the field will never change by default and cannot handle fields with dynamic types easily. (By default, Elasticsearch will drop records if a field has a different type than in a previous record.) It also struggles to handle the case where every record has new unique keys (e.g., using a UUID as a key) [18]. At Uber we used to use Elasticsearch, but migrated away (to ClickHouse) due to these issues as well as its excessive resource usage.

ClickHouse is a columnar RDBMS with features for handling JSON data. At Uber we built a layer to transform records before ingestion into ClickHouse. After experimenting with different storage formats, we settled on storing a handful of common fields in dedicated columns, and storing all other unique key-value pairs that are less than or equal to 3 levels deep in a pair of arrays per type. For instance, string-type fields were stored in two arrays, "`String.names`" and "`String.values`". To query a field, we first find the key in the `String.names` array, use the index of that key to index the `String.values` array, and finally compare the value against the query. We also keep a _`source` field that contains raw JSON and build inverted index on top of it. Finally, to improve query performance for frequently accessed fields, our abstraction layer temporarily extracts these fields into temporary dedicated columns (materialized columns). This setup results in a compression ratio lower than 4:1.

2.3 Compressors

General-purpose compressors like Gzip [23] and Zstandard [22] use a sliding window approach proposed by Lempel and Ziv [33]. They locate duplicates within a fixed-size sliding window, so they cannot detect duplications if the distance between the two duplicated patterns is larger than the size of the sliding window. Therefore storing duplicated patterns close to each other would maximize the compression ratio.

Searching, unfortunately, requires decompressing the entire data. These compressors typically encode duplicates in length-distance pairs [28, 33]. Starting from the current position, if the next L (length) characters are the same as the ones starting at D (distance) behind, the next L characters can be encoded with (D,L). This (D,L) pair is directly embedded in the compressed data, hence search requires decompression from the beginning.

Log compression and search. Existing log compression techniques focus only on unstructured (i.e., free text) logs. CLP [27] uses a custom-designed log parser to split each message into three components: timestamp, static text (i.e., log type), and variable values, structurizes logs into a table of three columns. CLP stores the static text and repetitive variable values into respective dictionaries, and the dictionaries also serve as coarse-grained index during search to minimize decompression and scan. It then compresses the three-column table in columnar order.

LogGrep [31] also compresses unstructured logs and allows search. It uses a training phase to identify the common patterns of messages. LogGrep identifies patterns in much finer granularity (e.g., a variable can be further divided into subcomponents if a different subcomponent is repetitive). Therefore a message is split into a larger number of subcomponents without clear mapping to program semantics, and it uses tables to store the complex mapping to assemble these subcomponents into the original log message.

3 Characterizing Semi-structured Log Data

We first provide a characterization study on semi-structured log data before describing the design of μSlope. While prior works have characterized semi-structured *user-generated* data [30], we are the first to provide an understanding of *machine-generated* semi-structured log data.

We collected 16 frequently used log datasets (LogA-LogP) from Uber and 5 log datasets from open-source software (Apache Spark™, MongoDB, CockroachDB®, Elasticsearch, and PostgreSQL). All of these datasets are in JSON format. We limit each dataset to 1,000,000 log records. We also provide a characterization of real-world queries on semi-structured log data by analyzing a total of 23,091 queries spanning twenty days from Uber; 7665 of them are unique.

Schema Variation. We first study the schema variation. JSON's schema-less nature means that the variation of schemas between records can range from zero (i.e. all records have the same schema) to 100% (i.e. all schemas are different). Recall from §2.1 that the schemas of two log records are considered the same if and only if their schema trees are identical. The degree of variation is a critical consideration to the design of μSlope and prior systems. If there is no variation, then one can easily store logs in an RDBMS by materializing one column for every leaf-node node.

Figure 4 shows the unique schemas for each dataset. All except two datasets have more than 1 unique schemas, with LogE having the largest variation (6,176 unique schemas). The median number of unique schemas of all datasets is 40.

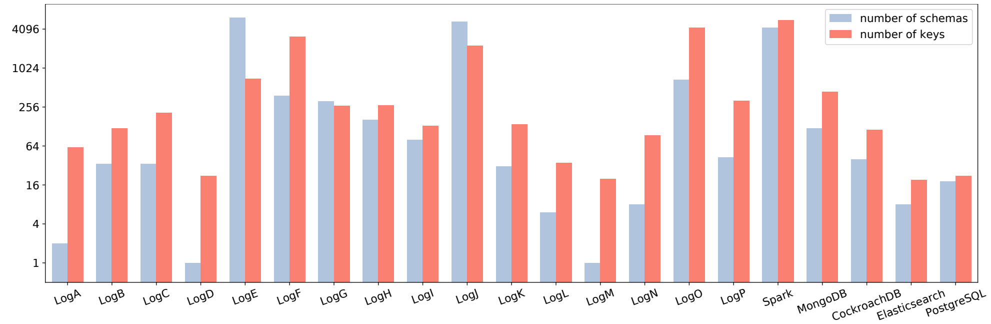


Figure 4: Number of unique schemas and keys for each dataset.

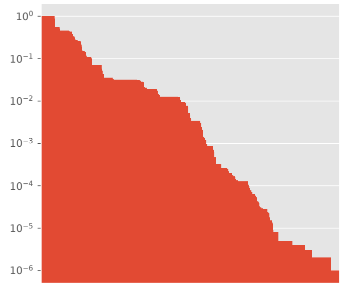


Figure 5: KF distribution of LogE

Despite the relatively large degree of variation, there also exists a large degree of *repetition*. On average, there are 25,000 records in each dataset with the *same* schema; if we increase the sample size, this repetition will be larger. Even for the most noisy dataset, LogE, there are still an average of 162 records per schema.

To understand the schema variation, we further measure the variation of individual keys. Figure 4 also shows the number of unique keys for each dataset. Two keys are considered the same if and only if their full name and value types are the same. A key's full includes all the nested keys (i.e., predecessor nodes in the schema tree). For example, "serviceA.traceID" is the full name whereas "traceID" is not. The median number of unique keys is 138, and it varies greatly. LogM has only 20 unique keys (and it has only 1 schema), whereas Spark logs have 5,627 unique keys.

This result suggests that the variation in schemas is likely due to the variation of individual keys instead of their combinatorial effect. In theory, a large variation in schemas could be the result of a small number of keys: n unique keys could result in 2^n different combinations, hence 2^n schemas. However, this is not the case with logs. In fact, in 18 out of 21 datasets the number of unique keys is larger than the number of schemas, showing the opposite of a combinatorial effect.

To get better understand how keys are distributed within a dataset, we measure the key frequency (KF). It is defined as

$$KF(x) = \frac{number\ of\ records\ that\ contain\ key\ x}{total\ number\ of\ records}$$

Figure 5 takes LogE as an example to show the distribution of KF. Each bar represents a key. LogE has 6176 schemas and 704 keys. There are a small number (21) of keys that have KF = 1.0, indicating that they appear in every record. These are the keys uniformly added by the logging library, such as "timestamp" and "level".

On the other hand, there is a long-tail in the KF distribution of LogE. 83.0% of the keys has a KF <0.1. Most of them are different data structures in the program that documents the program state relevant to a specific event. There are also cases where the variation in schemas is caused by a large variation in the name of a key, like Spark using the pathname as the key name, or some datasets using the UUID as the key name.

Type Composition. Next, we break down the value types. Recall from §2.1, the value type can be an object, array, or one of the primitive types. We further refine the types as follows. First, we break down the number type into integer and float. For strings, we separate single-word values from multi-word ones (using white space as word delimiter), because the former is likely a variable (e.g., an identifier) whereas the latter is free-text log. We call the former *variable* and the latter *log-text*.

Figure 6 shows the breakdown of the value types in each dataset. On average, 70.8% of the values in each dataset are variables (i.e. single-word strings). 10.8% are objects, i.e. non-leaf nodes in the schema tree leading to nested keys. In comparison, the percentage of boolean, float, null are low, averaging 1.74%, 1.21%, 0.22%.

The percentage of **array** fields are also low at 0.79%. In addition, only 28 of the 7,665 (0.4%) of the unique queries explicitly search on an array field. Furthermore, these explicit array searches only match 0.05% of the data on average.

Each log record contains an average of 1.6 log-text keys. In addition, 41% of the unique queries contain filters on log-text, so efficient search on log-text is important. In addition, a record contains 4.0 integer fields on average.

Repetitiveness of Variables. Next, we zoom into variable fields (i.e., single-word strings), because they dominate the composition of logs. The question we care about is: Are these values repetitive? We use the repetition ratio to measure the repetition of variable fields. It is defined as

$$repetition\ ratio = \frac{number\ of\ all\ variable\ values}{number\ of unique\ variable\ values}$$

A high repetition ratio means that unique variable values are much fewer and these fields are repetitive. Figure 6 shows the repetition ratio. The median repetition ratio is 37.8 and the average is 58.2 across all datasets. It can be as high as 433.4 (LogD) and even the minimum is still 9.29 (LogC). This means dictionary deduplication can be effective.

The variable fields are also frequently queried. The average query contains 3.450 filters with 2.663 of those being variable filters. For example, `level:"warn" OR level:"error"` is a query that has two string filters on the `level` key. The largest query has 93 string filters. Furthermore, filters that implicitly

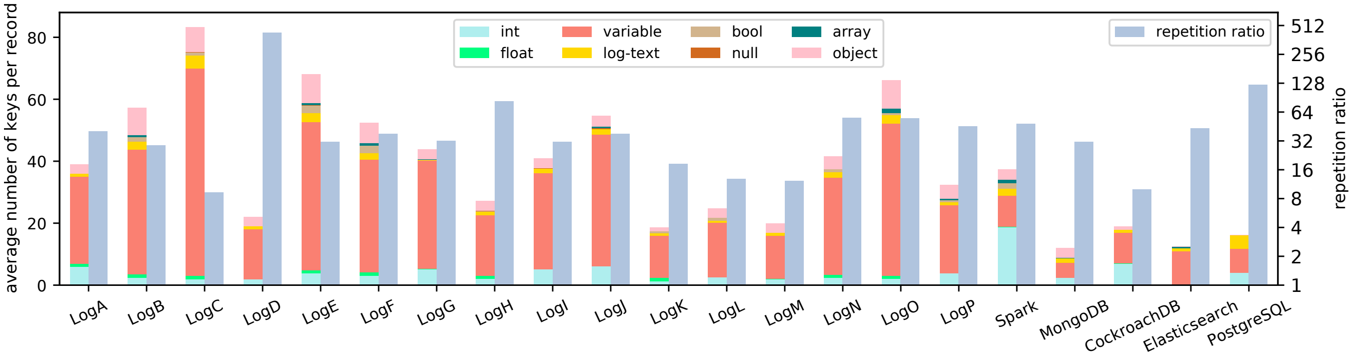


Figure 6: Average number of keys per record, broken down by different value types and repetition ratio of variables.

match all keys – "wildcard keys" – are common in practice. For example, `*:"aUUID"` matches any keys as long as its value is “aUUID”. Of the 7,665 unique queries 2,271 of them contain a wildcard key. These filters are easier to express for users of search, but pose a significant performance hazard. In the worst case such filters can impose scans over the entire record, eliminating much of the benefit of querying structured data. However, using a dictionary to store the variables would significantly speed up such queries as we only need to search the dictionary for matching values.

Importance of Schema Search. Nearly one third (29%) of the unique queries do not match any of the schema structures. That is, they can be returned without scanning the values. Examples include searching for a nonexistent key, the key exists but the value type does not match, or the conjunction of keys/value types do not exist. For example, engineers performs such search to regularly verify the nonexistence of certain error events. However, for existing SSDMSes (such as MongoDB and PostgreSQL) this opportunity is wasted, because the schema structure is interspersed with the values.

Summary and Takeaways. Schemas are dynamic, and those unstructured keys are frequently queried. Therefore simple extension of RDBMS to only materialize those structured keys as columns is insufficient, and we need to precisely track the schema of semi-structured data. On the other hand, a large number of records have the same schema, presenting opportunities to group them by the same schema so that each group is well-structured.

70.8% of the keys are single-word strings. They are highly repetitive, and commonly queried on. This indicates that storing them in a variable dictionary would effectively deduplicate them, and at the same time, speedup the search.

Finally, efficiently storing the schema structure and decouple it from the record value data would significantly speedup the 29% of the queries that can be completed only by querying the schema structure.

4 Overview of μSlope

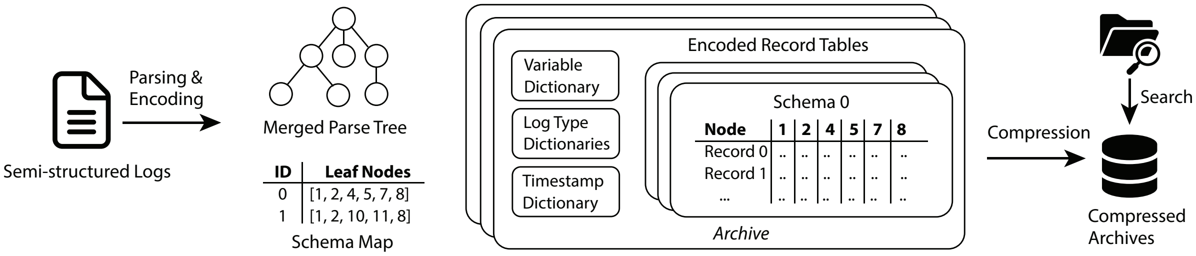
μSlope is a resource-efficient SSDMS that we designed from the ground-up. μSlope has three major designs that are novel compared to other SSDMSes: (1) Decoupling the storage of schema metadata from the storage of each record; (2) Grouping records by schema to store them into well-structured tables, and apply efficient encoding; and (3) Optimized schema metadata lookup to speedup search.

Figure 7 shows μSlope's architecture. When data is ingested, μSlope parses each record and extracts its schema. It uses two core data structures to track the schema structure: the merged parse tree (MPT) and the schema map. The MPT and the schema map are collectively referred as *schema metadata*. It is critical to keep the schema metadata as small as possible, yet still captures the highly repetitive structure of the log.

The MPT is a more general form of the merged schema tree (MST) as described in §2.1. It has four differences when compared to MST. First, the MPT can contain special unnamed nodes that mark the truncation of some key value pairs. Multiple rare keys can be mapped to the same unnamed node when the key names contain random data, such as UUID or file path. Including such non-repetitive key names would bloat the metadata. In contrast, sometimes the value of a key could be highly repetitive. For example, all the records from the same application would have the same value under the application-name key. Therefore, the MPT also allows a node to include the value (only if the value is highly repetitive).

The third difference is that MPT can encode the structure of strings with key-value pairs. This allows μSlope to capture more structural information from strings, improving compression and search. For instance, given a record `{.. "message": ".. latency=35, status=OK, type=READ, ..."}`, μSlope would create three nodes for “latency”, “status”, and “type” respectively as the children of the “message” key. This requires that schemas contain an *ordered* region where we maintain some leaf nodes in a specific order, because the order of the keys in a string needs to be preserved.

Another difference is that MPT stores more fine-grained string types. A string value could either be a timestamp, a single-word string, or a log-text. Storing fine-grained types



ID	Leaf Nodes
0	[1, 2, 4, 5, 7, 8]
1	[1, 2, 10, 11, 8]

Node	1	2	4	5	7	8
Record 0	..	..	..	..	..	..
Record 1	..	..	..	..	..	..
...	..	..	..	..	..	..

Figure 7: Architecture of μSlope.

enables more filtering opportunities on MPT, resulting in better search performance.

The *schema map* stores each unique schema in the log dataset in a hashmap. We make the observation that a schema can be unambiguously identified with the list of *leaf nodes* in the MPT. For example, the schema map in Figure 7 shows the two schemas of the two records in Figure 1, where the node IDs corresponds to the MST shown in Figure 2. (In this example, the MPT and the MST are the same.)

The MPT and schema map deduplicate the highly repetitive schema structures. Unlike prior SSDMSes §2.2 that store a schema structure for every record, each unique schema is stored only once. In practice, the schema metadata size is typically less than 0.0001% of the total compressed data size. This design also enables fast search: the succinct metadata can be efficiently searched, providing powerful filtering capability.

μSlope uses one table for each schema to store the values. Therefore each table only stores the values of the records that have the same schema. As a result, there can be thousands of tables, one for each schema. The advantage is that each table is now perfectly structured, as all records have the exact same keys and value types. μSlope essentially structurizes those semi-structured data.

The tables are called *Encoded Record Tables* (ERT), because instead of storing their raw value, μSlope performs type-specific encoding. For example, single-word string will be stored in a *variable dictionary*, so only an ID is stored in the ERT. The dictionaries serve two purposes at the same time: it effectively deduplicates the highly repetitive patterns, and it serves as coarse-grained index for search so μSlope only needs to scan the ERT that contains the matching record.

Each ERT is stored and compressed in a columnar order. This significantly improves both the compression ratio and search performance [16], because a column groups the semantically similar values together so it maximizes general-purpose compressors' ability to find repetitions, and during search we only need to decompress and scan the columns whose keys were searched for.

Log Type Dictionary

ID	Log Type
L0	Can't fetch flow \INT, cell \DICTVAR

Timestamp Dictionary

ID	Format
T0	yyyy-MM-dd'T'HH:mm:ss'.'SSS

Variable Dictionary

ID	Variable Value
V0	error
V1	cell_32
V2	abc-xyz
V3	Error404
V4	driver_onboarding
V5	def-uvw
V6	vehicle_compliance

Schema 0 Encoded Record Table

Node	1	2		4	5	7	8	
Values	V0	L0	6,V1	V2	V3	V4	T0	1..8

Schema 1 Encoded Record Table

Node	1	2		10	11	8	
Values	V0	L0	8,V1	V5	V6	T0	1..6

Figure 8: How μSlope encodes log records.

μSlope leverages the efficiency of metadata and dictionary lookup to optimize the search performance. It uses Kibana Query Language (KQL) as its query language, which is both concise and powerful. μSlope transforms a query into an abstract syntax tree (AST) to perform a number of optimizations, including determining if the query matches any schema and if the filter pattern matches any dictionary values. If not, μSlope will terminate query processing early. Otherwise, only relevant ERTs are finally decompressed and searched through.

Both compression and search are embarrassingly parallel. During compression, when parsing a new record μSlope extracts the key-value pairs of each log record. Corresponding key nodes are added to the MPT, with leaf node IDs collectively representing a schema. Values are encoded by various methods and are stored in the ERTs. Upon identifying a new schema, μSlope dynamically creates a new ERT to store the encoded values. All the ERTs, dictionaries and schema metadata will be buffered in the memory. Once the buffer reaches a certain size, they are compressed using Zstandard before stored to disk, creating what we call an *archive*. μSlope then clears the buffer and dictionaries before compressing newly arrived records. Therefore different archives can be compressed, searched, and decompressed independently in parallel.

5 Compression

We use simdjson [11] parser to parse the JSON structures. Other log formats representable by the data model in §2.1 can also be supported by integrating a parser to extract the key-value pairs from the records.

μSlope uses different encoding techniques on different value types. Next we describe them in turn.

Strings. μSlope treats string values differently depending on whether it is (1) a timestamp, and if not, (2) a variable, i.e. a single-word string, or (3) log-text. μSlope uses CLP's parser to parse a string value. It uses heuristics to detect if the string is a timestamp. If so, μSlope encodes it into a Unix® epoch time and stores the format pattern in the timestamp dictionary. As a result, each timestamp field is encoded into *two* columns in the ERT: a timestamp dictionary ID and a Unix epoch time. The ID column consumes negligible space after compression because most of the time there is only one timestamp format.

For a log-text, the CLP parser extracts the log type and variables. The log type is stored in the log type dictionary. Different types of variable values are encoded differently. Integers and floating point numbers are directly encoded in binary format. Other variables are encoded as a variable dictionary ID after storing them in the variable dictionary. Therefore a log-text also has two columns: a log type ID, and a list of encoded variables stored in a single column. We extend CLP's log parser [7] to allow users to specify rules to extract key/value pairs from log-text fields and store them as JSON fields. Therefore, if the log is in unstructured format (instead of JSON), or dominated by unstructured text log (i.e. majority of a record is an unstructured text message field with a few additional fields containing metadata like hostname, verbosity, etc.), μSlope essentially falls back to CLP encoding.

μSlope directly stores a single-word string in the variable dictionary and stores the dictionary ID in the ERT. The dictionary effectively deduplicates the repetitions in variables as we shown in §3. We use the same variable dictionary used for the log-text for maximum deduplication.

Figure 8 shows the contents of the dictionaries and encoded record tables for the example log records in Figure 1. Their MPT is the same as the MST shown Figure 2, except that the MPT keeps a fine-grained string type on each string field. The schemas are shown in Figure 7. We use different prefix and colors for the different types of dictionary IDs: log type ('L'), timestamp ('T'), and variable ('V'). For example, consider the first log record, which is stored in the ERT of schema 0. Four of its keys ("level", "serviceA.traceID", "error", "request.namespace") have single-word strings; they have the MPT node IDs 1, 4, 5, 7 respectively (Figure 2), and their values are encoded as V0, V2, V3, and V4 which are IDs into the variable dictionary. The "message" field (node ID 2) is a log-text. Its first column stores L0, which is the ID into the log type dictionary, and the second column stores the two encoded variables. Note that '\INT' and '\DICTVAR' in the log type are special placeholder bytes for variables (of integer and dictionary variable types respectively). The "timestamp" value is stored in the last 2 column (node 8); the value "1..8" is the encoded timestamp in Unix epoch time.

Integers, floating point, and boolean values are directly encoded in binary form in the same way as in CLP [27].

Arrays are stored as log-text strings by default, i.e., using CLP to parse it into a log type and a list of variable values. This means that arrays are typically searched by decoding and parsing their string representation. This is acceptable as §3 shows that arrays occur rarely and are seldom searched. Note that the log type parsed from an array string is stored in a separate log type dictionary to avoid polluting the regular log type dictionary for log-text. We also support fully parsing arrays and recording their structure in the MPT under a non-default configuration. This approach offers performance benefits for array searches, but typically results in growth in schema size due to the diverse internal structures of arrays.

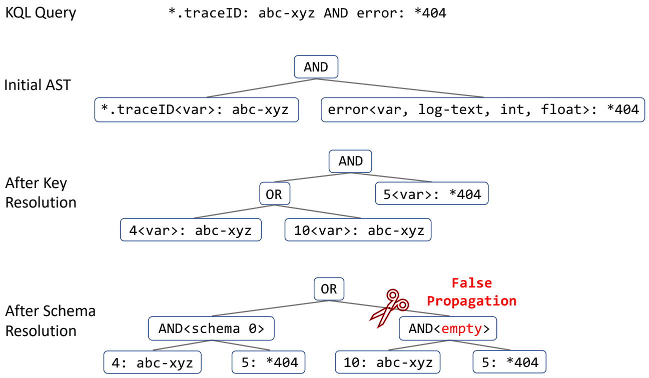


Figure 9: Example of query processing.

Preserving record order. Splitting the records into different tables means that we lose the order between records of different schemas. Using timestamp to order them is unreliable because records may have the same timestamp. To preserve the order, μSlope adds a column in each ERT to store the order of the record in the original log stream.

Random keys and invariant values. Recall that μSlope truncates the key name from the MPT if it is not repetitive, and includes values that don't change into the MPT. The heuristic we use is that if a key does not appear in more than 1% of the records of the archive, it will be truncated; whereas a value will be included in the MPT only if it never changes. We implement them by keeping counters for the keys and values as records are parsed and stored in the memory buffer; the decision to truncate a field or include a value in the MPT is made when we write the buffered data to disk (into the archive format). The structures of a truncated field and its successors (a truncated field may be a non-primitive type, in which case all the subfields will be recursively truncated) are encoded in a row-oriented format similar to BSON, and stored in a column that is mapped to a (special) MPT node located at the place of the truncated node.

6 Search

μSlope search accepts queries which combine filters on one or more keys. The key names in the query may contain wildcards and be of ambiguous underlying type. Search takes advantage of the MPT, *schema map*, and dictionaries to evaluate queries efficiently. This is a multi-step process, each step performs refinements and optimizations on a custom Abstract Syntax Tree (AST) μSlope built from the query. Next, we explain each step using an example shown in Figure 9.

Step 1: Constructing the initial search AST. Given a query μSlope transforms it into an AST where each leaf node specifies a filter on some key, and each non-leaf node is a logical AND or OR. Figure 9 shows an example KQL query. The query consists of two filters joined by AND. The key name in first filter contains a wildcard, meaning it can match any hierarchy of keys that end with "traceID". The second filter contains a wildcard in the value pattern.

μSlope parses this query into a search AST shown in Figure 9. Initially this AST contains two nodes, each maps to a filter. Each node also stores the possible value types inferred from the query. The value type of the first filter is unambiguous; it must be single-word string type (i.e., a variable) given the pattern is surrounded by '"'. The second filter, however, has ambiguous type; it can either be a variable, a log-text, an integer, or a floating point number.

Step 2: Key resolution further turns each key name into zero or more MPT leaf nodes by resolving ambiguities. Ambiguities come from two sources: (1) ambiguous key name, i.e. a key with wildcard can match more than one leaf of the MPT, and (2) ambiguous value type, when it is polymorphic. In both cases we replace the corresponding filter with an OR node where every child of the OR is the same filter with the key replaced by each of the matching MPT leaf nodes in turn. The "*.traceID<var>" in Figure 9 is an example of ambiguous key name. It is resolved into two MPT leaf nodes, node 4 and 10, which corresponds to "serviceA.traceID" and "serviceB.traceID" respectively. These two nodes are connected by OR in the refined AST after the key resolution.

The "error<var,int,float>:*404" node in the initial AST has potentially ambiguous value type. In our example, because the only possible type for the "error" key is a variable (i.e., single-word string), we replace it with a single node "5<var>:*404". However, if "error" has polymorphic type, say an integer, in the dataset, then we need to consider both possibilities and connect them by an OR.

When a key name matches no leaf nodes in the MPT, that AST node is eliminated by replacing it as false and propagating this false up the AST, a process known as *false propagation*. This can eliminate part or all of the query.

After key resolution each key in the search AST refers to a leaf node from the MPT. The one exception is if the searched key name is a single wildcard '*'. Expanding such keys would result in a bloated AST because it can match any key name, adding combinatorial overhead to the later steps (particularly if the query specifies multiple filters on '*' key). Wildcard keys are expanded dynamically only at the last step, before search on an ERT.

Step 3: Schema resolution looks up the *schema map* to find a set of schemas that match the record structure implied by the query. It first transforms the AST into OR of ANDs form (i.e. sum of products). The key insight is that for an AND to ever be true, all of its children must exist together in a schema. We implement this check by performing an intersection between the set of MPT node IDs of the children of an AND node with each schema. If the intersection is empty, the entire AND sub-tree is removed by treating it as false, and we propagate this false along the AST.

The last AST in Figure 9 shows the AST after schema resolution. The rightmost AND expression matches no schemas and can be removed since MST node 5 and 10 never appear in the same schema. In this case we are able to narrow down the ambiguous query to a single schema, schema 0, by only searching the schema metadata.

Step 4: Search on strings. Next μSlope searches the dictionaries on relevant string filters. Search needs to be performed over the log type dictionary, the variable dictionary, or the timestamp dictionary. Searches on log-text are handled the same way as CLP would. The ability of the dictionaries to reject string queries is critically important for performance. Consider the query *:<uuid>, which is commonly issued at Uber. In archives that do not contain this uuid, this query can be terminated early after searching the variable dictionary, avoiding any column scan. In general, an empty dictionary search would result in the AST node being eliminated, and the false value gets propagated to further simplify the AST.

Step 5: Column decompression and scan is guided by the remaining nodes in the AST. Specifically, the remaining AST tells us exactly which ERT(s), and which column(s), should be scanned. This minimizes the decompression and scan.

Note that we also add a simple timestamp range index at the archive level. This is used to avoid having to decompress and scan any data in the archive when there is no overlap with the time range specified by the query.

7 Implementation

The implementation of μSlope closely follows the design specified in the previous sections. However, some details not called out in the design are critical to the overall performance of the system, so we highlight them here.

We have written a custom JSON serializer in order to improve decompression and search speed. With each schema precisely defining the structure of a log record, μSlope is able to generate a bytecode that describes how to reconstruct records in terms of the columns they have been split into. Unlike JSON serializers designed for dynamic objects, our serializer doesn't require the creation or traversal of mutable in-memory data structures during serialization. Instead, it uses the bytecode generated at the table granularity to directly append the values to the serialization buffer. This approach has proven several times faster than conventional JSON serializers based on our experience.

Given that μSlope can sometimes produce archives containing many small ERTs, minimizing the overhead of storing and loading ERTs is crucial. In μSlope, ERTs are concatenated together into a single file, with a metadata file that describes the location of each ERT within the file and the number of log records within each ERT, which reduces I/O overhead. During search, μSlope scans the AST to determine which ERTs need loading, and then loads them following the storage order, thus eliminating random I/O. Additionally, it optimizes bytecode generation by only producing bytecode for serializing records from an ERT after finding at least one matching record.

8 Evaluation

We implement μSlope with about 18k lines of C++ code. We evaluate the performance of μSlope on both Uber logs and public logs. Specifically, we focus on the following aspects: (1) the compression ratio and speed; (2) the query performance; (3) worst case performance on synthetic logs; (4) the scalability of μSlope on large-scale logs.

8.1 Experiment Setup

Overall we conduct the experiments in two setups: (1) single-thread, single-process experiments on smaller datasets (referred as single-thread experiments), and (2) parallel setups on larger scale Uber's logs. For (1) we compare μSlope with a number of other SSDMSes. The logs are from the same services as described in §3, except that here we increased the data size. In addition, we include logs from five public software, which are generated by running HiBench [24] and YCSB [14] benchmarks. Table 1 shows the size and the number of records of each log dataset. These datasets are relatively small because (1) we had problems to ingest larger data to some of the tools we compare with (for example the ingestion throughput for Elasticsearch is 5MB/s), and (2) μSlope is embarrassingly parallel, therefore its single-threaded performance is the most critical. We also evaluate μSlope on larger Uber's datasets in §8.5.

Single-thread experiments were performed on a Linux server with Intel Xeon E5-2630v3 processor and 128GB of DDR4 memory. Both the uncompressed and compressed logs are stored in a distributed file system (MooseFS) running on multiple 7200RPM SATA HDDs.

We compare μSlope with SSDMSes including CLP 0.0.2, MongoDB 6.0.5, PostgreSQL 15.2, ClickHouse 23.3.1.2823, Elasticsearch 8.6.2, Zstandard 1.4.9 and XZ Utils (for LZMA compression) 5.2.2. (we were informed that Steed [30]'s artifact isn't yet available upon contacting the authors). MongoDB and PostgreSQL have native JSON support (i.e. BSON and `jsonb` data type respectively). For ClickHouse, we explore three setups to store JSON records: (1) in pair-wise arrays which was described in §2.2. Here we only use two arrays and do not differentiate types of the values. (2) in a single string field, which can be parsed by ClickHouse functions. Both setups are commonly used in practice for JSON management. (3) in a single JSON field, which is a new experimental data type introduced in v22.3. It can infer the schema of a JSON record and store every field in a separate file automatically.

	Name	Uncompressed Size	Number of Records
Uber Logs	LogA	30.0GB	22,996,492
	LogB	47.1GB	16,606,964
	LogC	60.4GB	15,306,125
	LogD	50.7GB	58,309,754
	LogE	91.8GB	22,345,071
	LogF	102.9GB	17,251,752
	LogG	30.9GB	3,046,845
	LogH	30.8GB	11,461,221
	LogI	39.7GB	27,209,375
	LogJ	36.0GB	13,605,274
	LogK	30.2GB	57,919,224
	LogL	37.1GB	45,827,554
	LogM	36.5GB	42,206,452
	LogN	38.0GB	22,307,407
	LogO	38.6GB	4,438,786
	LogP	38.3GB	34,840,347
Public Logs	Spark	2.0GB	1,011,651
	MongoDB	64.8GB	186,287,600
	CockroachDB	9.8GB	16,520,377
	elasticsearch	8.0GB	140,012,234
	PostgreSQL	392.8MB	1,000,000

Table 1: Log datasets used in our experiments.

For these systems we do not create any index for a fair comparison with μSlope, because μSlope is designed to be an archival SSDMS and does not have any external index. MongoDB automatically builds an index on the default key `_id` and we exclude the size of the index when calculating compression ratio.

We also compare μSlope with two general-purpose compressors Zstandard and LZMA. Zstandard is the underlying compression method for μSlope and LZMA is known for its high compression ratio.

We do not evaluate search on CLP because wildcard queries (which CLP supports) are incompatible with semi-structured data model. For example, a KQL query `error: "*404"` searches for the "error" field whose value ends with "404"; but CLP could return records like {"error": "0", "keyx": "404"}, because '*' could match an arbitrary amount of text. We do not evaluate search on Elasticsearch because we cannot ingest the three datasets where we designed query benchmarks on into Elasticsearch. Nevertheless, as we will show that Elasticsearch consumes too much storage space that cannot be used as an archival SSDMS.

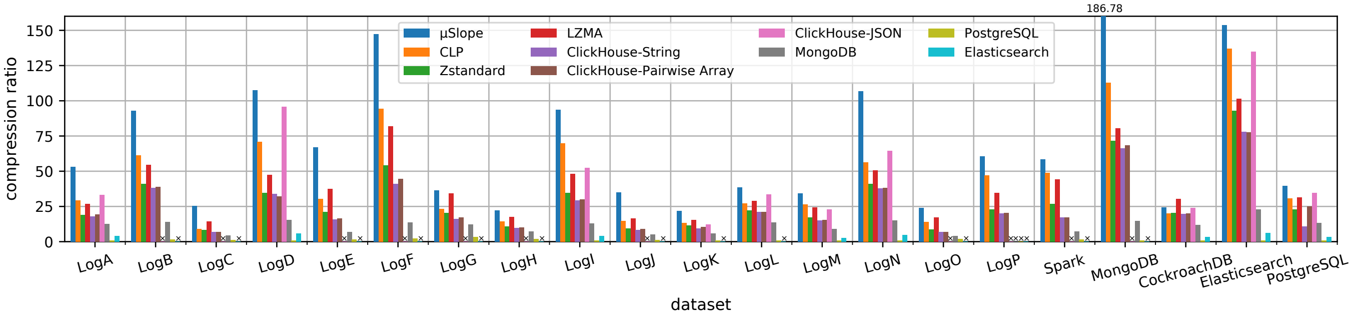


Figure 10: Compression ratio of μSlope and other tools on different JSON datasets. A "×" means the dataset cannot be ingested by the tool.

8.2 Compression Ratio and Speed

Figure 10 shows the compression ratio of μSlope and other tools on different datasets. The compression level of Zstandard is set to 3 (default) for μSlope. For a fair comparison, we use the same compression method and level for CLP, ClickHouse and MongoDB. PostgreSQL and Elasticsearch do not support Zstandard compression, so we use lz4 [32] instead. For LZMA, we use the default compression level 6.

Note that out of 21 datasets, ClickHouse-JSON can only ingest 10 and Elasticsearch can only ingest 9. The most common reason is that they cannot accept fields with the same key name but different types. Additionally, MongoDB and PostgreSQL cannot ingest one dataset because of the escape character. For the average compression ratio and speed comparison, we only include the datasets that are successfully ingested by these systems.

μSlope achieves the highest compression ratio on all JSON datasets. The average compression ratio of μSlope is 68.1:1, ranging from 21.9:1 (LogK) to 186.8:1 (MongoDB). On average, μSlope's compression ratio is 2.75x of ClickHouse-String's, 2.62x of ClickHouse-Pairwise Array's, 1.34x of ClickHouse-JSON's, 6.10x of MongoDB's, 16.50x of PosgreSQL's, and 15.71x of Elasticsearch's. It surpasses Zstandard's, LZMA's, and CLP's compression ratios by factors of 2.34x, 1.70x, and 1.50x respectively. but it is only 4.8% on average, which still makes μSlope's compression ratio the best among all.

We delve into the breakdown of compressed data size in μSlope, using LogP as an example. Out of the 710 MB total compressed data, the MPT and schema map only occupy 3.6KB and 1.9KB, respectively. Dictionaries account for 26.3% of the storage space, with the remaining 73.7% attributed to compressed columns of ERTs.

Figure 11 shows the average ingestion speed on all datasets. μSlope's ingestion speed is slower than ClickHouse-String and ClickHouse-Pairwise Array because ClickHouse-String directly store the raw JSON string and does not parse it, while ClickHouse-Pairwise Array only parses the top-level fields. In comparison, μSlope parses every field of the entire JSON record. μSlope's ingestion is slighter slower than CLP and faster than all other fully-parsed JSON tools, outperforming ClickHouse-JSON, MongoDB, PostgreSQL and Elasticsearch by 19.3%, 186.7%, 419.8%, 1127.3% respectively. Additionally, it exceeds LZMA's performance by 814.8%.

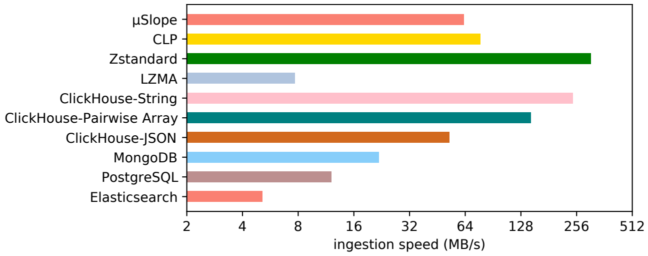


Figure 11: Average ingestion speed (log scale).

8.3 Search Performance

We use 15 queries to evaluate the search performance of μSlope and other tools on Uber LogF, LogO and MongoDB logs. For queries on LogF and LogO, they are the top queries performed in Uber (with repetitive patterns removed). For queries on MongoDB, we try to cover different possible patterns. Table 2 shows the queries in KQL [6]. For ClickHouse and PostgreSQL, we convert KQL queries to SQL queries with their built-in functions and operators. For MongoDB, we use their own query language.

Query B is a special case. It does not specify any search key, but searches for all fields for the matched UUID. Other tools does not natively support this kind of query so we have to convert it to a full-text search instead. It works for Query B, but may get incorrect results for other queries that span across keys and values. MongoDB is required to have a text index on that table to do a full-text search. However, after running for 196 seconds, it reports an error.

We clear the OS buffer cache before every run. This is to simulate search on archival storage. However, by default MongoDB uses about half of the memory (63.5G in our machine) to cache uncompressed data and the cache cannot be cleared, while others use only a minimum amount (or even no) or the cache can be cleared. For a fair comparison, we test MongoDB with the minimum cache size.

Figure 12 shows the query latency of those 15 queries on

Queries for LogF	
A	zone:... AND NOT @reserved.collector.filename:stdout AND runtime_env:staging
B	*: "d...-...-...-...-...9"
C	level: error AND message: d...*
D	timestamp >date("2022-04-14T08:00:00.000") AND timestamp <date("2022-04-14T08:15:00.000")
Queries for LogO	
E	headers.x-tenancy:testing* AND NOT headers.x-tenancy: testing/.../4...-...-...-...6d AND headers.caller-procedure:"fareEstimateV2" AND headers.x-source:public
F	headers.x-region-name:... AND headers.x-tenancy:"production" AND caller:*create*
G	level: error AND NOT @reserved.collector.filename: executor AND runtime_env:production AND partition: compute-... AND instance: 3...5 AND mesos_executor_id: t...5-6
H	level: error AND message: "Error handling inbound request."
I	glue.handler.method: get_ranked_products AND env: production AND level: error
Queries for MongoDB logs (public dataset)	
J	attr.tickets: *
K	id: 22419
L	attr.message.msg: log_release* AND attr.message.session_name: connection
M	ctx: initandlisten AND (NOT msg: "WiredTiger message" OR attr.message.msg: log_remove*)
N	c: WTWRTLOG AND attr.message.ts_sec >1679490000
O	ctx: FlowControlRefresher and attr.numTrimmed: 0

Table 2: Queries used in our experiments. “...” is used to anonymize (part) of the actual values.

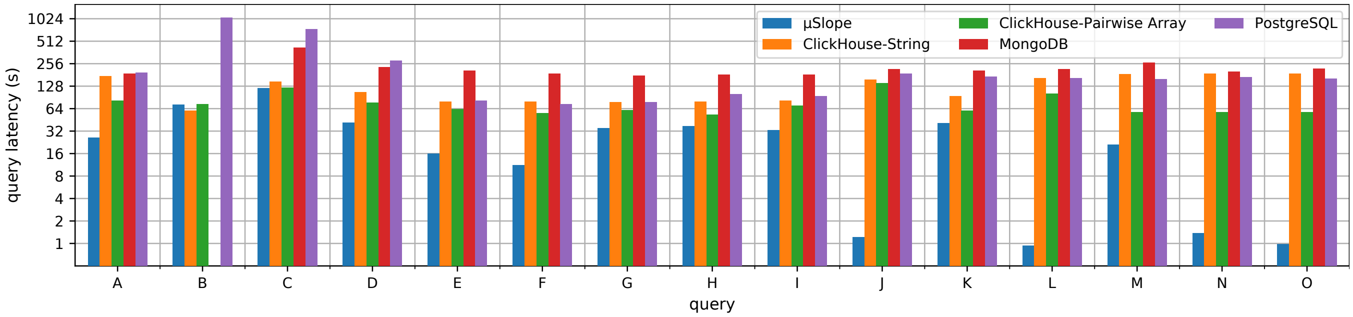


Figure 12: Query latency of μSlope and other tools (log scale).

different tools. It includes both the search time and the time to write results to the disk. On average, the speed of μSlope is 2.47x of the fastest setup of ClickHouse (ClickHouse-Pair Array), 8.09x of PostgreSQL’s, and 6.74x of MongoDB’s.

μSlope outperforms all other tools on 14 queries. The fast search performance comes from its use of metadata (MPT and schema map). For example, μSlope outperforms all other tools by at least 116x on Query J. This query checks the existence of a key and returns all the records containing that key. In this case, there are only a small number of schemas that contain this key, so after μSlope checks its MPT and schema map, it only needs to decompress a small number of ERTs. For other tools, they will have to scan nearly the entire dataset. Query L, N, O are also similar as there are only a small number of matching schemas and μSlope only needs to decompress small ERTs. Note that even for these queries, the schema metadata lookup is not the bottleneck. For Query J, for example, searching the MPT, schema map and ERTs only accounts for 5.5% of the total query time and loading dictionaries accounts for 73.0%. An even more extreme case is that μSlope can return no-match right after the MPT and schema map search, because, say, the query searches for a key that doesn’t exist. In fact, our query benchmark does not even contain such best-case scenario for μSlope.

For Query B, μSlope is slower than ClickHouse-String and ClickHouse-Pairwise Array because μSlope needs to scan all the ERTs and decode them. , while the two ClickHouse setups can perform a full-text search on the raw JSON string values, without the need to decode them.

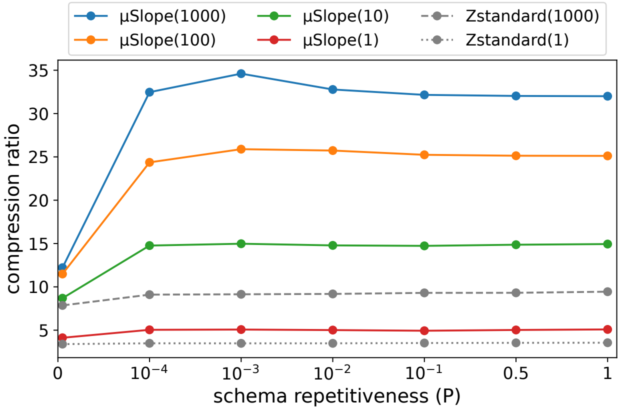


Figure 13: Compression ratio of μSlope against Zstandard on synthetic logs. The number enclosed in brackets within the legend represents the repetition ratio of variable values.

8.4 Synthetic Evaluation

The efficiency of μSlope relies on the repetitiveness of schemas and variable values. To demonstrate the boundaries of μSlope’s capabilities, we evaluate its compression and search performance on a corpus of synthetic log data, which

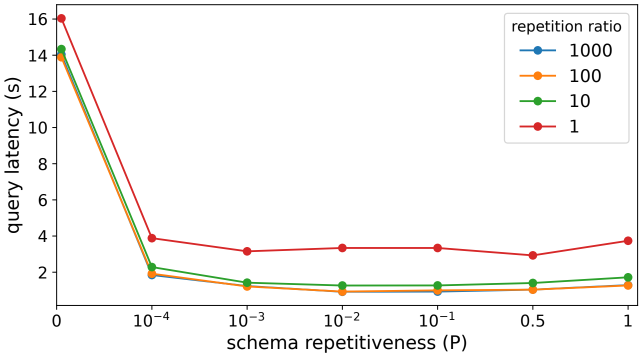


Figure 14: Query latency of μSlope for needle-in-haystack wildcard queries on synthetic logs.

varies in both repetitiveness of schemas and variable values.

Each synthetic dataset contains 1GB of data, consisting of 670K records. Each record has 20 fields. Every key and value is a UUID. We use UUIDs because they are one of the most common source of noises (i.e. non-repetitiveness) in the real logs. To vary the repetitiveness of variable values, the logs are generated with repetition ratios (§3) of variable values ranging from 1 to 1000, achieved by randomly drawing values from a uniform distribution. Specifically, we use four repetition ratios: 1, 10, 100, and 1000.

To vary the repetitiveness of schemas, we draw schemas from a power law distribution. Specifically, the n-th most frequent schema appears in $P \times (1-P)^n$ of the records (where n starts from 0). P is a value within the range of $(0,1]$, and it is a constant within one dataset. For example, the most frequent schema ($n=0$) appears in P of the records, the next most frequent schema ($n=1$) appears on $P \times (1-P)$ of the records, and so on. We use a total of 7 different P values as shown in Figure 13. The degree of schema repetitiveness increases with P. When P is the smallest every log record has a unique schema; when it increases to 1, all records have the same schema.

In total, we generate 28 (4 different repetition ratios combined with 7 different P values) synthetic datasets each with a different combination of schema repetitiveness and repetition ratio of the variable values. Figure 13 illustrates the interplay among compression ratio, schema repetitiveness, and repetition ratio of variable values. In all scenarios, μSlope outperforms Zstandard, with the compression ratio increasing as the repetition ratio of variable values increases. In the extreme case where P approaches 0, the compression ratio drops notably. This is attributed to each log record having a unique schema. The small tables and extra metadata overheads lead to a significant reduction in the compression ratio. However, the compression ratio quickly increases as P increases to the next smallest value (10^{-4}) and remains relatively stable.

We evaluate the search performance using a needle in the haystack query. One variable value is replaced with a fixed UUID value known as the "needle", so we can benchmark the query `*: "needle-value"`. The performance of this query is influenced by both the repetition ratio of variable values and schema repetitiveness. Lower repetition ratio leads to larger variable dictionaries, which are required to be loaded before decompression and scan, introducing a constant overhead before any results can be returned. This explains the consistent gap between each curve in Figure 14. As a result μSlope sometimes struggles to achieve low response time for datasets with a low repetition ratio, although this challenge can be partially addressed in practice by generating smaller archives.

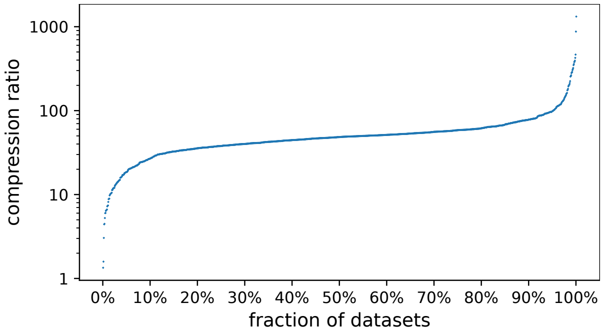


Figure 15: Compression ratio distribution of 1378 datasets in Uber's production logs.

Figure 14 shows a significant decline in query performance as P approaches 0. This is because in this extreme case where each record has a unique schema, we have a large number of small Encoded Record Tables where each has only one record. This significantly slows down the decompression and scan as we need to load a large number of small tables, and each table is decompressed using a different Zstandard stream.

8.5 Scalability Evaluation

We evaluate μSlope on 434TB of production logs from Uber representing 1,378 datasets, and select a 26.2TB subset from Uber's LogF to evaluate search scalability. This production dataset achieves an average compression ratio of 30.5:1, Figure 15 shows the compression ratio for each of the 1378 dataset in sorted order. The outliers with low compression ratio typically contain large amounts of random non-repeating binary data such as base64 encoded binary data and UUIDs. For example the index with the worst compression ratio has a column which appears to contain several megabytes of base64 encoded binary data in each log message.

To evaluate the scalability of search we run queries A-G from Table 2 on 26.2TB of Uber's LogF data with increasing amounts of parallelism. Values in the queries have been changed to match this dataset where appropriate. Results for query F have been omitted because its characteristics are identical to query E on this dataset.

Experiments are run across 8 containers, each has access to 96 cores, 2TB of network attached SSD, and 32GB of RAM.

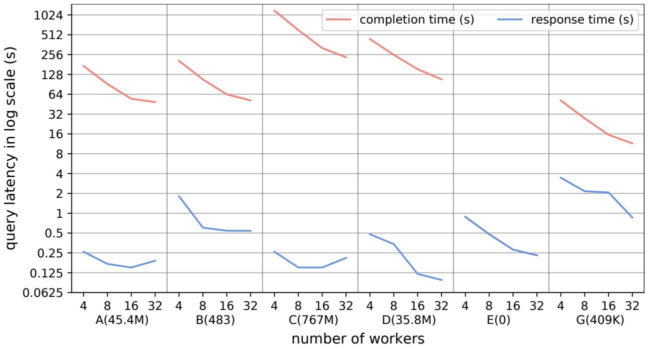


Figure 16: Query completion time and response time of μSlope (log scale) using 4, 8, 16 and 32 workers per container. The number of matching records for each query is indicated at the bottom.

There are a total of 2,155 archives after compression. These archives are evenly distributed across the 8 containers. The maximum skew between any two containers is 11% in both data size and in number of archives.

The searches are run using 4, 8, 16, and 32 workers per container. Each archive is processed by a single worker process. We show two results: (1) end-to-end completion time including marshalling all matching records, and (2) time to get the first matching record, i.e. response time. This experiment was conducted before we implemented our optimizations for JSON marshalling, so archives which contain many matching results can have an outsized impact on overall query performance. Figure 16 shows how the search performance scales with the increase of number of workers per container.

All of the queries scale well to 16 workers but have limited scalability to 32 workers. This limit is imposed by different kinds of skew in the dataset. For example in Query A a single archive becomes a bottleneck for completion time because it returns 5.9x more results than average, and in Query D all 19 archives with matching results happen to be allocated to the same container. Query E is extremely fast because it only needs to consult the MPT before terminating.

In practice, we manage this sort of skew within a dataset by producing smaller archives.

9 Limitations and Future Work

μSlope is a system designed for storing and searching archival semi-structured log data. It is not suitable for data that can be updated or deleted. Since μSlope leverages the repetitive nature of logs to achieve a high compression ratio, if the data has too many different schema structures or values are unique overall, μSlope may not be able to achieve a high compression.

As for search, μSlope provides support for basic queries, including term search, field search, wildcard search, and range search. However, currently it lacks support for more complex queries like joins. Besides, μSlope may struggle with queries that necessitate scanning the entire dataset and generating a large number of results.

The current implementation of μSlope compresses each table into its own Zstandard stream. We plan to implement optimizations to combine small tables into fewer streams (to improve compression ratio and amortize the cost of decompressing each small table), and split large tables into several streams by columns (to avoid decompressing columns in large tables not being searched on unless necessary). We also plan to improve scan performance and support more aggregation operators in the future.

10 Concluding Remarks

This paper presents μSlope, a resource efficient system for semi-structured log management that losslessly compresses the log data, and enables search without full decompression. Its design is guided by a careful analysis on the characteristics of real-world semi-structured logs and their query patterns. μSlope does not require any user annotation, and can automatically handle the dynamic schema structures. Our evaluation shows that μSlope achieves unprecedented compression ratio of up to 186.8:1, and its search speed is at least 2.47x of the fastest existing SSDMSes. μSlope is available at https://github.com/y-scope/clp.

Acknowledgements

We thank our shepherd, Andrew Warfield, and the anonymous reviewers for their insightful feedback and comments. Michael Stumm and Ashvin Goel have provided valuable feedback on an early draft of the paper. Devin Gibson has been partially supported by an NSERC Alliance Missions grant and a QEII-GSST scholarship.

References

[1] Amazon CloudWatch: Observe and monitor resources and applications on AWS, on premises, and on other clouds. https://aws.amazon.com/cloudwatch/, 2024.

[2] BSON. https://bsonspec.org, 2024.

[3] BSON example: How BSON is stored in MongoDB database. https://www.mongodb.com/basics/bson, 2024.

[4] Clickhouse. https://clickhouse.com, 2024.

[5] Grafana Loki. https://grafana.com/oss/loki/, 2024.

[6] KQL: Kibana Query Language. https://www.elastic.co/guide/en/kibana/current/kuery-query.html, 2024.

[7] Log-surgeon: a performant log parsing library. https://github.com/y-scope/log-surgeon, 2024.

[8] PostgrSQL JSON Types. https://www.postgresql.org/docs/current/datatype-json.html, 2024.

[9] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf, 2024.

[10] Search taking a lot of time using CLP. https://github.com/y-scope/clp/issues/154, 2024.

[11] simdjson. https://github.com/simdjson/simdjson, 2024.

[12] The JSON Data Interchange Standard. https://www.json.org/json-en.html, 2024.

[13] The Number of tweets per day in 2022. https://www.dsayce.com/social-media/tweets-day/, 2024.

[14] Yahoo! Cloud Serving Benchmark. https://ycsb.site/, 2024.

[15] YAML: YAML Ain't Markup Language. https://yaml.org/, 2024.

[16] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682. ACM, 2006.

[17] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[18] Konrad Beiske. Six Ways to Crash Elasticsearch, September 2014. https://www.elastic.co/blog/found-crash-elasticsearch#mapping-explosion.

[19] Craig Chasseur, Yinan Li, and Jignesh M Patel. Enabling json document stores in relational systems. In *WebDB*, volume 13, pages 14–15, 2013.

[20] Dominik Durner, Viktor Leis, and Thomas Neumann. Json tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 445–458. ACM, 2021.

[21] Elasticsearch B.V. Elasticsearch, 2024. https://www.elastic.co/guide/en/elasticsearch/reference/8.7/index.html.

[22] Facebook, Inc. Zstandard, 2024. https://facebook.github.io/zstd/.

[23] Free Software Foundation, Inc. GNU Gzip, August 2024. https://www.gnu.org/software/gzip/.

[24] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[25] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. Json data management: Supporting schema-less development in rdbms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1247–1258. ACM, 2014.

[26] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Ying Liu, and Hui Joe Chang. Closing the functional and performance gap between sql and nosql. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, page 227–238. ACM, 2016.

[27] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 21)*, pages 183–198. USENIX, July 2021.

[28] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[29] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '14, pages 815–826. ACM, 2014.

[30] Zhiyi Wang and Shimin Chen. Exploiting common patterns for tree-structured data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 883–896. ACM, 2017.

[31] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys' 23)*. ACM, 2024.

[32] Yann Collet. LZ4, 2024. http://lz4.github.io/lz4/.

[33] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing

Mike Chow[1], Yang Wang[1 †], William Wang[1], Ayichew Hailu[1], Rohan Bopardikar[1],
Bin Zhang[1], Jialiang Qu[1], David Meisner[1], Santosh Sonawane[1], Yunqi Zhang[1],
Rodrigo Paim[1], Mack Ward[1], Ivor Huang[1], Matt McNally[1], Daniel Hodges[1],
Zoltan Farkas[1], Caner Gocmen[1], Elvis Huang[1], and Chunqiang Tang[1]

[1] Meta Platforms
[†] The Ohio State University

Abstract

This paper presents ServiceLab, a large-scale performance testing platform developed at Meta. Currently, the diverse set of applications and ML models it tests consumes millions of machines in production, and each year it detects performance regressions that could otherwise lead to the wastage of millions of machines. A major challenge for ServiceLab is to detect small performance regressions, sometimes as tiny as 0.01%. These minor regressions matter due to our large fleet size and their potential to accumulate over time. For instance, the median regression detected by ServiceLab for our large serverless platform, running on more than half a million machines, is only 0.14%. Another challenge is running performance tests in our private cloud, which, like the public cloud, is a noisy environment that exhibits inherent performance variances even for machines of the same instance type. To address these challenges, we conduct a large-scale study with millions of performance experiments to identify machine factors, such as the kernel, CPU, and datacenter location, that introduce variance to test results. Moreover, we present statistical analysis methods to robustly identify small regressions. Finally, we share our seven years of operational experience in dealing with a diverse set of applications.

1 Introduction

In our hyperscale private cloud, tens of thousands of services run on millions of machines to serve billions of users, and engineers make thousands of code changes to these services daily. Performance or resource usage regressions caused by these changes may impact user experiences or even cause a site outage. Therefore, engineers critically rely on automated performance testing to catch regressions early.

Consider, for example, the frontend serverless platform called *FrontFaaS*. More than ten thousand engineers write code on this platform, with thousands of code changes committed daily and a new version released into production every three hours. If a code change causes even just a 0.01% regression in the platform's overall CPU usage, an alarm is raised. To our knowledge, strict thresholds of this level have not been studied before. We use this strict threshold because FrontFaaS consumes more than half a million machines and 0.01% would mean more than 50 machines. Moreover, if left undetected, many small regressions would accumulate over time. Each year, we catch regressions in FrontFaaS that amount to the capacity of more than one million machines.

This paper presents our performance testing platform called *ServiceLab*. It currently tests about one thousand diverse services and ML models, which, in aggregate, consume millions of machines in production. Although performance testing is widely used, there is no detailed report of its usage at hyperscale. Specifically, we have encountered several challenges that have not been studied before:

1. How to run tests on heterogeneous machines provided by the cloud while still ensuring comparable results?
2. How to detect regressions as small as 0.01%?
3. How to support hundreds of diverse services with one uniform testing platform?

We elaborate on each of these challenges below.

Use heterogeneous cloud machines. To detect small regressions, we must conduct numerous trials for an experiment and then apply statistical analysis. Since running these trials sequentially on one machine can take a long time (a trial takes over one hour on average), a natural solution is to run them in parallel on many machines. Ideally, these machines should be identical to reduce performance variance.

However, when a test workload is launched on a cloud, the cloud chooses machines to run the workload and even machines of the same instance type exhibit varying performance [47], due to differences in SSD wearing, memory chips from different vendors, and varying frequencies of CPU's uncore components like memory controller, etc. This phenomenon not only exists in public clouds but also in our private cloud that we use to run testing workloads. Note that

Contributions: Yang wrote the majority of the paper, followed by Mike and Chunqiang. Mike led the development of ServiceLab for multiple years, and other authors also made major contributions to its development.

our private cloud runs workloads on Linux containers instead of virtual machines (VMs) so there are no performance variances caused by VMs.

Although it is theoretically possible to reduce performance variance by maintaining our own dedicated pool of identical physical machines for testing, it is impractical for two main reasons: (1) testing workloads are spiky, and running them as on-demand workloads in the cloud is more cost-effective, and (2) maintaining a dedicated pool of tens of thousands of machines for testing requires an operations team that we cannot afford, which is exactly the problem that clouds aim to solve anyway.

Like in a public cloud, we can provision a batch of machines, keep a subset of "*nearly identical machines*" to run test workloads, and return the rest. The key question is how to select "*nearly identical machines*." Specifically, among the factors affecting a machine's performance, which are crucial for machine selection, and which can be ignored and addressed through statistical analysis?

To answer this question, we conducted a large-scale study with millions of performance experiments on various machines, using both microbenchmarks and real-world applications. We find that the performance variance on two machines is comparable to that on a single machine if the two machines share the same instance type, CPU architecture (e.g., Intel Cooper Lake), and kernel version, are located in the same datacenter region, and have CPU turbo disabled. An interesting observation is that the datacenter location matters, while other factors such as RAM vendor and RAM speed are less important. We will delve into this in §4.

Detect small regressions. For large services that consume tens of thousands of machines, we need to detect regressions as small as 0.01% while maintaining a low false positive rate. A high false positive rate not only wastes engineers' time in unnecessary debugging but also leads to engineers distrusting and ignoring the warnings even when they are correct. Our experience indicates that there is no one-size-fits-all statistical model that can accurately detect regressions for all services, due to the different outlier patterns of these services. To address this issue, we leverage multiple statistical models simultaneously and evaluate their false negatives and false positives on historical data to select the best model for each service. Although this ensemble approach may seem conceptually simple, we will discuss the intricacies of applying it at scale in highly noisy production environments.

Support diverse services. Our private cloud runs numerous services with intricate interdependencies, a complexity shared with other hyperscalers [30,38,48]. A single testing solution capable of covering all these services likely does not exist. Can we achieve the next best thing, i.e., having a single solution to cover the majority of code changes submitted by engineers? ServiceLab indeed accomplishes this. Currently, as a general-purpose testing platform, it covers more than half of the total code changes, surpassing the combined coverage of other specialized testing platforms.

ServiceLab takes the record-and-replay approach for testing, with three key distinctions. First, unlike past solutions that emphasize deterministic replay [8,20,24,62], ServiceLab replays requests captured from a production system (PS) to a system under test (SUT) without expecting the SUT to exhibit the same behavior as the PS. In fact, due to testing changed code, it is anticipated that the SUT may make outgoing calls to downstream services that differ from those made by the PS. Therefore, ServiceLab does not replay the responses from downstream services to the SUT.

Second, ServiceLab allows the SUT to call downstream services running in production, provided there are no adverse side effects. Although users can set up a group of interdependent services in ServiceLab to create a self-contained testing environment without relying on the production environment, this approach is not consistently implemented due to practical reasons. For instance, making a per-test replica of certain massive datasets accessed by the SUT, such as the social graph for billions of users, is economically impractical.

In ServiceLab, the SUT can call downstream production services, and most of those calls do not incur side effects, as they are read-only or idempotent. If a SUT's call to a downstream service does cause side effects, ServiceLab provides a mock framework to assist the SUT in mitigating it. For example, instead of writing to a production database, the writes can be redirected to a test database.

Third, due to the complexity of hyperscale services, ServiceLab does not attempt to provide a simple but inflexible solution that requires no involvement from service owners, because such a solution would only work for a small fraction of services. Instead, ServiceLab allows and encourages the service owner's participation. For example, when testing a sharded stateful service, it is the service owner's responsibility to populate the necessary states before the test starts.

With the three key distinctions above, while ServiceLab's record-and-replay approach may necessitate occasional involvement from the service owner and does not extend to certain complex services, it effectively covers the majority of code changes submitted by engineers.

Contributions. We make the following contributions:

- We address the performance variance issue arising from running tests in the cloud. Specifically, we conducted millions of experiments to identify the factors that contribute most significantly to performance variance across machines. Such a large-scale study has not been reported before.
- We develop statistical analysis methods to robustly identify performance regression as small as 0.01%, even when tests do not use identical machines. This represents a significant refinement of existing methods, as no prior research has achieved this level of a low threshold.
- This is the first holistic report of a hyperscale testing plat-

form, including its design and our seven years of operational experience in dealing with a diverse set of applications.

2 ServiceLab from a User's Perspective

Before presenting the internals of ServiceLab, we first describe its usage from a user's perspective. ServiceLab can be used in different testing modes. The *efficiency mode* tests a code or configuration change's impact on key metrics such as latency or CPU/memory usage. The *capacity mode* tests a code or configuration change's impact on the maximum throughput that can be achieved, which affects the amount of capacity needed to run the service. The *hardware mode* compares the performance of different hardware running the same code. Below, we focus on the efficiency mode.

2.1 ServiceLab in the Development Workflow

Figure 1 depicts our development workflow, where ServiceLab is involved in the review-time test, commit test, deployment test, and config test. We elaborate on them below.

Meta uses the monorepo approach [12] to store code for all projects in one repository. When developing a new feature for an application, the developer clones the repository and makes local changes without affecting others. Once the code is ready, they submit the change, referred to as a *diff*, for peer review. Both functional and performance tests are automatically executed for the diff. The peer reviewer examines the code and test results, requesting changes as needed before approving the diff. Upon approval, the developer commits the diff, triggering post-commit tests.

On a set schedule, the continuous-deployment tool compiles a new executable for the application and creates a release candidate (RC). It conducts tests to compare the RC with the executable running in production. The RC is abandoned if a regression is identified. Otherwise, it is deployed into production in stages, and the application's health metrics, including performance metrics, are monitored continuously. If any health issue is detected, the deployment is reverted.

A common practice is to use a configuration parameter known as a *gate* to control access to the new code path. Initially, the gate is disabled so that the application continues to execute the old code path even after the new release is deployed. Then, the developer makes a remote configuration change to toggle the gate, enabling the application to execute the new code path. If any issues arise, the gate can be instantly disabled to revert back to the old code path without requiring a new code release.

2.2 Setting Up Tests with ServiceLab

To register a system-under-test (SUT) with ServiceLab, the application owner provides the following information:

- The selection criteria for code or configuration changes to trigger a test (note that it may not be necessary to run tests on every change);
- A container manifest that specifies the executable to test and how to set up Linux containers to run the executable;
- The metrics to be aggregated at the end of a test run, and the condition to fail the test;
- A traffic-recording configuration that instructs the RPC system how to sample production traffic for later replay;
- The rate at which the recorded traffic will be replayed during a test run.

ServiceLab supports both synthetic and record-and-replay traffic for testing, but primarily relies on the latter because it more accurately represents the production system. This approach records live production traffic and then replays it on a separate application instance in the testing environment, which may run modified code. It is the application owner's responsibility to ensure that a replay in the testing environment does not cause undesirable side effects on the production system. Moreover, a stateful application needs to set up its state properly so that it can handle the replayed traffic.

Once a SUT is registered at ServiceLab, it undergoes tests in four phases. The *build* phase compiles all required code into a package. The *allocation* phase acquires necessary machines from the cloud. The *running* phase initiates the target application on the allocated machines and replays the recorded workload. The *analysis* phase conducts statistical analysis on the results to draw a conclusion.

3 Applications Tested by ServiceLab

Currently, ServiceLab tests about one thousand diverse services and ML models, and their collective capacity consumption in production amounts to millions of machines. We describe several representative and large workloads below.

3.1 FrontFaaS Serverless Platform

FrontFaaS is one of the most complex software ecosystems in our private cloud. It is a serverless function-as-a-service (FaaS) platform that runs on more than half a million machines and has tens of thousands of developers making changes to its code base, with thousands of code commits every workday. ServiceLab tests FrontFaaS to detect CPU usage regression as small as 0.01%. It holistically tests different aspects of FrontFaaS: its PHP runtime called HHVM [27], the FaaS code written by tens of thousands of developers, and that code's impact on downstream services like databases.

Testing the language runtime. HHVM performs just-in-time (JIT) compilation for efficient execution. The HHVM team relies on ServiceLab to collect performance signals on compiler optimizations, monitoring metrics such as instructions per cycle, execution time, and cache misses. In addition to the core code written by the HHVM team, HHVM links with many libraries developed by other teams, any of which may cause regressions. HHVM tests compare the code running in production with the code in the *trunk* (i.e., the latest code

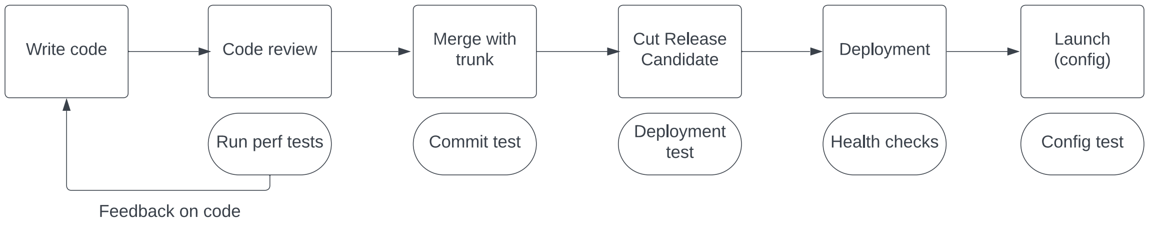


Figure 1: Code change workflow. In this figure, *tests* refer to both functional and performance tests, but this paper focuses on performance tests.

in the monorepo shared by all developers), enabling developers to catch regressions before a new release candidate is created. If a regression is detected, ServiceLab uses bisection to identify the root cause.

Selecting diffs to test. In addition to testing HHVM, ServiceLab also tests FrontFaaS' application-level FaaS code. FrontFaaS is the primary entry point for user-facing traffic for our products and runs thousands of unique application endpoints. With thousands of FaaS code changes (diffs) occurring every workday, it is not cost-effective to test every change. Moreover, since a change is unlikely to affect all thousands of application endpoints, it is unnecessary to replay the traffic for all those endpoints during a test.

A ServiceLab component called *DiffSuggester* selects which diffs to test based on a calculated *impact score* and also determines the traffic for which endpoints to replay during a test. DiffSuggester traverses the compiler's abstract syntax tree to identify functions modified by the diff. It calculates an impact score for each modified function by leveraging a profiling dataset of FrontFaaS' execution in production to estimate the global cost of the function, considering both its execution frequency and resource consumption per invocation. The diff's impact score is simply the sum of the impact scores for all impacted functions. If a diff's impact score is above a threshold, ServiceLab will run experiments for it. The threshold is statically chosen based on the number of machines available to run experiments and the distribution of diffs' impact scores. Moreover, DiffSuggester also uses the production profiling data to infer which application endpoints are impacted by the diff and selectively replay traffic for those endpoints with the right proportion.

Dealing with side effects. Because of the complexity of FrontFaaS, it is too costly to set up an entirely isolated testing environment for it. It invokes hundreds of downstream services, which recursively have their own dependencies. All these are hard to replicate in a testing environment and keep them faithful to the production environment. Moreover, given numerous concurrent tests, it is economically impractical to make a per-test copy of certain massive datasets accessed by FrontFaaS, such as the social graph for billions of users.

Therefore, ServiceLab allows a test instance of FrontFaaS to call downstream services running in production and carefully manages any adverse side effects. The non-functional side effects, such as test-induced load on downstream production systems, is not a concern because that test load is negligible compared to the production traffic from billions of users. The functional side effects, such as writing to a production database, is the main concern and is managed carefully.

By default, FrontFaaS' writes to databases, caches, and data warehouses are automatically dropped via a shim layer in the client libraries, while reads to these production systems are allowed. Unlike data stores where differences between read and write can be easily identified, for generic RPC calls, ServiceLab and the RPC system cannot easily infer whether an RPC method has undesirable side effects or not. Therefore, the RPC system drops calls to downstream production systems by default to ensure safety, while users can provide a list of specific RPC calls that are allowed to proceed. However, this method may prevent certain code paths from being executed and result in ServiceLab missing the opportunity to detect regressions on those code paths. If the owners of certain FrontFaaS endpoints really want to cover those code paths, it is their responsibility to modify the code's behavior so that it can run in ServiceLab to exercise those code paths without causing adverse side effects to production systems. We will delve into this in §5.3.

Testing performance impact on downstream services. A FrontFaaS diff may not cause regressions in the resource usage of FrontFaas itself but may regress in the load it imposes on downstream services. Specifically, the social graph database (*TAO* [11]) is one of the most important downstream services for FrontFaaS, and ServiceLab also detects increased reads to TAO. During a test, ServiceLab monitors the number of read requests that FrontFaaS issues to TAO when processing a replayed end-user request. Statistics are gathered at the granularity of each type of end-user request because the number of reads to TAO may vary widely depending on the type of the end-user request. Similar to reporting regressions on FrontFaaS' own metrics, ServiceLab also reports regressions in reads to TAO.

3.2 Sharded and Stateful Services

LASER is a low-latency key-value store that is frequently accessed by FrontFaaS on the critical path of serving user requests. LASER primarily serves as an indexing service for data in the data warehouse. Its index can be updated either by real-time stream processing that extracts data from data streams or by running daily MapReduce computation to build the index and then performing a bulk load into the key-value store. LASER is sharded and managed by a shared, central control plane similar to ShardManager [36] and Slicer [4], which dynamically assigns shards to different LASER servers.

Testing LASER faces several challenges. First, to bring up a LASER instance in the isolated test environment, we have to disconnect it from the central shard control plane and specifically instruct it to load certain shards, instead of relying on the control plane's dynamic shard assignment. Second, we allow LASER to perform read-only accesses to the data warehouse to set up its stateful index for testing. Finally, in production, requests routed to a specific LASER shard have an RPC header with the shard ID that matches with the shard; otherwise, the requests would be rejected. LASER uses record-and-replay for testing, which broadly samples requests for different shards. When requests are replayed, ServiceLab dynamically adds an RPC request header that matches with the shard ID of the LASER server under test.

LASER uses three major metrics for regression detection. These metrics and their regression thresholds are CPU usage (2%), anonymous memory usage (5%), and SSD storage usage (5%).

3.3 ML Prediction

MLPredictor is a shared ML deployment platform used by ML engineers to effortlessly deploy and manage thousands of ML models without the need for an understanding of the underlying infrastructure. This "serverless" approach conceals the operational complexities of large-scale distributed systems, which are often unfamiliar to ML engineers.

MLPredictor uses record-and-replay, along with ServiceLab's *capacity mode*, to test performance under varying load levels. ServiceLab incrementally increases the load level of the replayed traffic until MLPredictor breaches its service level objective (SLO), helping identify both the maximum throughput and potential capacity regressions. Initially, we recorded traffic for different models using uniform sampling, leading to an overwhelming number of samples from high-traffic models. Later, we switched to interval-based reservoir sampling [5,57], capping the number of samples for a popular model at a constant per time interval.

MLPredictor uses the maximum requests rate for regression detection, with a threshold of 5%.

3.4 Data Aggregation

DataAggregator is a CPU-intensive backend service that handles all news feed rankings. It is invoked by FrontFaaS upon a user request, and its role is to collect all relevant information about posts and analyze all the features (e.g., how many people have liked this post previously) to predict the posts' values to the user. New releases of DataAggregator are deployed to production multiple times throughout the day, and it primarily uses ServiceLab for release-time testing.

Instead of using record-and-replay, it uses a forker service to duplicate live production traffic and send it to the testing environment in ServiceLab. The forker sends the production system's responses back to users but drops the test instance's responses so that they will not affect users. DataAggregator prefers testing with shadow production traffic instead of recorded traffic because the setup is straightforward for them, and the existence of the forker even predates ServiceLab.

DataAggregator uses 68 key metrics for regression detection. Examples of the key metrics and their regression thresholds include container-level CPU usage (1.25%), process-level CPU usage (0.6%), and p99 memory usage (3%). Some metrics are related to the application logic, e.g., log error or warning counts (5%), no stories returned (2%), and latency to process all stories in the ranking service (30%).

3.5 XFinder

XFinder is a large service performing ads aggregation and ranking. Upon receiving a user request, it fans out requests to many leaf services, aggregates, and ranks the results before returning them to the user. XFinder uses record-and-replay, but to obtain accurate results, it requires near real-time traffic recorded from production within the past hour. Each week, it conducts over 3,000 and 1,000 experiments on code and configuration changes, respectively. To understand the impact of a change more precisely, instead of A/B tests, it runs 3-sided experiments: (1) the version currently running in production; (2) the latest version before this change; and (3) the new version with the change to be tested.

XFinder uses 65 key metrics for regression detection. These key metrics all use a regression threshold of 0.5%. The key metrics include total CPU time, log error or warning counts, count of ads returned, number of calls to downstream services, and failure rates of these calls.

3.6 Ranker

Ranker executes a graph of rules for ranking. A diverse set of application clients calls Ranker with different rules to provide ranking for their specific purposes, and these rules impact Ranker's performance. Ranker relies on record-and-replay to capture these rules. Requests from each application client are sampled on the client side and stored in the data warehouse. Each major client corresponds to a different shard of Ranker deployment, and these different shards run the same Ranker executable but serve different clients. Previously, Ranker created a mix of requests when replaying them for testing in ServiceLab. However, maintaining the correct ratio of requests in the mix became a burdensome process, and an incorrect ratio would lead to missed regressions. As a result, Ranker now runs separate experiments to replay traffic from different major clients.

Ranker uses 30 key metrics for regression detection. The key metrics and thresholds include container-level CPU usage (9%), container-level memory usage (7%), CPU MIPS busy (5%), and application metrics such as different types of candidates fetched (20%).

3.7 Applications not Using ServiceLab

ServiceLab tests now cover more than half of the total code changes at Meta. The remaining applications choose other testing methodologies for various reasons as described below. A common theme among them is that setting up a service for testing in ServiceLab requires effort, and sometimes a simpler alternative exists.

First, Meta's continuous deployment tool, Conveyor [22], and its in-production detection tool, FBDetect, can catch performance regressions either during the staged deployment process or during steady-state execution in production. Despite the higher risk of catching issues in production, these tools work sufficiently well for some services, leading those services to skip pre-production testing in ServiceLab.

Second, some services have complex interdependencies, and services like Meta's cluster manager ecosystem [42,54] even depend on the physical data center environment. These complex services have their own sophisticated ways of setting up their testing environments, which are often overly complicated to migrate to ServiceLab.

Third, some stateful services require a massive amount of data for effective testing. It is too slow to populate such data in newly allocated containers during each ServiceLab test run. Therefore, these services maintain their own dedicated and persistent test environments with prepopulated data, without relying on ServiceLab.

Fourth, some services do not consume significant capacity and do not have stringent performance requirements. As a result, thorough performance testing is not a priority for them. Their developers often prefer simpler ad-hoc testing methods, as opposed to the burden of setting up and maintaining their service setup in ServiceLab.

Finally, in a large organization with tens of thousands of developers, our experience indicates that achieving universal adoption of a technology is challenging unless it becomes a company priority, as demonstrated by the Push4Push program driving the universal adoption of the continuous deployment tool at Meta [22]. So far, ServiceLab has relied entirely on organic, bottom-up adoption without top-down push.

4 Taming Performance Variance

A key challenge in designing any testing platform is managing variance in testing data to separate signals from noises. To set the stage for the discussion, we first define some terminology. Assessing a code change's performance impact uses an *A/B test* to compare two *test runs*, one with the change and one without. A *trial* is a singular A/B test, and an *experiment* comprises multiple such trials. An *A/A* test compares two runs of the same code.

Performance differences may stem from (a) *accidental variance* caused by code's random factors such as the timing of lock contention; (b) *environment variance*, stemming from testing environment differences like CPU generation and kernel version; and (c) *true regression* in the code change. Our goal is to minimize the impact of accidental and environment variance to identify true regression.

To detect true regressions as small as 0.01%, we must aggressively reduce both accidental and environmental variance, as they could conceal small regressions. To reduce accidental variance, we collect a large amount of test data and then apply statistical analysis. To reduce environmental variance, we always acquire entire machines from our private cloud to run tests, avoiding the "noisy neighbor" problem. However, sequentially executing all test runs on one machine, while minimizing environmental variance, leads to prolonged test times and a slowdown in the iteration speed of software development.

One fundamental decision we have made is to run tests concurrently on different machines to expedite testing. Initially, the ServiceLab team operated its own dedicated machine pool and meticulously configured the machines to be nearly identical to reduce environment variance across machines. However, as the pool size expanded, maintaining it became uneconomical, leading us to switch to using our private cloud's shared machine pool. Moreover, the cloud allows ServiceLab to use temporarily reclaimed resources called "Elastic Servers", akin to Spot Instances in AWS, for testing. Since Elastic Servers can be revoked, our cloud employs predictive models to infer the availability of Elastic Servers and run tests correspondingly. When Elastic Servers are revoked unexpectedly, ServiceLab simply re-runs the interrupted tests.

When acquiring machines from the cloud, ServiceLab can specify a certain coarse-grained configuration such as CPU cores and memory, but cannot control other details, such as memory chip or kernel version. Note that the cloud automatically updates kernels at its own schedule to ensure security compliance. ServiceLab can provision a batch of machines, retain a subset of "*nearly identical machines*" to run test workloads, and return the rest. We do not require machines to be identical in every aspect as finding a sufficient number of such machines is difficult. Next, we discuss how to select "*nearly identical machines*" by using factors that impact a machine's performance most.

4.1 Machine Factors Impacting Performance

We analyze millions of test records to identify key factors impacting a machine's performance. Our analysis involves two large datasets. The Release to Production (RTP) dataset comprises 21.5 million records, each executing a CPU or memory benchmark. The ServiceLab dataset contains 186K records, each testing a real production application. Each record in both datasets specifies the test result alongside the used hardware and software configuration. Leveraging both datasets is crucial as they complement each other. The RTP dataset provides diverse hardware results, though its benchmarks are less complex. Conversely, the ServiceLab dataset contains real application results, but the machines used are less diverse.

We use the ANOVA method [52] to identify factors that best explain the variance in the data. ANOVA is similar to linear regression but operates on categorical data. Its output, the coefficient of determination (R^2), represents the proportion of the variance in the dependent variable (performance metrics in our case) that is predictable from the independent variables (e.g., CPU generation or kernel version). Our goal is to find a minimal subset of key machine factors (independent variables) that can explain as much variance as using many factors. This allows us to use these key factors for machine selection. Otherwise, a large number of factors would make it hard to find matching machines due to overly aggressive filtering. To achieve our goal, we first use many factors to establish an approximate upper bound for R^2, and then explore different subsets of factors to approach the upper limit.

To set the stage for discussion, we will first describe how physical machines are classified with three levels of granularity. At the most coarse level, machines are classified into tens of *ServerTypes*. Examples of ServerTypes include single-CPU general-purpose machine, two-CPU general-purpose machine, GPU training machine, GPU inference machine, etc. The median granularity, known as *ServerSubType*, takes into account more hardware information, such as RAM size and CPU architecture (e.g., Intel Skylake, Cooper Lake, etc.). The finest granularity, referred to as *ServerModel*, includes the model names of all major components, such as CPU, RAM, NIC, disk, etc. Typically, users specify ServerType when requesting machines from our private cloud. While specifying ServerSubType is allowed, it is discouraged because it limits flexibility for the cloud to choose machines. Users are not allowed to specify specific ServerModel. Concretely, our private cloud uses O(10) *ServerTypes*, O(100) *ServerSubTypes*, and more than 10,000 *ServerModels*. They are equipped with O(100) CPU models, O(100) RAM models, O(1000) disk models, and O(100) NIC models.

To approximate the upper bound of R^2, we use ServerModel as one factor since it includes almost all hardware information and add non-hardware factors like the kernel release version. We first report our results on the RTP dataset. These factors can achieve an R^2 of 0.89 for the CPU benchmark and an R^2 of 0.97 for the memory benchmark. In the remainder of this section, we will focus on the CPU benchmark as the memory benchmark exhibits much less variance.

We explore various factor subsets to determine if a small combination can achieve an R^2 close to the upper limit. Using three factors—ServerType, CPU architecture, and kernel release—we attain an R^2 of 0.87 on the CPU benchmark, closely approaching the upper bound. In practice, we observe that the cloud can generally provide matching machines based on these three factors. Note that this subset is not the only viable option. As hardware factors are correlated, some factors can be replaced by others. Additionally, we find that certain factors, such as RAM speed and RAM vendor, have minimal impact, even in memory benchmarks.

Analyzing the ServiceLab dataset reveals two additional important factors: CPU turbo and the datacenter region where the test was executed. Their impact varies across applications, and adding these factors can increase R^2 by up to 0.23. In comparison, in the RTP dataset, adding these factors only increases R^2 by 0.01 for the CPU benchmark. The influence of CPU turbo, previously reported in research [40], manifests only in the ServiceLab dataset, not in the RTP dataset. This difference arises due to constant CPU activity in the RTP benchmarks. The datacenter region is significant for real applications tested in the ServiceLab dataset because many of them have external dependencies. For example, if the test instance of an application reads from a production database in the region, the test result would be affected by the database's performance, which tends to vary across regions. In contrast, the RTP benchmarks have no external dependencies.

While the key factors account for 87% of the variance, the remaining 13% is attributed to other smaller factors. For example, in machines with CPUs of the same model, the frequency of their uncore components, such as cache and memory controller, can vary, resulting in approximately a 2% performance difference across tests. However, these factors cannot be used for selecting machines from the cloud as they are not exposed by the cloud.

Our analysis further reveals that certain CPU models and kernel versions contribute significantly more variance than others. Like prior works [40], we use Coefficient of Variance (CoV), defined as the ratio of the standard deviation to the mean, to compute the variance of a set of values. Specifically, regarding CPU models, Intel Xeon E5-2680 v4 @ 2.40GHz has the highest CoV at 42%, while AMD EPYC 7D13 36-Core Processor has the lowest CoV at 5.6%, with an overall P50 at 19%. Regarding the kernel, version 5.6.13-0 has the highest CoV at 52%, and 5.2.0-240 has the lowest CoV at 9.5%, with the overall P50 at 36%. While investigating the root cause of CoV is beyond the scope of this paper, ServiceLab avoids using CPUs or kernel versions with high variance.

In summary, the strategy we use is to select similar machines with matching kernel versions, ServerTypes, CPU architecture, and datacenter regions, while disabling CPU turbo. To assess whether performance variance within the similar machines selected by our criteria is comparable to that for a single machine, we compare their CoVs. The comparison is conducted using the RTP dataset with turbo disabled. The CoVs for same-machine tests are 5.9% at P50 (50 percentile) and 28% at P99, while for similar-machine tests, they are

5.7% at P50 and 38% at P99. The P50 values are nearly identical, with a higher difference at P99. Overall, the difference is deemed acceptable, considering the advantage of running tests in parallel.

Applicability to public cloud. To implement our machine selection method in a public cloud, we recommend using bare metal instances, which are offered by all major cloud providers, rather than the more commonly used virtual machine instances. Although it is reported that lightweight hypervisors like AWS Nitro System can match the performance of bare metal machines [34], they may still introduce greater performance variability than bare metal machines. We have not validated our method in virtualized environments.

4.2 Statistical Methods

Despite using matching machines, experiments still exhibit variance. We conduct experiments multiple times and employ statistical methods to determine the level of regression with a confidence interval. In general, we observe that there is no one-size-fits-all model due to the diverse requirements and varying performance-data distribution of different services. Therefore, ServiceLab incorporates multiple models with a mechanism to learn the best model for each service based on historical data.

Recall that a *trial* is a singular A/B test, and an *experiment* comprises multiple trials. An *A/A* test compares two runs of the same code. A test may generate multiple data points. For example, a test may measure CPU utilization for an hour and generate a CPU-utilization data point per minute.

A model used by ServiceLab is a combination of a statistical test method and a data preprocessing method. ServiceLab uses the following statistical test methods:

- **Student's t-test** [17]. If an experiment only contains a single trial, we use the student's t-test to determine whether there is a significant difference between the means of the A side and the B side.

- **Permutation test** [6]. If an experiment includes multiple trials, for each trial, we first compute the difference in means between the A side and the B side. This step results in a vector of m values called $\vec{M}$, where m is the number of trials. Then we posit the null hypothesis H_0: $\mu_\Delta = 0$, where μ_Δ is the mean of $\vec{M}$. We apply a permutation test for this hypothesis as follows. We generate a large number of permuted samples from $\vec{M}$ and calculate the mean for each. Then we derive the p-value from the proportion of permuted sample means that are as extreme as or more extreme than the observed mean of $\vec{M}$.

- **Confidence interval test**. The above tests infer the distribution of the data from the experimental data. Since we can only run a limited number of trials within an experiment and some tests may incur outliers, such inference may not be accurate. The confidence interval test builds the data distribution from historical data. Specifically, it leverages A/A tests from the past two weeks to build the distribution of $mean(A') - mean(A)$, and further computes the confidence interval given the p-value, i.e., the probability of the observed difference of means being smaller than the confidence interval is larger than $1 - p$. Then, for an experiment, it can test whether the B side follows the same distribution as the A side by determining whether $mean(B) - mean(A)$ is smaller than the confidence interval.

ServiceLab uses the following data preprocessing methods:

- **Square root transformations.** An important preprocessing step involves square root transformations. This is motivated by recognizing significant heterogeneity in the cost of requests, with certain requests disproportionately impacting mean metric values. Such disparities are exacerbated across multiple trials, leading to skewed aggregations. The square root transformation mitigates this, ensuring a more uniform contribution from each request to the trial's mean metric value. This adjustment has been empirically validated to enhance detection accuracy, especially for high-demand services.

- **Outlier detection.** We use conventional outlier mitigation methods, such as winsorization [18], which are particularly effective in moderating the elevated variance observed when services operate under strenuous conditions. Specifically, we either delete data points that are above a certain percentile (called outlier-elim) or cap those data points at the percentile value (called outlier-cap).

Out of all possible combinations of statistical test methods and data preprocessing methods, currently ServiceLab uses seven combinations: t-test-none, t-test-sqrt, t-test-outlier-elim, t-test-outlier-cap, permutation-test-none, permutation-test-sqrt, and confidence-interval-none, as well as some service-specific models.

ServiceLab uses an adaptive method to determine the best model for each <*service, metric*> combination. It conducts periodic A/A experiments and artificial A/B experiments (i.e., A/A experiments with injected regression on one side) to generate a "ground truth." Then, ServiceLab tests each model on the results of these experiments to obtain the model's false positive rate (from the A/A experiments), the false negative rate (from the artificial A/B experiments), and the detectability [10] (from the A/A experiments). ServiceLab then selects the model with the highest score, which is a linear combination of the false positive rate, false negative rate, and detectability, under the constraint that its false positive rate is below a threshold. ServiceLab runs this model selection algorithm periodically to adapt to changes in existing services and accommodate new services and metrics.

In our production, 51% of the services have adopted the confidence-interval-none model, 21% have adopted the t-test-sqrt model, 21% have adopted the adaptive method, 5% have adopted the t-test-none model, and 1% have adopted the permutation-test-none model. Not all services use the adaptive method, either because they find a fixed model always works well or because they have not tried the recently introduced adaptive method.

The breakdown of the models chosen by the adaptive method is as follows: confidence-interval-none (49%), t-test-none (19%), t-test-outlier-cap (10%), t-test-outlier-elim (9%), permutation-test-none model (5%), t-test-sqrt (4%), and permutation-test-sqrt (3%). Over a 90-day period, for services using the adaptive method, more than 50% of them have changed models at least once, and more than 10% of them have changed models at least four times. This indicates that the best model for a service can change as the service's code and characteristics evolve.

Next, we discuss our journey to arrive at the current set of models. Initially, ServiceLab only supported single-trial experiments and thus used the student t-test with outliers handled by winsorization. When working with services requiring multiple trials, we found that the t-test did not work well. Outliers in both trials and requests within a trial affected the experiment results, showing up as either false positives or missed regressions due to excluding outliers. Consequently, for multiple-trial experiments, we added the permutation-test-none and permutation-test-sqrt models. The confidence-interval test was added to handle noisy metrics or ones that are not continuous like CPU or memory. We found that for these metrics, looking at the historical data to find the regression threshold would work better than dealing with a t-distribution (as the t-test and other variants do). Finally, motivated by the observation that different metrics follow different distribution patterns and such patterns may change over time, we added the adaptive method to help users find the best model.

Finally, we describe two optimizations that improve the accuracy of the statistical methods.

Test warm up time. Identifying and excluding initial warm-up periods in service operations is crucial for isolating steady-state performance metrics. We employ an algorithm using exponential moving averages to determine the point at which a time series reaches approximate stationarity. Observations before this point are discarded. As the duration of the warm-up phase depends on the test environment, this determination is made on an individual trial basis, ensuring that only matured performance data undergoes further analysis.

Periodic A/A experiments. ServiceLab conducts periodic A/A tests, and aggregates the results into a user dashboard, enabling users to monitor the false positive rate and detectability. This dashboard aids users in modifying workload settings to enhance the statistical signal of their experiment. For instance, users can adjust parameters such as increasing the number of trials, extending experiment duration, removing noisy metrics, or changing the aggregation method.

5 ServiceLab Design

This section presents the design of ServiceLab, utilizing the architecture diagram shown in Figure 2 in our discussion.

5.1 Experiment Lifecycle

During an experiment's lifecycle, it transitions through several phases: *queued→build→allocation→running→analysis*. An experiment begins when a user or an automation tool submits a request via the Windtunnel API, which enqueues the request into a *DurableQ* (durable queue) and creates an entry in the Windtunnel DB to represent the experiment, setting its phase as *queued*. The phase transition of an experiment is managed by a *processor*, and multiple processors can work independently to manage different experiments. When a processor determines it can take on additional work, it polls the DurableQ to claim a queued experiment and locks the corresponding Windtunnel DB entry to prevent other processors from performing duplicate work.

After some input validation and preprocessing, the processor transitions the experiment to the *build* phase, where the experiment's executables are created. The processor does not compile the executables directly but instead sends a request to a separate build service, which acts as a caching layer to prevent duplicate builds.

Once all executables are built, the experiment enters the *allocation* phase. Each team is configured with a certain testing-machine quota that they are allowed to use. The processor tracks the already used portion of the quota and determines when to allocate machines for experiments, enforcing priority and fairness. An experiment may need to run multiple jobs, such as one for the A side of the A/B test and another for the B side. As all jobs of an experiment must be allocated from the same datacenter region to minimize variance (§4.1), the processor decides from which region to allocate the jobs based on the remaining quotas in different regions. Additionally, the processor filters machines based on ServerType, CPU architecture, and kernel version to minimize variance across the selected machines (§4.1).

In addition to allocating the system-under-test (SUT) jobs, the processor also allocates a traffic-replay job and a test-harness job. The test-harness job drives the experiment and monitors the test's status. Depending on the experiment's purpose, different test-harness jobs can be used. For example, to measure the maximum throughput that the SUT can sustain, the *capacity* test harness can gradually increase the test throughput until the SUT violates its SLOs, such as response time, error rate, or CPU utilization exceeding a threshold.

Once all the necessary jobs are allocated, the experiment enters the *running* phase. If the experiment is testing a configuration change, the corresponding configuration canary [13,15,53] is set up correctly on the test machines. Subsequently, the traffic-replay job loads the previously recorded requests that will be replayed during the experiment. Finally, the processor instructs the test harness to start the test. Through-

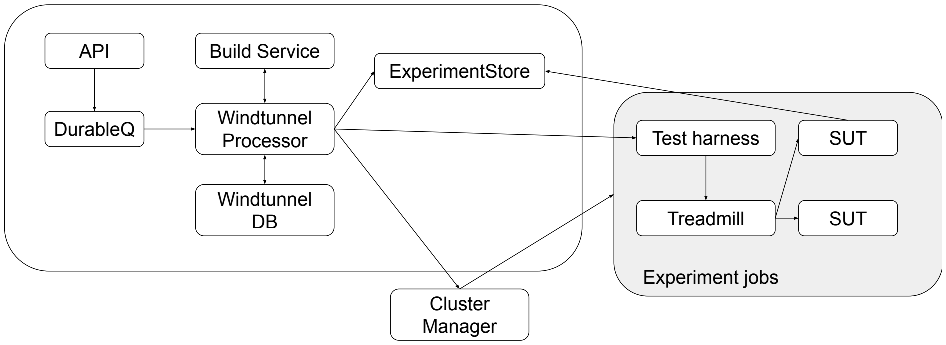


Figure 2: ServiceLab architecture. *Windtunnel* is the orchestration engine and *Treadmill* replays traffic and runs tests.

out the experiment, the test harness monitors the health checks of all jobs and fails the experiment if any job fails its health check. Meanwhile, the SUT exports performance metrics to monitoring databases during the experiment.

After the test finishes, the processor deallocates all jobs and transitions the experiment to the *analysis* phase. Aggregation and statistical analysis of performance metrics are performed by a service called *Experiment Store* (ES). The results are written to the Windtunnel DB, which users can view from a UI. These results can also trigger certain actions, such as blocking a service from being released into production.

5.2 Traffic Record-and-Replay

At Meta, all services use the Thrift [49] RPC protocol, which is leveraged by ServiceLab to record production traffic transparently. The user specifies the Linux containers where RPC traffic should be recorded. Upon receiving a request on these containers, the Thrift server's recording module flips a coin to decide whether to sample the request. To minimize the impact on RPC latency, the recorded data is written asynchronously. The user configures the request sampling rate, and by default, requests are sampled uniformly. For a service with a highly variable request rate, reservoir sampling [5, 57] ensures that sampled requests are evenly spread over time, with at most K samples during each T time interval. In practice, the median sampling rate is 0.03%, and above that, 22% of services set sampling rate to 1%.

By default, ServiceLab assumes that RPC requests are independent—meaning the execution of one request does not depend on the execution of the previous request. However, this assumption may not hold for some services. In these cases, the service can record all requests, and then during replay, all recorded requests are replayed in order. For some services, request dependencies are encoded at the RPC layer and can thus be recorded accurately and transparently.

For sharded services, since sharding is done at the application layer and is invisible to the RPC layer, ServiceLab relies on the service owner to specify the set of Linux containers to capture requests for all shards.

ServiceLab leverages the open-source load testing platform Treadmill [61] to replay requests. Treadmill employs an open-control loop to send requests at a fixed rate. We have implemented several modifications to Treadmill, extending it to load recorded requests from a datastore and replay Thrift RPC requests. In support of A/B experiments, we further enhanced Treadmill to ensure consistent pacing and request rates for both sides. A single Treadmill instance loads an identical set of requests to be dispatched to both SUTs, synchronizing the sending of requests to ensure simultaneous receipt on both sides. For the capacity mode, a control loop in the test harness monitors the SUT and instructs Treadmill to dynamically adjust the request rate.

Additionally, services may have a warm-up period during which performance measurements should not be taken until the service reaches a steady-state behavior. ServiceLab can be configured to enable a *warm-up phase*, allowing a lower request rate to be set during this phase to gradually increase the load on the service. The service exports a counter to indicate whether it has reached warm-up. Treadmill waits for both sides to be warmed up before synchronizing and sending requests once again for steady-state performance measurement. For example, HHVM employs a JIT compiler, and steady-state performance measurements should start after JIT compilation is sufficiently warmed up.

Finally, services with high variability in request processing time due to diverse request types are harder to handle. FrontFaaS is one such example. We can reduce variability by testing only with request types relevant to a specific code change, as opposed to all request types.

5.3 Handling Service Dependencies

Meta products are built out of tens of thousands of services with intricate interdependencies, akin to those documented in prior research [30, 38, 48]. For example, FrontFaaS invokes hundreds of downstream services. Consequently, testing a service in isolation is challenging due to these interdependencies. ServiceLab tackles this issue through various approaches.

First, users can set up a group of interdependent services together in ServiceLab, creating a self-contained testing environment. While theoretically possible, this approach is not consistently implemented in practice due to various reasons. For instance, replicating the massive datasets accessed by services, like the social graph for billions of users, is often economically impractical.

Second, ServiceLab allows a system under test (SUT) to invoke certain services in the production environment, provided there are no adverse side effects. In ServiceLab, most calls to downstream production services do not incur side effects because they are read-only or idempotent. Moreover, the testing load imposed on these downstream services is often negligible compared to their ample capacity to serve billions of users. However, as these downstream services often exhibit performance variance across datacenter regions, meaningful comparisons can only be drawn from tests conducted in the same region (§4.1).

Third, for a SUT that potentially can cause side effects on downstream production services, ServiceLab requires the service owner to modify the SUT's behavior to prevent those side effects. For example, the SUT may use a mock interface of a database so that it writes data to a test database instead of the production database. Moreover, to prevent a SUT from accidentally accessing a production service, the RPC layer can be instructed to block all traffic to production services except those on an allowed list. Mocking or blocking traffic can result in certain code paths not being executed, potentially causing false negatives in testing results. However, § 6.1 shows that the false negative rate of ServiceLab is acceptable.

Fourth, the business and performance metrics logged by the SUT are kept separately from those generated by its counterpart in production. This ensures that the analytics for these metrics do not interfere with each other.

In summary, ServiceLab provides tools to assist service owners in managing service dependencies during test environment setup but does not offer complete isolation out of the box. As a result, some complex services (e.g., MySQL) are not tested in ServiceLab. They either use a specialized test environment or conduct canary tests directly in production by deploying new code to some instances of the production service and comparing those instances with the rest. Despite its limitations, ServiceLab is successful as a general-purpose testing platform, covering more than half of the total code changes by all services and surpassing the combined coverage of all other specialized testing platforms.

6 Production Experience

During its steady state, ServiceLab constantly leverages tens of thousands of machines to test hundreds of services and hundreds of ML models. We use production data to answer the following questions:

1. What are the statistics for different use cases (e.g., regression thresholds, number of trials, etc.)?
2. What are the false positive and false negative rates of ServiceLab?
3. How much regression did actually ServiceLab prevent?

6.1 Testing FrontFaaS

As FrontFaaS is our largest programming platform and has more code changes than other services, we report its statistics separately. ServiceLab has been running for FrontFaaS for over 5 years in production. It has a regression threshold as low as 0.01%, and by default, it runs 25 trials in each experiment. On average, developers made over 100,000 FrontFaaS changes per month. ServiceLab ran at least one experiment on 23% of those changes during that period. Leaving out 77% showcases the importance of ServiceLab's DiffSuggester in reducing the machine capacity needed for testing. For the code changes tested by ServiceLab, ServiceLab signaled performance regressions on 0.3% (5,560) of those changes.

ServiceLab assigns regression tickets to developers, and we calculate ServiceLab's accuracy based on the developers' actions in these tickets. We classify a signaled regression as a true positive if the developer fixed the issue or marked the issue as "*expected*," perhaps due to a new product feature requiring more resources. We classify it as a false positive if the developer identified it as such. If the developer did not provide a clear answer, we classify the regression as unknown. Among all signaled regressions, 57% (3,173) are true positives, 15% (823) are false positives, and 28% (1,564) are unknown. Assuming the unknowns have the same false positive rate as others, the overall false positive rate is about 21%.

Although the false positive rate of 21% may seem high initially, it actually signifies a significant success of ServiceLab because FrontFaaS uses a very low regression threshold, 0.01%. Out of all FrontFaaS diffs submitted by developers, only 0.014% experience a false positive flagged by ServiceLab, calculated as $23\% \times 0.3\% \times 21\% = 0.014\%$. Assuming a developer writes one diff per day, they will experience a false positive about once every 20 years! While promoting the adoption of ServiceLab, we learned that the per-developer experience significantly affects whether developers ignore the regression tickets assigned by ServiceLab. If a developer frequently receives false-positive tickets, they are likely to ignore them after repeated futile investigations. Conversely, if they receive a false-positive ticket only once every 20 years, they will likely always take ServiceLab regression tickets seriously and investigate them. The good developer experience even at a very low regression threshold of 0.01% demonstrates the robustness of ServiceLab's statistical methods.

Figure 3 shows the distribution of the level of regression of those true positive cases. The median value (p50) is 0.14%, p90 is 1.7%, and p99 is 38.7%. Summing them together, they account for 12284% of regression over five years. Since very large regressions are often caused by experimental purposes, if we only sum those causing less than 1% regression, they account for 545% of regression, which translates to over 2 million machines (i.e., 545% × the number of machines used by FrontFaaS). This shows that ignoring small regressions is not acceptable, as they will accumulate to a large number over time. That is why FrontFaaS uses a strict threshold.

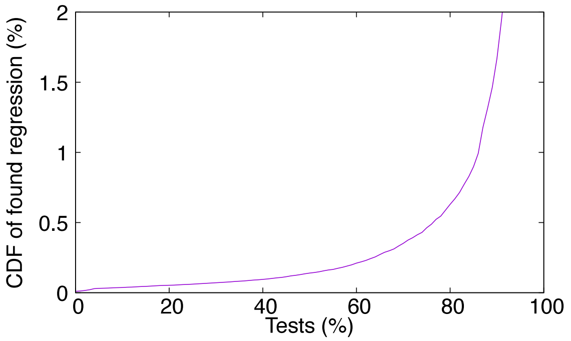


Figure 3: Cumulative distribution of regressions detected by ServiceLab for FrontFaaS.

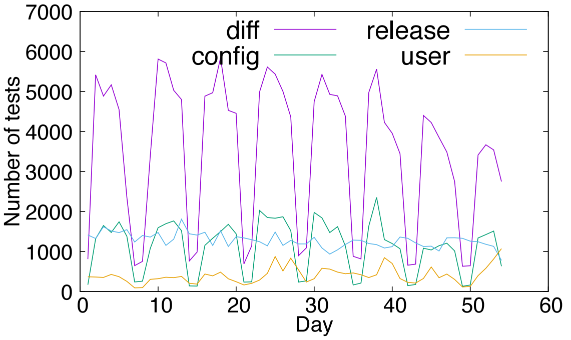


Figure 4: Number of experiments completed each day.

To approximate the false negative rate, we rely on reports from in-production monitoring of performance changes. FrontFaaS has a production monitoring system that examines the per-function CPU usage of functions and raises a signal if it detects a performance regression. It then attempts to triage the performance regression to a code change. Out of all changes, 0.02% (2038) of changes were found to cause regressions but have been missed by ServiceLab, leading to a false negative rate of 32%, calculated as $\frac{2038}{2038+5560\times(1-21\%)}$. However, this number should be viewed with caveats because 1) there are potential performance regressions that cannot be root-caused to their original changes, which are not included in this number, and 2) there is no guarantee that the production monitoring system is fully accurate.

In summary, despite FrontFaaS' low threshold of 0.01%, ServiceLab achieves a reasonable false positive rate and false negative rate, and helps us prevent a significant amount of regressions, which could accumulate over years.

6.2 Testing Other Services

While FrontFaaS is reported in its own category, in this section, we report the aggregate statistics for all non-FrontFaaS services in one category. Figure 4 shows the number of ServiceLab experiments completed each day for non-FrontFaaS services. The majority of completed experiments are run automatically as part of code changes, configuration configs, or service releases. ServiceLab supports a total of 483 distinct use cases, and their breakdown is shown below. Note that ServiceLab also tests hundreds of distinct ML models, which are counted as a single use case.

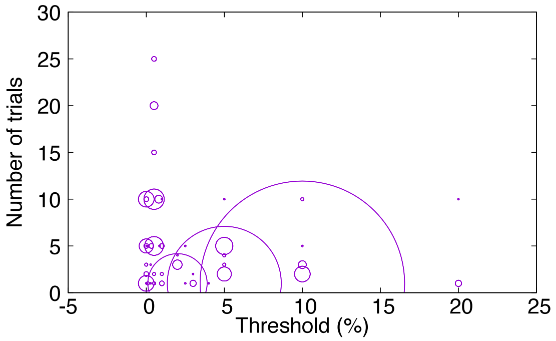


Figure 5: Thresholds and number of trials. The size of the circle represents the count of use cases with the same setting.

- 44% (N=211) of the use cases have experiments that automatically run on code diffs.
- 15% (N=74) run automatically on code commits.
- 21% (N=100) run automatically on configuration changes.
- 22% (N=107) run as part of their release process.

The distribution of the number of trials in experiments is as follows: p50=1, p90=10, p99=10, and p100=25. The distribution of the execution time of trials is as follows: p50= 2,820 seconds, p90= 4,200 seconds, p99=p100=259,200 seconds. Among the 483 use cases, 413 have defined a relative threshold on some metric; 5 have defined an absolute threshold on some metric; the remaining ones do not define any threshold. We focus on the 413 cases with a relative threshold in the following discussion.

Each use case may contain multiple metrics with different thresholds. Since the number of trials and trial duration are usually determined by the strictest threshold, we define the threshold of a use case as the smallest threshold among all its metrics. 23% of the use cases have a threshold smaller than 1%, while p50=5%, p90=10%, and p99=20%. This, once again, emphasizes the importance of using small thresholds.

Figure 5 plots the threshold and the number of trials used by different use cases. A circle in this figure represents the count of use cases using a specific setting. This figure shows that a large number of use cases use a relaxed threshold of 5% or 10% with only one trial, but a small number of use cases use a very small threshold with up to 25 trials. This small subset includes many of the largest services.

Services often run preliminary experiments with a large number of trials to determine how many trials are needed to achieve a certain confidence interval in regression detection. Specifically, they run multiple trials of A/A tests, compute the difference between each pair of A/A test (i.e., $\frac{A'-A}{A}$), and then determine the confidence interval (i.e., 95% within two standard deviations assuming normal distribution). Figure 6

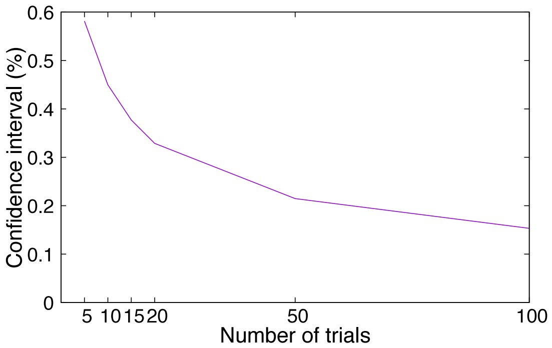


Figure 6: The number of trials required for detecting small regressions.

shows, for one service, how the confidence interval decreases with more trials. Users can then decide the number of trials according to their required confidence interval.

We examine how often ServiceLab signals a regression on code changes during the 54 day period shown in Figure 4. During this time, 15,058 code changes were tested and ServiceLab signaled on 2,742 (18.5%) of those code changes with at least one metric crossing its configured threshold. For non-FrontFaaS services, ServiceLab reports on a diverse set of metrics. Across these code changes, 2,714 different metrics were considered as significant. 80% of the signaled metrics had a threshold of less than 2%. Unlike the uniform FrontFaaS platform used by over ten thousand developers, for these 413 diverse use cases, there are no uniform tools and hence no clear marking about whether a reported regression is a true or false positive. In subsequent sections, we will present some examples with these use cases to understand their impact.

6.2.1 Example of True Positives

ServiceLab helps non-performance experts understand the performance implications of their code. For example, consider one case where ServiceLab successfully detected and prevented a CPU performance regression in XFinder (§3.5) before the change landed in production.

In the change, the developer inadvertently copied a large data structure when introducing a new function. The ServiceLab experiment that ran for this change detected a significant CPU regression of around 20%. ServiceLab flagged this change to both the developer and performance engineers working on XFinder. The developer was working on a product feature across multiple services, and was neither familiar with the XFinder codebase nor C++. After ServiceLab flagged the regression, the developer applied a fix by adding `const` when passing the parameter to the function, eliminating the memory copy of the data structure.

In another incident involving the Ranker service (§3.6), a change increased the service's memory usage by 50%. The change involved enabling a new ranking library that increased memory usage due to loading additional ranking configurations. The increase in memory was expected due to the additional functionality; however, the amount of increased memory was not. ServiceLab detected the memory regression before a release deployment. In this case, the developer who included the additional ranking library knew that there would be an added resource cost. However, ServiceLab helped the developer and service owners understand the resource cost of the regression before deployment. The developer reverted the change and found optimizations to minimize the use of the ranking library by excluding unused ranking configurations.

6.2.2 Example of False Positives

In another incident, a ranking service using ServiceLab occasionally experienced high rates of false positives due to a production issue with a downstream dependency. A production misconfiguration led to imbalanced load among the machines in the downstream service. During experiments, some of the SUTs would send requests to these overloaded instances of the downstream service. The queuing resulting from those overloaded downstream instances affected the performance measurement in ServiceLab, resulting in false positives. To remediate this issue, the production routing configuration that led to the imbalanced load was fixed. This remediated the load imbalance issue in production and also eliminated the false positives in ServiceLab.

6.2.3 Examples of False Negatives

False negatives are incidents where ServiceLab does not report a regression but a regression actually occurs. These cases are often reported by service owners. In one incident with XFinder (§3.5), a developer was implementing a new feature to read from an online classifier instead of an offline classifier. The change introduced a new function call making use of the new classifier to better classify the type of ads to return. The change resulted in an increase of 0.62% in the total capacity used by XFinder. ServiceLab failed to report a regression for this change since this regression only applied to a subset of request types, and those types were not represented in the set of requests replayed in the experiment. Those request types were newly added after the request trace was captured.

6.2.4 Summary

In our experience, the top reason for false positives is that another event, such as another test or deployment either in the SUT or in the downstream services, is happening concurrently with a ServiceLab test, which will disrupt the result of the ServiceLab test. The top reason for false negatives is that a newly introduced feature is not tested since the requests for replaying were recorded when this feature does not exist.

6.3 False Positive in A/A Experiments

As described in § 4.2, periodic A/A experiments provide an empirical measurement of whether a metric would be considered significant with the same experiment inputs. Periodic A/A experiments run every two hours and test for statistically significant differences without considering any signaling thresholds. Over a two-month period, we examined 6,783 metrics from A/A experiments where signaling was enabled. Among these 6,783 metrics, the p50, p90, and p99 metric had a false positive rate of 0.6%, 40%, and 64%, respectively. This signifies the inherent variance in the services and the test environment. It also emphasizes the importance of our method of using results from A/A experiments to help select the best statistical model for each service (§4.2).

6.4 Key Takeaways

We have learned several key lessons from our experience of operating ServiceLab over seven years. Initially, the ServiceLab team maintained a dedicated pool of identical physical machines for testing to reduce performance variance. However, as ServiceLab adoption increased, we had to switch to using heterogeneous cloud machines. This change was driven by the high maintenance burden of a dedicated machine pool and the lower cost of running some tests using the cloud's elastic capacity.

Testing a wide range of services is a key design goal for ServiceLab. To achieve this, unlike traditional systems that aim for completely isolated and reproducible environments, ServiceLab allows Systems Under Test to call external dependent services that handle live production traffic. This approach significantly broadens the scope of services that ServiceLab can test, accommodating those with complex interdependencies that are too intricate or costly to replicate fully in a test environment. Moreover, ServiceLab is extensible and allows for developer customizations, recognizing that a one-size-fits-all approach would fall short in supporting diverse services. For example, while traffic record-and-replay simplifies test setup, some services face strict time constraints for replay, and others choose to use synthetic traffic.

7 Related Work

Performance Variance. Performance variance is a well-known issue for performance experiments and reproducibility, especially when the performance of two versions to compare is close. There are multiple lines of work in this direction: 1) some works mitigate the problems by re-designing systems [9, 16, 21, 25, 45, 51, 60], tuning configuration parameters [37, 39, 59, 64], or changing hardware [40, 58, 63]. 2) Some works try to detect machines that are significantly slower than others [19, 23, 28, 29, 43], so as to exclude such outlier machines from performance experiments. We also run routine performance tests to filter those outlier machines. 3) Some works propose statistical methods for performance comparison [26, 31, 40].

The closest work is the study by Maricq et al. on performance variance in CloudLab [40]. ServiceLab differs from the CloudLab study in several ways. First, the CloudLab study assumes repeated experiments are run on the same or identical machines, whereas ServiceLab identifies heterogeneous machines with comparable performance to run experiments in parallel. Second, the CloudLab study focuses on the number of experiments needed to achieve a certain confidence interval, whereas ServiceLab addresses the problem more holistically, using an ensemble of statistical models, A/A tests, and artificial A/B tests. Finally, the CloudLab study only runs microbenchmarks in a single-machine environment, whereas ServiceLab must be robust enough to work in real-world scenarios with full services and complex interdependencies.

Performance Testing. Synthetic benchmark [14, 41, 46, 50, 55] and record-and-replay [1–3] are two primary methods for performance evaluation. ServiceLab supports both but primarily uses record-and-replay due to its high fidelity in testing real applications.

Treadmill [61] and TailBench [32] overcome several common pitfalls of performance testing frameworks with synthetic traffic, allowing them to precisely measure at microsecond-scale. Lancet [33] incorporates online statistical tests to ensure the obtained measurements are statistically sound. Primorac et. al. leverage kernel-bypass networking and advanced NIC features to further improve the precision of microsecond-scale tail latency measurements [44].

Performance data from Google's gmail [7] shows that workloads change constantly, both QPS and response size. Hence we need to do real production traffic record and replay. Recent studies including Kraken [56] and WSMeter [35] directly utilize production traffic to carry out the performance tests, to address the limitation of synthetic benchmarking in how accurately they can reproduce the complex production environment. Similarly, deterministic record and replay are commonly leveraged to reduce the non-determinism to simplify multiprocessor software development and testing, which can be done at multiple levels (e.g., virtual machine-level [20], OS-level [8], and library-level [24]).

8 Conclusion

We have presented ServiceLab, which tests a large, diverse set of applications to catch small performance regressions. It selects similar but non-identical machines for testing and learns the best statistical model for each service. During seven years of production, ServiceLab has helped us prevent a significant amount of regression, which could accumulate over time if not detected promptly.

Acknowledgments

This paper presents over seven years of work by past and current members of several teams at Meta, including ServiceLab, HHVM, Ads, Release Engineering, and Instagram. In particular, we would like to thank those who have contributed to ServiceLab and this paper: Matt Meyers, Elena Mironova, Yun Jin, Harish Dattatraya Dixit, and Gautham Vunnam. We also thank all reviewers, especially our shepherd, Atul Adya, for their insightful comments.

References

[1] Azure Public Dataset. https://github.com/Azure/AzurePublicDataset.

[2] Google Cluster Workload Traces 2019. https://research.google/resources/datasets/google-cluster-workload-traces-2019/, 2019.

[3] Google Workload Traces 2022. https://research.google/resources/datasets/google-workload-traces-2022/, 2022.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 739–753, 2016.

[5] Charu C Aggarwal. On Biased Reservoir Sampling in the presence of Stream Evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618, 2006.

[6] Marti J Anderson and John Robinson. Permutation tests for linear models. *Australian & New Zealand Journal of Statistics*, 43(1):75–88, 2001.

[7] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, 2018.

[8] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic Process Groups in dOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[9] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, 2018. USENIX Association.

[10] Howard S Bloom. Minimum Detectable Effects: A Simple Way to Report the Statistical Power of Experimental Designs. *Evaluation review*, 19(5):547–556, 1995.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[12] Nicolas Brousse. The Issue of Monorepo and Polyrepo In Large Enterprises. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, pages 1–4, 2019.

[13] Amazon CloudWatch - Creating a canary. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries_Create.html.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[15] David Daly. Creating a Virtuous Cycle in Performance Testing at MongoDB. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 33–41, 2021.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[17] Wilfrid J Dixon and Frank J Massey Jr. Introduction to Statistical Analysis. 1957.

[18] Wilfrid J Dixon and Kareb K Yuen. Trimming and winsorization: A review. *Statistische Hefte*, 15(2-3):157–170, 1974.

[19] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[20] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[22] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 325–342. USENIX Association, 2023.

[23] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 1–14, Oakland, CA, February 2018. USENIX Association.

[24] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, volume 8, pages 193–208, 2008.

[25] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, USA, 1st edition, 2013.

[27] HHVM. https://hhvm.com.

[28] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018. USENIX Association.

[29] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, New York, NY, USA, May 2017. ACM.

[30] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX, 2023.

[31] Raj Jain. *The Art Of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling*. john wiley & sons, 2008.

[32] Harshad Kasture and Daniel Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[33] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, Renton, WA, July 2019. USENIX Association.

[34] Matt Koop. Bare metal performance with the AWS Nitro System. https://aws.amazon.com/blogs/hpc/bare-metal-performance-with-the-aws-nitro-system/.

[35] Jaewon Lee, Changkyu Kim, Kun Lin, Liqun Cheng, Rama Govindaraju, and Jangwoo Kim. WSMeter: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers. *SIGPLAN Not.*, 53(2):549–563, mar 2018.

[36] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, 2021.

[37] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[38] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[39] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.

[40] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[41] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2020.

[42] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[43] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, Renton, WA, July 2019. USENIX Association.

[44] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to Measure the Killer Microsecond. *SIGCOMM Comput. Commun. Rev.*, 47(5):61–66, oct 2017.

[45] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, 2018. USENIX Association.

[46] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[47] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[48] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[49] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper*, 5(8):127, 2007.

[50] Standard Performance Evaluation Corporation. https://www.spec.org/.

[51] Akshitha Sriraman and Thomas F. Wenisch. uTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, 2018. USENIX Association.

[52] Lars St and Svante Wold. Analysis of variance (ANOVA). *Chemometrics and intelligent laboratory systems*, 6(4):259–272, 1989.

[53] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[54] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat,

Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.

[55] Transaction Processing Performance Council. The TPC home page. http://www.tpc.org.

[56] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, 2016.

[57] Jeffrey S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, mar 1985.

[58] Guosai Wang, Lifei Zhang, and Wei Xu. What Can We Learn from Four Years of Data Center Hardware Failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[59] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and Auto-Adjusting Performance-Sensitive Configurations. *SIGPLAN Not.*, 53(2):154–168, March 2018.

[60] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, February 2017. USENIX Association.

[61] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 456–468. IEEE Press, 2016.

[62] Wei Zheng, Ricardo Bianchini, G John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *Proc. USENIX Annual technical conference*, volume 96, 2009.

[63] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, 2018. USENIX Association.

[64] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 338–350, New York, NY, USA, 2017. Association for Computing Machinery.

MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale

Arnab Choudhury[1], Yang Wang[1 †], Tuomas Pelkonen[1], Kutta Srinivasan[‡], Abha Jain[1],
Shenghao Lin[1], Delia David[1], Siavash Soleimanifard[1], Michael Chen[1], Abhishek Yadav[1],
Ritesh Tijoriwala[1], Denis Samoylov[1], and Chunqiang Tang[1]

[1] Meta Platforms
[†] The Ohio State University
[‡] LinkedIn (work done while at Meta)

Abstract

In public clouds, users must manually select a datacenter region to upload their ML training data and launch ML training workloads in the same region to ensure data and computation colocation. Unfortunately, isolated decisions by individual users can lead to a mismatch between workload demand and hardware supply across regions, hurting the cloud provider's hardware utilization and profitability. To address this problem in Meta's hyperscale private cloud, we provide a global-scheduling abstraction to all ML training workloads. Users simply submit their training workloads to MAST, our global scheduler, and rely on it to intelligently place both data and training workloads to different regions. We describe three design principles that enable MAST to schedule complex ML training workloads at a global scale: temporal decoupling, scope decoupling, and exhaustive search. MAST successfully balances the load across global regions. Before MAST, the most overloaded region had a GPU demand-to-supply ratio of 2.63 for high-priority workloads. With MAST, this ratio has been reduced to 0.98, effectively eliminating the overload.

1 Introduction

The success of ML applications [8, 46] has resulted in ML training becoming the fastest-growing datacenter workload. Public cloud providers run ML training workloads in multiple geo-distributed datacenter regions [3,4,15] to ensure sufficient capacity. Accordingly, users need to manually select a region to upload their ML training data and then launch training workloads in the same region to ensure colocation of data and computation. Unfortunately, such manual selection can lead to a regional mismatch between workload demand and hardware supply. For instance, one region may exhaust its capacity, accumulating a long queue of pending jobs, while another region has surplus capacity remaining idle.

Meta's private cloud used to experience this load imbalance. It comprised tens of datacenter regions, millions of machines, and tens of thousands of GPUs. Similar to public clouds, users initially had to manually select regions to store training data and launch workloads. Users' suboptimal decisions previously led to an imbalance in the GPU demand-to-supply ratio, reaching as high as 2.63 in certain regions for high-priority workloads, which was later reduced to 0.98 through optimizations described in this paper.

While much research has been conducted on scheduling ML workloads in a single cluster [1, 2, 5–7, 9, 13, 17, 21, 23–25, 30, 31, 33–35, 39, 40, 45, 49, 51–54, 57, 59], there has been little effort to address the issue of regional mismatch between workload demand and hardware supply. To address this challenge, our private cloud has introduced the *global-scheduling* abstraction that shields users from the complexity of regions. With the global-scheduling abstraction, users simply submit their ML training workloads to our global scheduler called MAST (short for ML Application Scheduler on Twine [44]) and rely on it to intelligently place both training data and workloads into different regions.

To provide the global-scheduling abstraction, MAST faces two major challenges:

- *Data-GPU colocation*: Without careful coordination, there is a risk of location mismatch between GPUs and data. For instance, one region may have the necessary training data but run out of available GPUs, while another region may have available GPUs but lack the required training data. Due to the massive volume of training data and the limited cross-region network bandwidth, on-demand cross-region data migration can be both costly and time-consuming.
- *Scalability*: MAST allocates not only GPU machines for training but also CPU machines for data preprocessing [58]. As CPU machines may be dynamically reassigned across ML and non-ML workloads based on demand, conceptually, MAST needs to find machines to run ML workloads out of millions of machines spread across tens of regions. Global resource allocation at this scale has not been studied before.

Contributions: Chunqiang, Kutta, and Tuomas initiated the MAST project in 2020. In terms of paper writing, Yang wrote the majority of the paper, followed by Chunqiang. In terms of coding, Arnab, Kutta, and Tuomas led the project's development at different times. All other authors also made major contributions to the project's development.

We leverage three principles to address these challenges: *temporal decoupling*, *scope decoupling*, and *exhaustive search*. We elaborate on these principles below.

Temporal decoupling. We divide the scheduling responsibilities into two paths: a fast path for real-time job scheduling and a slow path that continually optimizes data and machine assignment in the background. The slow path intelligently replicates ML training data across regions, enabling the fast path to more easily colocate computation with data. Despite the relaxed timing, cross-region data placement remains very challenging. It requires continuous optimization of the placement of billions of data partitions across tens of geo-distributed regions, considering per-region capacity constraints and the data access pattern of millions of daily ML training jobs and analytics jobs from Spark [55] and Presto [42].

We model data placement as a mixed integer programming (MIP) problem, and the scarcity of GPUs drives novel decisions in our solution. Due to the high cost and demand of GPUs, we target maximizing GPU utilization. Imposing a hard constraint in the MIP problem that GPU demand must be lower than GPU supply in every region, as in prior work [20] for CPU and storage, often renders the problem unsolvable. Instead, MAST allows GPU oversubscription and preempts low-priority jobs as needed. This approach mandates a reassessment of the objective function and constraints in the MIP problem, not only for GPU-related terms but also for other resources that GPUs depend on. We share insights gained from multiple iterations refining the MIP problem through production experience (§3).

To tackle the scalability challenge, as illustrated in Figure 1, MAST adopts a three-level scheduling hierarchy: Global ML Scheduler (GMS)→Regional ML Scheduler (RMS)→Cluster Manager (CM). In addition to managing data placement, the slow path also helps scale RMS by constraining its search for available machines. It dynamically pre-assigns machines to *dynamic clusters*, allowing RMS to only search through machines within the ML dynamic clusters and disregard non-ML dynamic clusters.

Scope decoupling. A job scheduling system has three main responsibilities. First, it manages the job queue, which entails queuing and prioritizing jobs when there are insufficient resources to run all jobs. Second, it handles resource allocation, which involves computing bin-packing-like solutions by modeling machines as bins and tasks as objects. Third, it manages container orchestration, which executes the bin-packing plan, runs containers, and monitors their health. Traditional systems [19, 47, 48, 56] handle all these responsibilities within the same scope, i.e., within a cluster.

Our key insight is that sharing the same scope for all three responsibilities unnecessarily limits scalability, reducing the potentially larger scopes of job queue management and resource allocation to the minimal scope of container orchestration. Note that container orchestration is the least scalable due to its heavy duties and, consequently, has the smallest scope.

In contrast, as shown in Figure 1, our *scope-decoupling* principle allows the three responsibilities to operate at different scopes: (1) the job queue is managed by GMS at the global scope, covering all pending jobs for all regions; (2) resource allocation is managed by RMS at the regional scope, taking into account all machines in a region's ML dynamic clusters; and (3) container orchestration is managed by the CM at the smallest dynamic-cluster scope. This approach allows job queue management and resource allocation to operate at bigger scopes to minimize stranded resources and optimize job placement. A key challenge is to make GMS and RMS sufficiently scalable to operate at their bigger scopes, which is further discussed in §4.2.1 and §4.3.1.

Exhaustive search. Existing systems [10, 20, 28] often adopt the federation approach to scale out. When a new job arrives, the Federation Manager employs simple heuristics to assign the job to the least loaded cluster, and then its cluster manager manages all subsequent operations, including job queuing, resource allocation, and container orchestration. However, as ML training clusters are almost always fully utilized, scheduling a new job often requires a complex decision to preempting existing lower-priority jobs. This complexity makes the simplistic federation approach less effective.

Our key insight is that, unlike short-lived analytics jobs [10, 38, 41, 50], ML training jobs often run for extended periods on expensive GPUs. Therefore, instead of searching just one cluster to allocate resources hastily, it is beneficial to conduct an *exhaustive search* across all relevant clusters for higher quality placement. As depicted in Figure 1, MAST's multiple RMSs can concurrently compute resource allocation plans for one job in different regions, with the optimal plan determined through a final auction process. A key hurdle is ensuring the scalability of RMS, which is discussed in §4.3.1.

Contributions. We make the following contributions.

- We propose the *global-scheduling* abstraction to shield users from the complexity of geo-distributed datacenters and improve hardware utilization through joint placement of data and training workloads across regions.
- We propose three principles–*temporal decoupling*, *scope decoupling*, and *exhaustive search*–to achieve high-quality data and computation placement in a scalable manner.
- We demonstrate the effectiveness of global ML scheduling through our hyperscale deployment of MAST and validate its design using production data.

2 Background of ML Training at Meta

In this section, we provide the necessary background to set the stage for future discussions.

Datacenter and hardware. Our private cloud comprises tens of regions and millions of machines. A region comprises

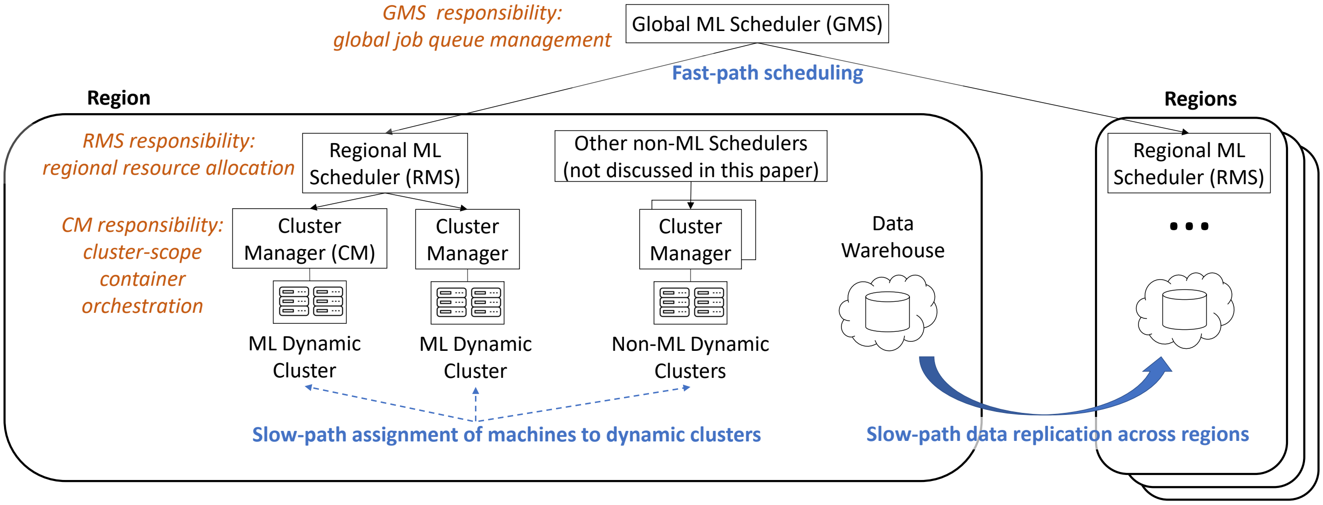

GMS responsibility:
global job queue management
Global ML Scheduler (GMS)
Fast-path scheduling
Region
Regions
RMS responsibility:
regional resource allocation
Regional ML Scheduler (RMS)
Other non-ML Schedulers (not discussed in this paper)
Regional ML Scheduler (RMS)
CM responsibility:
cluster-scope container orchestration
Cluster Manager (CM)
Cluster Manager
Cluster Manager
Data Warehouse
ML Dynamic Cluster
ML Dynamic Cluster
Non-ML Dynamic Clusters
Slow-path assignment of machines to dynamic clusters
Slow-path data replication across regions

Figure 1: Conceptual architecture of MAST. Global ML Scheduler (GMS), Regional ML Schedule (RMS), and Cluster Manager (CM) handle different scheduling responsibilities at different scopes: global, regional, and cluster, respectively.

multiple datacenters that are close to one aother. The cross-region network bandwidth is about 10 times lower than the bisection bandwidth between datacenters within a region. Parts of a datacenter are occupied by ML training clusters, with machines equipped with multiple GPUs and connected by both 8x200Gbps RoCE network and 4x100Gbps Ethernet.

ML training is data intensive and prefers colocating the compute and data of a training workload. For tasks that belong to an ML training workload, we prefer to place them in the same rack, cluster, datacenter, and region, in that order. Separating the compute and data across regions or placing tasks in different regions would result in unacceptable performance.

Historically, datacenter hardware has been procured incrementally depending on the specific needs at different times, resulting in uneven distribution of hardware types across regions. This is discussed in Flux [11] and also shown in Figure 2. This disparity makes colocation of data and compute difficult, requiring global optimization. For example, since *Region6* is short of GPUs, it is better to place data used by CPU-based analytics jobs in *Region6*. If a few GPU-based ML training workloads share the same data as those analytics jobs, we should schedule them in *Region6* as well. However, if there are too many such ML workloads, we will have to replicate their data to other regions and execute them there.

Dynamic clusters. As shown in Figure 1, a slow-path component called RAS pre-assigns machines to *dynamic clusters*, which are known as "*reservations*" in the RAS paper [36]. This enables the Regional ML Scheduler (RMS) to scale by searching only through machines that are within the ML dynamic clusters. Typically, an ML dynamic cluster comprises both GPU and CPU machines. To update dynamic clusters, RAS takes as inputs all machines in a region and the new or updated specification for each dynamic cluster's intended size and preference for certain hardware types. RAS formulates a MIP problem to allocate machines to dynamic clusters. MAST consumes the outputs of RAS (i.e., the dynamic clusters created by RAS), and MAST's scheduling decisions do not influence or feed back into RAS.

We provide a brief summary of RAS and refer readers to the RAS paper [36] for details. RAS ensures that the total machine capacity allocated to a dynamic cluster meets the requirements specified by administrators and includes sufficient buffers to handle both random and correlated machine failures. Correlated failures, such as power outages in large fault domains within a datacenter, can render tens of thousands of machines unavailable. RAS distributes a dynamic cluster's machines across different fault domains to ensure that sufficient healthy machines remain available when a large fault domain fails. Additionally, RAS reduces unnecessary cross-datacenter communication by ensuring a proper ratio of compute machines to storage machines in each datacenter. Finally, RAS reruns its optimization periodically (e.g., every 30 minutes) to adapt to changes. For example, when new datacenters are brought online, RAS can reduce the buffer size needed for handling correlated failures by further spreading out a dynamic cluster's machines into these new datacenters.

ML training workload. A training workload comprises multiple heterogeneous jobs, each job comprises multiple homogeneous tasks, and a task is mapped to a Linux container. Therefore, the hierarchy is workload→job→task. For example, a training workload may comprise (1) a training job that executes back-propagation training; (2) a data-preprocessing job [58]; (3) a parameter-server job; and (4) an evaluator job that evaluates the generated model. A workload's all tasks need gang scheduling, i.e., they must be allocated together. If a training job uses less than a full GPU, in theory, the GPU can be shared by multiple jobs using Multi-Instance GPU (MIG) [37] or other software approaches. In practice,

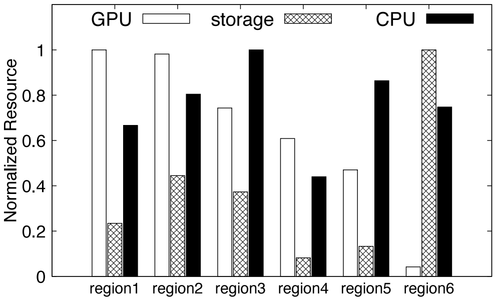


Figure 2: Uneven distribution of hardware across regions. Storage is normalized by capacity, and GPU and CPU are normalized by server count.

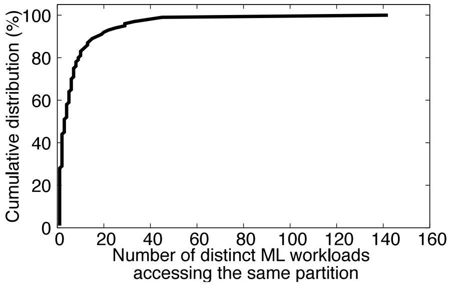


Figure 3: Hotness of data partitions, measured by the number of distinct ML workloads accessing each partition.

however, all our training jobs use at least one full GPU due to the large amount of training data.

Data warehouse. Our data warehouse stores exabytes of data in a three-level hierarchy: hundreds of namespaces→millions of tables→ billions of data partitions. A partition is immutable once created, but new partitions can be added to an existing table. For example, every day, the "*user_activity*" table can add a new partition to record user activities in the past 24 hours. Some data partitions are simultaneously used by ML training and data analytics, such as Spark [55] and Presto [42]. We have developed a system called Tetris, which optimizes data placement across regions, taking into account the data-access patterns of Spark, Presto, and ML training jobs.

Sharing of data partitions by workloads. Figure 3 shows that data partitions are often shared by multiple ML workloads. At the P50, P90, and P99 percentiles, a data partition is shared by 3, 17, and 45 distinct workloads, respectively. Data sharing complicates the problem of data placement, as migrating one data partition across regions may require the migration of multiple workloads dependent on the partition. Furthermore, it is necessary to replicate the hottest partitions across multiple regions to prevent load imbalance, as a large number of workloads dependent on those partitions will otherwise be forced to run in a small number of regions.

Long execution time of ML training jobs. ML training is resource intensive and can take a long time to finish. At Meta, ML training workloads often take 10 times longer to finish than Spark [55] analytics jobs. Therefore, a suboptimal placement decision has a bigger negative impact on ML training. This motivates the *exhaustive search principle* described in §1. Moreover, when workloads run longer on a larger number of machines, the workload scheduling throughput decreases. Therefore, as shown in Figure 1, it is feasible to manage the job queue and resource allocation at the global and regional scope, respectively, rather than at the smaller cluster scope that leads to more fragmentation.

Quota and job preemption. Training workloads with different priorities are assigned capacity quotas per priority level. If a team's capacity usage is within their quota, MAST guarantees starting their training workloads within a certain latency. Once a team exceeds their quota, they can still submit workloads to run opportunistically at the lowest priority, subject to preemption when a higher-priority workload arrives. Consequently, training clusters are always fully utilized due to low-priority workloads for experimental purposes. Scheduling a new workload often involves a complex decision to preempt lower-priority jobs. This complexity renders a simple Federation Manager less effective.

Checkpoint for recovery. A training workload periodically checkpoints its state. When a machine fails, the cluster manager restarts its workload on a replacement machine, allowing it to recover its state from the checkpoint and resume execution. Before preempting a low-priority workload for a high-priority one, it also saves a checkpoint for later restoration. As we continuously reduce the time needed to save a checkpoint, we are moving towards more frequent checkpoints to minimize the amount of lost work between two checkpoints during recovery. This has become increasingly important as the size of training workloads for large-language models keeps growing, and recovery becomes more costly.

Separate application-level schedulers for ML and non-ML workloads. As depicted in Figure 1, ML and non-ML workloads are managed by distinct schedulers. The extensible architecture of Twine [44] allows all workloads to share a common cluster manager for machine and container management, while employing different application-level schedulers for specific workloads. For instance, MAST is used for ML training workloads, Shard Manager [29] for stateful databases, Turbine [32] for stream processing, and Chronos for analytics jobs. Each of these application-level schedulers is optimized for a specific purpose. Shard Manager, for example, is optimized for high database availability, Chronos for high scheduling throughput of short-lived analytics jobs, and MAST for high-quality decisions and data-GPU colocation.

3 Slow-path Data Placement

To enable global ML scheduling, it is crucial to have both the necessary hardware and training data for an ML workload available in certain datacenter regions simultaneously. Following the *temporal-decoupling* principle, MAST optimizes cross-region data placement on a daily basis using a slow-path component called Tetris, which determines data placement and replication for the underlying storage system. Given that data analytics (e.g., Spark [55] and Presto [42]) and ML training can concurrently access the same data, Tetris jointly optimizes data placement for them. The entities accessing the data, such as Presto queries, Spark jobs, and ML workloads, are collectively referred to as "*jobs*" for simplicity.

3.1 Context of Data Placement

Recall that our data warehouse uses a three-level hierarchy: hundreds of namespaces→millions of tables→billions of data partitions. The large number of data partitions presents a significant scalability challenge for data placement. To improve scalability, Tetris determines data placement in two steps: first by placing tables to regions, which means that all partitions of a table must reside in the same region; and then by placing partitions to data centers within each region. Since the algorithms for both steps are similar, we mainly present the table-to-region placement algorithm.

Each table has a home datacenter region, which is where its new data is generated. Subsequently, the table may be replicated to other regions for various purposes. If an ML job takes a table as input, then at least one of the table's regions should have the types of GPUs required by the job. We refer to this property as "*data-GPU collocation*." Due to the large size of our data warehouse (exabytes) and limited cross-region network bandwidth, replicating data takes time. Consequently, a table's replicas in other regions are more stale compared to those in the home region. Therefore, high-priority jobs requiring fresh data must run in the home regions of their input tables, and these home regions must have the types of GPUs required by those jobs. We call this property "*training-at-home-region*." For low-priority jobs, *training-at-home-region* is preferred but not required.

Tetris first determines the home region of each table and then determines replica regions. We are still evaluating the feasibility of determining both simultaneously. While it may result in improved placement, currently we find its computational cost to be too high.

To plan data placement, Tetris requires the usage information of each job's input tables, hardware needs, and estimated runtime. High-priority training jobs, with trained models deployed for immediate production use, are typically retrained daily or more frequently with updated data. The usage information of such recurring jobs rarely changes, so Tetris can derive this information from their historical data. Tetris does not predict usage information for new training jobs that show up for the first time. Instead, MAST's fast-path online scheduling manages first-time jobs upon submission. If no region has both the necessary data and hardware to run a first-time job, MAST will initiate data movement and wait for its completion before scheduling the job. However, many first-time jobs benefit directly from data placement plans for recurring jobs and will not be blocked on data replication, as jobs from the same team frequently share input tables and require the same types of hardware. For instance, multiple one-time experimental jobs are often submitted to fine-tune a parameter of a production recurring job. Our production data in §5 shows that although about 70% of the jobs are first-time jobs, only a small fraction of them trigger on-demand data movement. Note that on-demand data movement may also be triggered due to failures or in rare cases where a recurring job changes its input tables or hardware needs.

3.2 Problem Formulation Overview

We formulate home-region placement as the mixed-integer programming (MIP) problem shown in Figure 4. While MIP has been applied in resource allocation [11, 20], the key insights often lie in the details of a specific problem formulation. In Tetris, we need to carefully evaluate various approaches for resource allocations:

- The *hard-quota* approach mandates that resource demand must stay below supply in every region.
- The *hard-balance* approach does not enforce *hard-quota* but mandates that the overload situation (i.e., demand above supply) must not deteriorate for any region due to a new placement.
- The *soft-balance* approach does not enforce *hard-quota* or *hard-balance* but instead aims to balance the demand-to-supply ratio across all regions as much as possible.

After iteratively improving Tetris based on production experiences, we have learned that different resource types require different approaches. For GPUs, imposing hard constraints in the MIP problem, as in previous work for CPU and storage [20], often renders the problem unsolvable due to the scarcity of GPUs. Hence, we adopt the *soft-balance* approach and preempt low-priority jobs to accommodate high-priority jobs as needed. Specifically, we introduce a penalty if a region's GPU demand deviates from the ideal case where all regions' GPU demand-to-supply ratios are the same. In our initial implementation, the situations of overload and underload were penalized equally. However, in practice, overload is more problematic as it causes longer wait times for the impacted jobs. Therefore, our current implementation more severely penalizes overload.

For storage space, the *hard-quota* approach is necessary because we cannot delete data when demand exceeds supply. However, *hard-quota* alone is insufficient, as an imbalanced data distribution across regions may occur, leading to GPUs in some regions being bottlenecked on I/O bandwidth to access the data. Therefore, we also apply *soft-balance* to storage. In the past, we also experimented with using *hard-balance*, but it proved ineffective as it completely prevents moving data from a region to another region with a higher demand-to-supply rate, which is sometimes necessary for other goals.

Concretely, our MIP problem formulation aims to achieve the following soft goals:

1. Minimize cross-region traffic for reading data during training (Expression 1).
2. Balances the demand and supply of GPUs across regions (Expression 2).
3. Balances the demand and supply of storage space across regions (Expression 3).

In addition, it aims to meet the following hard constraints:

4. Each region has sufficient storage space for the tables it stores (Expression 4).
5. Each application's CPU usage does not exceed its quota (Expression 5). Here, applications refer to those generating data for or sharing data with ML training, such as analytics jobs. An application may comprise multiple jobs.
6. The demand of jobs satisfying the training-at-home-region property remains above a certain threshold (Expression 6).

3.3 Problem Formulation Details

This section can be safely skipped during the initial reading, as it mainly details the problem formulation.

Minimize:

$$w_1 \sum_{\text{job}_j} \sum_{\text{table}_i \in \text{job}_j} \text{size}(\text{table}_i)\ \left[1 - I(R(\text{job}_j), R(\text{table}_i))\right] \tag{1}$$

$$+ w_2 \sum_{\text{region}_i} \sum_{\text{GPU}_j} \sum_{P_k} w_{p_k} \sigma_1(\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}, \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) \tag{2}$$

$$+ w_3 \sum_{\text{region}_i} \sigma_2(\text{Demand}^{\text{region}_i}_{\text{storage}}, \text{Supply}^{\text{region}_i}_{\text{storage}}) \tag{3}$$

Subject to:

$\forall \text{region}\ \ r$

$$\sum_{\text{table}_i} \text{size}(\text{table}_i)\ \ I(R(\text{table}_i), \text{region}_r) < \text{storage\_capacity}(r) \tag{4}$$

$$\forall \text{app} \sum_{\text{job}_j \in \text{app}} \text{CPU}(\text{job}_j)\ \ I(R(\text{job}_j), \text{region}_r) < \text{CPU\_capacity}(\text{app}, r) \tag{5}$$

$$\sum_{\text{high-priority job}_j} \text{GPU}(\text{job}_j) I(\text{GPU-Type}_{\text{job}_j}, R(\text{job}_j)) \geq \text{threshold} \tag{6}$$

Where:

$$\sigma_1(\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}, \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) =$$

$$\left(\frac{\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}}{\sum_{\text{region}_i} \text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}} - \frac{\text{Supply}^{\text{region}_i}_{\text{GPU}_j}}{\sum_{\text{region}_i} \text{Supply}^{\text{region}_i}_{\text{GPU}_j}} \right)^2 \tag{7}$$

$$\times \text{ sigmoid}(\max(0, \text{Demand}^{\text{region}_i}_{\text{GPU}_j} - \text{Supply}^{\text{region}_i}_{\text{GPU}_j}) \tag{8}$$

$$\sigma_2(\text{Demand}^{\text{region}_i}_{\text{storage}}, \text{Supply}^{\text{region}_i}_{\text{storage}}) =$$

$$\left(\frac{\text{Demand}^{\text{region}_i}_{\text{storage}}}{\sum_{\text{region}_i} \text{Demand}^{\text{region}_i}_{\text{storage}}} - \frac{\text{Supply}^{\text{region}_i}_{\text{storage}}}{\sum_{\text{region}_i} \text{Supply}^{\text{region}_i}_{\text{storage}}} \right)^2 \tag{9}$$

Figure 4: Formulation of the data placement problem.

Among the expressions in Figure 4, $R(\text{table}_i)$ is the only decision variable, which determines the home region of table_i. The region in which job_j is placed is represented by $R(\text{job}_j)$, which is not a decision variable and is inferred from the placement of tables. A job may access multiple tables. For Spark or Presto jobs, the home region of the majority of the tables accessed by the job determines $R(\text{job}_j)$. For an ML training job_j, if the home regions of all tables accessed by the job are the same, then their home region determines $R(\text{job}_j)$. Otherwise, Tetris sets $R(\text{job}_j)$ to NULL temporarily, and will fix it at a later stage by replicating tables to ensure that at least one region has all these tables (§3.5). Note that while a Presto or Spark job can read some of its input tables across regions, an ML job must read all its input tables from the local region. This difference is due to the fact that GPUs used by ML jobs are much more costly and should not be stalled on reading data during execution. Further note that $R(\text{job}_j)$ is an auxiliary variable used when determining $R(\text{table}_i)$. After the completion of table placement on the slow path, MAST's real-time job scheduling on the fast path (§4) has the freedom to place the job in a region different from $R(\text{job}_j)$, depending on the available resources and table replicas at the scheduling time. Similarly, all the *Demand* variables are also auxiliary variables, inferred from $R(\text{table}_i)$ and $R(\text{job}_j)$.

Other symbols are defined as follows: $\text{size}(\text{table}_i)$ (size of a table), $\text{CPU}(\text{job}_j)$ (CPU hours required by a job), $\text{GPU}(\text{job}_j)$ (GPU hours required by a job), and $\text{GPU-Type}_{\text{job}_j}$ (types of GPUs required by a job) are estimated from the past history of the tables and jobs. $\text{storage\_capacity}(r)$, $\text{CPU\_capacity}(\text{app}, r)$, and *Supply*, are set by the administrator. The weights, w_1, w_2, w_3, w_{p_k}, and w_{job_j}, can be tuned by the administrator to prioritize certain terms or jobs.

$I(a,b)$ is a binary operator that evaluates to 1 if its two operands meet certain conditions and 0 otherwise. Specifically, $I(R(\text{job}_j), R(\text{table}_i))$ checks if the job and the table are in the same region. $I(R(\text{job}_j), \text{region}_r)$ checks if the job is placed in region r. $I(R(\text{table}_i), \text{region}_r)$ checks if the table is placed in region r. $I(\text{GPU-Type}_{\text{job}_j}, R(\text{job}_j))$ checks if job_j's region has the needed type of GPUs to run the job.

Expression 1 aims to minimize the total size of tables that are read across regions, i.e., when job_j needs to access table_i but they are not in the same region. The notation $\text{table}_i \in \text{job}_j$ denotes the tables accessed by job_j. For an ML training job_j, if its home region $R(\text{job}_j)$ is NULL (i.e., the home regions of the tables accessed by the job are not the same), Expression 1 will assume that all of these tables are accessed across regions.

Expression 2 *soft-balances* the supply and demand of GPUs across regions. GPU_j is a specific type of GPU and P_k is a job priority level. $\text{Demand}^{\text{region}_i}_{\text{GPU}_j, P_k}$ is the demand for GPU_j at region_i at priority level P_k. $\text{Supply}^{\text{region}_i}_{\text{GPU}_j}$ is the supply for GPU_j at region_i. Demands and supplies are measured in GPU hours. Expression 7 minimizes the imbalance of supply and demand across regions. If the load is perfectly balanced, Expression 7 is zero. Expression 8 minimizes region oveload. Without overload, Expression 8 is a constant, $\text{sigmoid}(0) = 0.5$. As the region becomes more severely overloaded (i.e., demand above supply), Expression 8 approaches 1, and the attempt to minimize Expression 8 would lead to reducing overload.

Expressions 3 and 9 *soft-balance* the supply and demand of storage space across regions. However, there is no term similar to Expression 8 to minimize storage space overload, as it is disallowed to allocate more space than available, enforced by Expression 4. Similarly, Equation 5 ensures that every region's CPU demand is below supply. Note that there is no such hard constraint for GPUs, as it would often lead to an unsolvable problem due to GPU scarcity.

Expression 6 only applies to time-sensitive high-priority jobs, ensuring that most of these jobs are trained in their home regions, i.e., satisfying the *training-at-home-region* property. Moreover, we empirically validate but do not mandate that the number of jobs satisfying this property does not decrease over time.

3.4 Efficient Approximate Solution

The MIP problem in Figure 4 is NP-hard. Tetris uses a hill-climbing algorithm [26] to compute an efficient approximate solution. Starting from the current placement, Tetris goes over each table and finds the best region for it out of different home region choices. The best region should have the lowest cost (Expressions 1-3) and can satisfy all constraints (Expressions 4-6). If the best region differs from the current home region of the table, the table is added to a move queue. The queue is ordered by the gain of table moves, i.e., the initial cost before the move minus the new cost after the move.

After determining the new home region for all tables, Tetris iterates over the queue to move tables, starting from the table with the highest move gain. Before moving a table, it recalculates its best region because moving tables in prior iterations may have changed the cost of moving this table. The moves continue until either the queue becomes empty or a daily move quota is reached due to limited cross-region network bandwidth. Tetris runs this algorithm daily which takes about five hours to complete, although the actual data replication may take much longer to finish.

To move a table, Tetris replicates the table to the new home region before deleting its data in the old home region. A data-migration service schedules and executes cross-region data replication. Tetris can set a soft deadline for a migration operation, and the migration service allocates the necessary bandwidth accordingly. Our network's cross-region traffic receives differentiated quality of service to ensure that background bulk data replication does not affect latency-sensitive services. As for the data-replication size, the P50 (50th percentile) is 565MB, the P90 is 103GB, and the P99 is 7.5TB. In terms of data transfer time, the P50 is 2.1 hours, the P90 is 3.7 hours, and the P99 percentile is 4.9 hours.

It is possible that the new home region becomes suboptimal during the move due to various reasons. Typically, the next day's rerun of Tetris will correct the problem. However, if MAST needs to schedule the corresponding job before the rerun, it can use heuristics to determine whether on-demand data movement is needed to temporarily fix the issue.

In addition to the hill-climbing algorithm, we have evaluated other solutions such as commercial MIP solvers [12, 18], but a comprehensive comparison with local-search alternatives is yet to be conducted. We chose hill-climbing for several reasons. Firstly, it offers greater scalability than other solutions we have explored. Secondly, hill-climbing simplifies the definition of the MIP problem formulation. As mentioned in §3.3, $R(\text{table}_i)$ is the sole decision variable, with $R(\text{job}_j)$ inferred from $R(\text{table}_i)$, and GPU and storage demands per region further inferred from $R(\text{job}_j)$. In traditional MIP formulas, $R(\text{job}_j)$ must also be declared as a decision variable constrained by $R(\text{table}_i)$, and the same must be done for all demand requirements. This would lead to mathematical formulas much more complex than those shown in §3.3. In contrast, with hill-climbing, we can compute $R(\text{job}_j)$ and demand requirements from $R(\text{table}_i)$ using code, which offers more flexibility. Lastly, the outputs of hill-climbing are interpretable and easy to debug, as we understand the exact reasons to migrate tables in each iteration. Given that hyperscale ML training is still relatively new and the problem formulation continues to evolve, the interpretability and debuggability of hill-climbing present significant advantages.

3.5 Creating Extra Table Replicas

In addition to a table's home region, it may have replicas in other regions for various reasons. For disaster recovery (DR) purposes, each table has a replica outside of its home region. The region that stores the DR replica is selected based on a DR policy, which takes into account factors such as the probability of correlated regional failures and the availability of hardware to handle increased load after a disaster.

After the hill-climbing algorithm finishes, some ML jobs may not be able to run in any region, because their home region $R(\text{job}_j)$ is NULL or does not have the type of GPUs needed by the job. The DR copy may fix some of these jobs, and for the remaining ones, Tetris creates additional replicas of the tables accessed by them in other regions, allowing them to be scheduled there. For example, suppose table_i is accessed by job_1 and job_2, which require GPU_1 and GPU_2, respectively. However, GPU_1 and GPU_2 are not available together in any region. Thus, Tetris may set $R(\text{table}_i)$ to be a region that has GPU_1 to run job_1 and then create a replica of the table in a region that has GPU_2 to run job_2. If multiple regions have GPU_2, Tetris selects the region with the highest supply.

A small fraction of tables, known as hot tables, are accessed by a large number of jobs (Figure 3). If these tables are replicated to only a few regions, it may create a significant load imbalance, as a large number of jobs dependent on these tables will be forced to run in a few regions. To address this problem, Tetris widely replicates hot tables per GPU type.

		Scope of job queue management		
		Cluster	Regional	Global
Scope of resource allocation	Cluster	(1) Borg, Hydra, Yugong	(2)	(3)
	Regional	(4) ✗	(5)	(6) MAST
	Global	(7) ✗	(8) ✗	(9) “Ideal”

Table 1: Design space partitioned by the scheduling scope. The symbol ✗ indicates invalid solutions.

For each region and each GPU$_j$, Tetris computes the region’s storage quota for GPU$_j$ based on the supply of GPU$_j$. Tetris sorts tables accessed by jobs using GPU$_j$ based on the tables’ hotness, and replicates as many hot tables as possible until it reaches the storage quota for GPU$_j$.

Extra table replicas provide MAST’s fast path with greater flexibility to choose the region for hosting a job at runtime. However, replicating a massive amount of data across regions could take hours, delaying the start of some time-sensitive jobs. To address this problem, Tetris takes a combination of measures. First, Expression 6 enforces that sufficient jobs are trained in their home regions. Second, instead of replicating the whole table, Tetris can be configured to only replicate the partitions needed by the training job. Finally, Tetris prioritizes first replicating data needed by high-priority jobs to meet their deadlines.

Overall, the additional table replicas created by Tetris increase storage consumption by approximately 75% to 125%. However, given the high cost of GPUs, we deem this a justifiable trade-off.

4 Fast-path Job Scheduling

While the slow path asynchronously prepares data for ML training, the fast path schedules ML workloads in real-time. Before presenting MAST’s scheduling solution, we explore the design space to understand its rationale.

4.1 Exploring the Scheduling Design Space

Traditionally, the federation approach [10, 20, 28] employs *early binding*, dispatching a new job to a cluster based on the current estimated cluster load, even if the cluster has no available resources to run the job immediately. In contrast, MAST adopts *late binding*, dispatching a job to a cluster only when certain that the cluster has available resources to run the job immediately.

Moreover, traditional scheduling systems handle all scheduling functions at the cluster scope: job queue management, resource allocation, and container orchestration. Below, we explore solutions that manage job queues and resource allocation at the regional or global scope.

Comparing solutions. Table 1 shows the solution space. Solution (1) is the traditional approach, with Borg [48] as an example, where job queue, resource allocation, and container orchestration are all managed at the cluster scope. Solution (6) is our approach, where the job queue is managed at the global scope and resource allocation is managed at the regional scope. Solution (9) is the “ideal” approach, where both the job queue and resource allocation are managed at the global scope. With a global view of all jobs and machines, theoretically, it can achieve optimal job placement, but the limited scalability of this approach is a main shortcoming.

For ML training workloads, it can be proven that among algorithms that schedule one job at a time, MAST achieves the same optimal job placement as (9) due to several reasons. First, all tasks of a training workload must be allocated to the same region for locality, simplifying resource allocation calculations to within regions. Moreover, following the *exhaustive-search* principle, MAST computes a resource allocation plan for the workload in *every* region with training data, and then chooses the best plan for execution. As a result, it can achieve optimal placement for individual workloads. However, solution (9) has an advantage over MAST in jointly optimizing the placement of a set of workloads. For instance, after MAST places *workload 1* in region X, it may discover that no region can accommodate *workload 2*. In contrast, solution (9) may intentionally place *workload 1* in region Y and leave region X to handle *workload 2*.

Solutions (2), (3), and (5) improve upon solution (1) as they can either better balance the load or manage allocation at a larger scope, but they still cannot provide the same level of scheduling quality as (6). Solutions (4), (7), and (8) are invalid as their scope of resource allocation is bigger than their scope of job queue management. Among the solutions in Table 1, assuming (9) is not scalable, our preference is (6) > (3) > (5) > (2) > (1).

Federated systems. Hydra [10] and Yugong [20] use the federated approach. Both fall under solution (1) as they employ early binding of a job to a cluster, and then manage the job queue and resource allocation with the cluster. This approach aligns well with the nature of lightweight analytics jobs, the focus of Hydra and Yugong. Such jobs demand high scheduling throughput but typically do not involve complex decisions like job preemption.

4.2 Global ML Scheduler (GMS)

Adhering to the *scope-decoupling principle* and as illustrated in Figure 1, MAST splits the scheduling responsibilities among different components: the Global ML Scheduler (GMS) manages the global job queue, the Regional ML Scheduler (RMS) allocates regional resources, and the Cluster Manager (CM) is responsible for container orchestration.

The main responsibility of GMS is to select the next workload to schedule among all pending workloads in the global job queue. For each pending workload, GMS calculates a $<$*priority*, *credit*$>$ tuple. It schedules the workload with the highest *priority* and, in case of a tie, selects the one with the highest *credit* for scheduling.

The *priority* is affected by quota usage. Each workload belongs to a tenant, i.e., a team. Tenants are assigned a priority level and a quota for running their workloads. The quota specifies the maximum number of GPUs and CPUs that a tenant can use simultaneously. Workloads from a tenant that has not exhausted its quota are assigned the tenant's priority, categorized as *within-quota workloads*. Conversely, when a tenant has used up its quota, its workloads are assigned the lowest priority and categorized as *over-quota workloads*. These workloads run opportunistically and can be preempted when a higher priority workload arrives. Tenant priorities are manually assigned based on business priorities. The strict adherence to these priorities does result in preemption, which is the intended effect. Currently, MAST uses seven priority levels. The distribution of workloads across these priorities (from highest to lowest) is as follows: 3%, 20%, 16%, 54%, 0.2%, 0.5%, and 0.02%. The remaining 6% of workloads do not specify a priority and are consequently treated as the lowest priority.

The *credit* of a workload is calculated as follows. Intuitively, a workload W has a higher *credit* if it has been waiting longer (Expression 10) or belongs to a tenant that has used fewer resources than others (Expression 11).

$$\text{credit(L)} = w_{\text{workload\_age}} \times \min(\text{L.age}, C_{\text{age\_cap}}) \tag{10}$$

$$+ w_{\text{fair\_share}} \times (1 - \frac{\text{window\_avg(L.tenant.resources\_used)}}{\text{window\_avg(all\_tenants.resource\_used)}} \tag{11}$$

Here, L represents a workload, $C_{\text{age\_cap}}$ is a constant, and $w_{\text{workload\_age}}$ and $w_{\text{fair\_share}}$ are tunable weights.

To determine the $<$*priority*, *credit*$>$ tuple for a workload, GMS calculates each workload's *credit* and sorts pending workloads based on it. It scans them to assess if they can be scheduled within their tenant's quota. GMS maintains a *resource_used* variable for each tenant, initialized to include resources used by the tenant's running workloads. When scanning a pending workload W, GMS checks if adding W's resource requirement to *resource_used* would exceed the tenant's quota. If so, W is assigned the lowest priority level; otherwise, W inherits its tenant's priority level, and GMS updates the tenant's *resource_used* to include W's resources.

Periodically, GMS executes a *GMS-scan pass* to update $<$*priority*, *credit*$>$ tuples for all pending workloads. This approach is chosen because the state change of one workload may impact others' $<$*priority*, *credit*$>$. Specifically, a workload's *credit* is influenced by other tenants' resource usage (Equation 11). Additionally, a workload's *priority* is tied to its tenant's other workloads' resource usage. This updating-all-workloads strategy enables MAST to implement sophisticated quota and priority management policies. The scalability of GMS with this approach depends on the frequency and duration of the GMS-scan pass.

4.2.1 Scalability of GMS

In practice, GMS scales well for ML workloads due to several factors. First, ML training workloads, running for extended periods on many machines, necessitate higher-quality scheduling decisions but lower scheduling throughput compared to short-lived batch jobs [10, 38, 41, 50]. Second, GMS has minimal responsibilities, calculating <priority,credit> for each pending workload and storing it in the job-queue database. Placement plans are computed by RMS, not GMS. Third, our evaluation in §5.4 indicates that the current GMS implementation can support workload growth by a factor of 8.8. Finally, currently implemented in Python for simplicity, if it becomes a bottleneck, we plan to scale it further by a factor of 10-100 by switching to C++ and parallelizing computation for different tenants.

4.3 Regional ML Scheduler (RMS)

RMSs perform auctions in a distributed manner to schedule ML workloads. Each RMS constantly checks the job-queue database maintained by GMS, and attempts to schedule the ML workload with the highest $<$*priority*, *credit*$>$.

To schedule a workload, an RMS consults a real-time component of Tetris to check whether its local region has the required data and necessary hardware types. If not, the RMS abandons the auction. If all RMSs abandon the auction, potentially occurring with the first-time execution of a new workload, MAST will start data replication, waiting for its completion to ensure some regions have both the necessary data and hardware.

Typically, due to Tetris, multiple regions have the required data and hardware types for the workload. Following the *exhaustive-search* principle, in each such region, RMS calculates a placement plan for the workload along with a corresponding placement-quality score (P_{score}). RMSs engage in an *auction* to identify the RMS with the highest P_{score}, which will execute the workload. If no region can generate an immediate placement plan for the workload, it enters the *waiting* state.

A region may have multiple ML dynamic clusters, and the workload can comprise multiple jobs, each assignable to a different cluster. Adhering to the *exhaustive-search* principle, for each job in the workload, RMS calculates a placement plan and P_{score} for every ML dynamic cluster in the region. It chooses the cluster with the highest P_{score} to host the job. The overall P_{score} for the workload is determined by summing up the P_{score} for each job in the workload.

When comparing two placement plans for a job, the one with a higher P_{score} wins. A plan has a higher score if it uses available resources to run the new job without preempting any running jobs. If preemption is necessary, a higher score is achieved by preempting jobs with lower priority, fewer jobs, or those running for a longer duration. The last condition implies refraining from preempting newly started jobs.

To generate a placement plan for a job in an ML dynamic cluster, RMS checks if it can allocate the job using available resources without preempting running jobs. For enhanced task locality within the same job, RMS sorts available machines based on rack IDs, allocating machines in the same or nearby racks in batches. While scanning, RMS aims to use the minimum number of machines by prioritizing those with the highest available CPU/GPU resources.

If preemption is necessary, RMS prioritizes preempting lower-priority jobs. It sorts all running jobs based on priority and running time, initiating the scan with the job having the lowest priority and longest running time. The scanning process continues until sufficient resources are found to execute the new job, combining available resources with those to be released through preemption.

RMS minimizes preempted jobs. For example, if a new job needs 10 GPUs and the scan reveals options of 4, 5, and 12 GPUs by preempting job_1, job_2, and job_3 respectively, the optimal choice is preempting job_3 without affecting job_1 or job_2. To achieve this, RMS re-sorts preempted jobs by size, prioritizing larger jobs first.

Multiple optimizations improve RMS performance, with the most effective being the use of a *negative cache*. When RMS cannot allocate resources for a job, it saves the decision in the cache. If it later attempts to allocate a job of the same or larger size, the cache signals that the allocation will fail. Specifically, the negative cache is initialized at the start of every GMS-scan pass that examines all pending jobs, and is cleared after the GMS-scan pass finishes. It is implemented as a hashtable that stores the scheduling properties of the jobs that could not be scheduled during the GMS-scan pass. These scheduling properties encompass the number of GPUs and CPU cores requested by the job, memory, hardware type, and so on. Such information consumes little memory, and there is no need for cache eviction during a GMS-scan pass. Overall, the negative cache is highly effective as unsuccessful placement attempts far outnumber successful ones due to nearly constant full resource allocation. It filters out the majority of these unsuccessful attempts early on, thanks to a cache hit rate of about 80% in practice.

4.3.1 Scalability of RMS

RMS demonstrates sufficient scalability, as evidenced by the analysis below. Scalability is examined concerning the number of regions (r) and the amount of ML hardware per region (h). Adding more regions does not increase the computation load of RMS, as it schedules workloads only for data stored in the respective region. When h remains constant, the number of such workloads also remains unchanged. However, linear growth in h results in quadratic growth in RMS's computation load. The complexity of RMS's exhaustive search is $E(h) = O(D(h) \times J(h))$, where D is the number of ML dynamic clusters and J is the number of jobs scheduled on these clusters. As both D and J are proportional to h, $E(h)$ experiences quadratic growth.

Our evaluation in §5.4 demonstrates that RMS can handle a 12x increase in h compared to our current production load. This scalability is likely sufficient even for the long run, given that the growth of h is constrained by the fixed electricity supply of a datacenter region. Currently, the largest RMS manages around 20 dynamic clusters, comprising a total of 64,000 CPU machines and 20,000 GPUs. In the improbable scenario of RMS becoming a bottleneck, we plan to parallelize scheduling for non-conflicting workloads with training data in non-overlapping regions. Additionally, if needed, RMS can be sharded to scale out, with each shard handling a subset of ML dynamic clusters.

4.4 Cluster Manager

Our cluster manager (CM), Twine [44], has the distinguishing feature of managing a dynamic cluster whose machine membership may be continuously updated by RAS [36]. The CM instances managing ML and non-ML clusters are separate and do not interfere with each other, while RAS can dynamically move machines between them to avoid stranded capacity.

We choose not to use a single, generic cluster to handle mixed ML and non-ML workloads, as it is suboptimal for our large-scale operations. Our large fleet size necessitates partitioning machines into independent clusters for effective management. Combining ML and non-ML workloads in a cluster compromises optimization for either type, whereas our scale benefits significantly from workload-specific optimization. For instance, online services prefer spreading across fault domains, whereas ML training workloads prefer not to be spread widely for better network performance. Furthermore, as gang jobs, ML training workloads prefer, for example, 10 out of 100 jobs to fail entirely while the remaining 90 jobs continue, as opposed to each job experiencing a 10% task termination. Optimizing for spread would lead to the latter undesirable situation. Previously, CM handled these complex differences between ML and non-ML workloads on the fast path of real-time job scheduling, often resulting in suboptimal choices due to limited computation time. Consequently, we adopted the strategy of RAS running on the slow path to pre-built separate ML and non-ML dynamic clusters that are deeply optimized for respective workloads, while simplifying the responsibilities of CM on the fast path.

CM and RMS collaborate in managing workloads. For example, when new machines are added to an ML dynamic cluster, the corresponding CM notifies RMS of this change. With complete information cached in its memory, RMS can efficiently compute placement plans. When a region's placement plan is selected as the best plan for execution, its RMS directs the corresponding clusters' CMs to execute the plan and run the relevant jobs. This may require preempting running jobs and checkpointing their current status, initializing containers for the new jobs, and restoring their job states if they were previously preempted.

4.5 Fault Tolerance

GMS, RMS, and CM are fault-tolerant and highly available, operating in a leader-follower mode. Specifically, two instances of GMS run in different regions, two instances of RMS operate in the same region, and three instances of CM serve the same cluster. They all follow a stateless design, storing their persistent state in a shared and replicated database. In the event of a leader failure, the follower can reconstruct its state from the database and the lower-level component (i.e., GMS from RMS and RMS from CM).

4.6 Limitations

In our hyperscale production environment, we prioritize implementation simplicity and robustness, leading to some implementation limitations rather than inherent design flaws. One such limitation is the inability to distribute a job's tasks across different clusters, despite the capability to allocate workload jobs to various dynamic clusters. RMS can compute a placement plan for distributing a job's tasks across different clusters by leveraging its comprehensive view of all resources in the ML clusters within the region. However, the integration with our cluster manager [44]'s "virtual job" feature, crucial for effectively managing scattered tasks as one virtual job, is not yet implemented.

Another implementation limitation is that currently, RMS schedules only one ML workload at a time. While it parallelizes the computation of placing multiple jobs from one workload into different clusters, it does not commence scheduling the next workload until a decision has been made for the current one. This simple approach is used because it is adequate and still has headroom to support further growth, as discussed in §5.4. However, if a bottleneck arises in the future, we are prepared to transition to scheduling multiple workloads in parallel. Moreover, scheduling multiple workloads simultaneously presents opportunities for enhancing scheduling quality. Note that although the current implementation schedules one ML workload at a time, different workloads can still run in parallel. Once a workload X is dispatched to run, without waiting for X to finish execution, the scheduler immediately schedules the next workload Y.

5 Evaluation

Our evaluation attempts to answer the following questions:

1. What are the important ML workload statistics?
2. Can MAST achieve a high resource allocation rate?
3. Is Tetris effective in ensuring colocation of data and compute resources?
4. Are GMS and RMS sufficiently scalable?
5. How long does it take to schedule a workload?
6. How does MAST compare with alternative solutions?

5.1 ML Training Workload Statistics

Currently, MAST is scheduling tens of thousands of ML training workloads daily across tens of regions, consuming $O(100,000)$ GPUs and $O(100,000)$ CPU machines. About 70% of the workloads use GPUs for training, while the remaining use CPUs for training. About 30% of the workloads are recurring, while the remaining are first-time workloads. On average, each workload gets preempted once, which leads to about 100,000 scheduling attempts daily. High-priority workloads may never get preempted and low-priority workloads may get preempted multiple times.

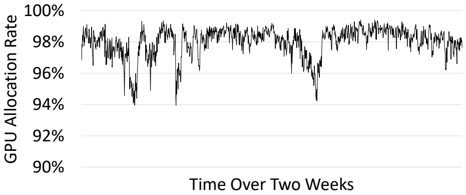


Figure 5: GPU allocation rate.

In terms of the number of machines utilized by a workload, the values for the 50th, 90th, and 99th percentiles are 72, 180, and 205, respectively. As for the number of GPUs used by GPU-consuming workloads, the 50th, 90th, and 99th percentiles are 16, 64, and 128, respectively, and the largest workload today, LLM pre-training, uses tens of thousands of GPUs. We measure the duration of workloads in terms of their execution time until the subsequent preemption, as it is more pertinent for a scheduling system. For GPU workloads, the 50th, 90th, and 99th percentiles of the duration are 20 minutes, 6.7 hours, and 66 hours, respectively. As for CPU workloads, the corresponding percentiles are 38 minutes, 7.9 hours, and 38 hours. Overall, training workloads run on many machines for an extended period. Therefore, it is worthwhile to spend time computing high-quality scheduling decisions.

5.2 Effectiveness of Global ML Scheduling

Thanks to the flexibility of placing both data and workloads globally, MAST has achieved a high average *allocation rate* of 98% for its GPU machines, as shown in Figure 5. The 2% loss is due to factors such as the overhead of preemption, inherent latency of scheduling, and imbalanced data and GPU distribution across regions. The allocation rate is determined by dividing the total hardware hours allocated to workloads by the total available hardware hours. Note that the GPU allocation rate differs from GPU utilization because even if some GPUs are allocated to a workload, they may be underutilized due to various factors, such as the workload's internal communication bottlenecks. We use the allocation rate as the metric because it is more relevant for a scheduling system, which is the primary focus of this paper, while GPU utilization is more pertinent for the ML training framework.

In comparison, the allocation rate for CPU machines is lower. Dedicated CPU machines for ML workloads have an allocation rate of 87%, while elastic CPU machines, which are borrowed temporarily from non-ML workloads, have an allocation rate of 72%. The lower allocation rate for CPU machines is largely due to slight overprovisioning to guarantee that costly GPU machines are never left idle due to a lack of available CPU machines to work with.

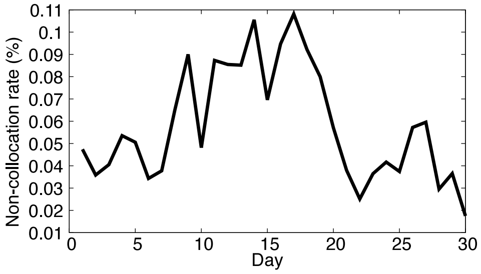


Figure 6: Percentage of workloads that have to wait for data.

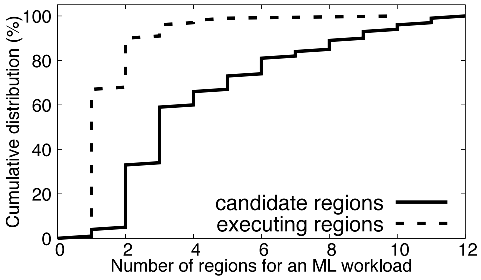


Figure 7: Cumulative distribution of candidate and executing regions per workload over 30 days.

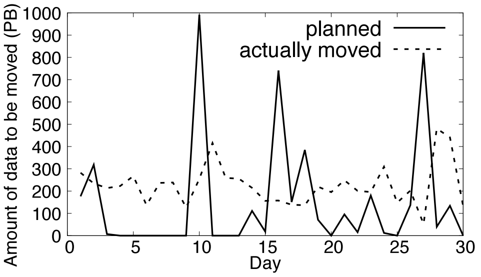


Figure 8: Daily data movement by Tetris.

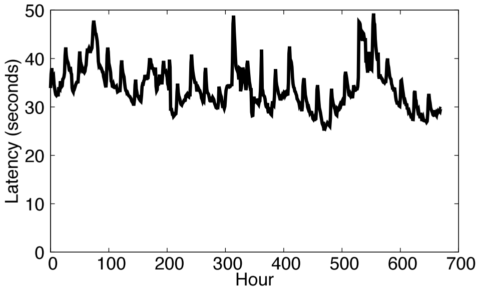


Figure 9: The latency of a GMS-scan pass in GMS (each data point is the average latency within one hour).

5.3 Effectiveness of Data Placement

Figure 6 shows the percentage of workloads that cannot satisfy the data-GPU-collocation property (§3.1) and thus have to wait for on-demand data movement to complete. The non-collocation rate is usually below 0.1%, demonstrating the effectiveness of Tetris. Although Tetris creates extra table replicas to increase the collocation rate (§3.5), non-collocation may still occur due to a workload appearing for the first time or due to machine maintenance rendering certain data or hardware unavailable. Considering 70% of our workloads are first-time ones, this figure shows most of them can still benefit from the data placement planned for recurring workloads.

Thanks to Tetris, often multiple regions can host the same workload, which gives MAST the flexibility to migrate the workload across regions on different days. In Figure 7, the "*candidate regions*" have the required training data and hardware types to host a workload, while the "*executing regions*" have actually hosted the workload on different days. As shown in the figure, the majority of workloads have two or more candidate regions, and approximately 40% of them have four or more. The number of executing regions per workload is smaller; roughly 30% of the workloads have more than one executing region, indicating they are relocated across regions during this period. This demonstrates that global ML scheduling indeed works as intended to dynamically optimize workload placement across regions.

Figure 8 shows the amount of data that Tetris move daily. The spikes in the planned movement are due to onboarding a new workload, which caused a large amount of data to be moved across regions. However, the data migration service has a limit on the maximum amount of data moved per day. Therefore, the actual amount of data moved per day is flatter. This figure shows that Tetris proactively moves hundreds of petabytes of data across regions daily. This enables the fast path to more easily colocate computation with data, and achieve the high GPU allocation rate of 98%.

Tetris's hill climbing algorithm runs daily on a single machine and it typically takes about five hours to finish. Although its CPU utilization is not very high, it spends a significant amount of time on I/Os to fetch various metadata necessary for computing the data placement plan. Currently, the performance of Tetris is not a major bottleneck, and can be further optimized as needed.

5.4 Scalability of GMS and RMS

Scalability of GMS. GMS periodically computes the $<priority, credit>$ tuples for all workloads (§4.2). We refer to one round of such computation as a *GMS-scan pass*. The scalability of GMS depends on the frequency and latency of the GMS-scan pass. Although its theoretical complexity is $O(N \log N)$ due to sorting, where N is the number of workloads, its actual execution time is approximately $O(N)$, dominated by the sequential computation of $<priority, credit>$ for each workload. Figure 9 shows the average latency of the GMS-scan pass. On average, it takes approximately 34 seconds for GMS to rank 6,000-10,000 workloads. Our evaluation shows that running the GMS-scan pass once every 5 minutes still produces high-quality scheduling. This implies that GMS can support $\frac{5\text{ minutes}}{34\text{ seconds}} = 8.8$ times more workloads. As described in §4.2.1, if needed, we can scale the GMS by another factor of 10-100 by switching from Python to C++ and parallelizing its computation for different tenants.

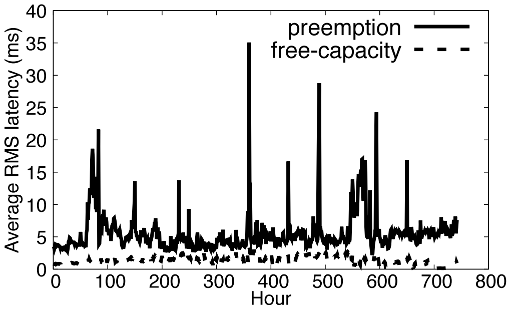


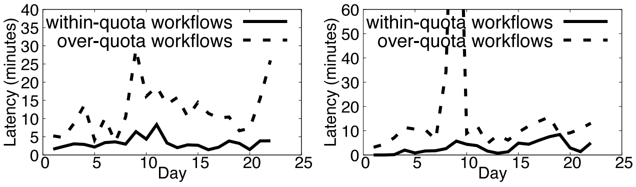
Figure 10: Average latency of scheduling a workload at the slowest RMS. It is computed by taking the maximum of the average latencies computed at each RMS every minute.

Scalability of RMS. Since our current RMS implementation does not initiate scheduling of the next workload until a decision has been made for the current one (§4.6), the maximum throughput of RMS can be estimated from its latency to schedule a single workload. As shown in Figure 10, the slowest RMS takes approximately 1.3ms and 5.5ms to schedule a workload without and with preemption, respectively. Even assuming that all scheduling requires preemption, the RMS can schedule 24*3600*1000/5.5=15 million workloads per day. Considering the current throughput of about 100K scheduling attempts per day and the quadratic growth of the computation load caused by exhaustive search (§4.3.1), the RMS can support approximately $\sqrt{\frac{15M}{100K}}$=12x more workloads and 12x more ML hardware. See §4.3.1 for a discussion on how to further scale RMS.

5.5 Scheduling Latency

Figures 11(a) and (b) show the average and P95 latency, respectively, for pending workloads to enter the running state, including queuing delay, scheduling-algorithm run time, and preemption time but not workload execution time. The preemption time is a major factor in the total delay. If workload A preempts B, rescheduling B counts as a new scheduling event in these figures, starting from the time B is preempted to the time B runs again. The preemption time of B counts towards A's latency as A needs to wait for the preemption to finish. One primary service level objective for MAST is P95 latency. These figures show that MAST has maintained consistently low latency for within-quota workloads. However, for over-quota workloads, the latency can occasionally be erratic, depending on the workload mix. This emphasizes the importance of distinguishing between within-quota and over-quota workloads.

To start a workload, MAST needs to acquire containers from the cluster manager and set up all containers. The P50 latency of this step is 150 seconds, while the P90 is 278 seconds, and the P99 is 449 seconds. For massive LLM jobs, the whole process could take 10 minutes or longer.



(a) Average

(b) P95

Figure 11: Latency for a pending workload to start running.

5.6 Comparison with Alternative Solutions

Tetris. The closest work to Tetris is Alibaba's Yugong [20], which uses MIP to place data for analytics jobs based on CPU (but not GPU), storage, and network constraints. It enforces hard quotas and would not produce a solution when resources are insufficient. However, as GPUs are scarce in our environment with consistently higher demand than supply (§3.2), Yugong would never provide a solution.

Since Yugong is not comparable to Tetris, we evaluate Tetris's different versions to demonstrate the importance of its key features. In the initial stage (V0) of MAST, users manually selected regions for table placement, and tables could not migrate across regions. In 2022, V1 was developed, which automated data placement, aligning with the approach in Section 3, albeit with key distinctions. V1 was a major improvement over V0, reducing the GPU demand-over-supply ratio for high-priority jobs in the most overloaded region from 2.63 to 0.98, with the standard deviation dropping from 0.76 to 0.30. Moreover, it boosts the training-at-home-region rate from 90.82% to 93.02%.

Despite V1's success, various limitations prompted the development of V2. As described in § 3.3, key differences include V2 penalizing GPU overload more than penalizing underload, and incorporating *soft-balance* alongside *hard-quota* for storage. Additionally, V1 defined the collocation rate uniformly for all workloads, treating small and large, high-priority and low-priority workloads alike. Recognizing the practical importance of large high-priority workloads, V2 has revised this definition to be the demand (i.e., GPU hours) of high-priority workloads that can satisfy colocation. With these enhancements, V2 significantly increased the collocation rate under the new definition from 78% to 96% compared to V1.

Fast-path Scheduler. We built a simulator to evaluate different scheduling algorithms of the fast path. The simulator takes a trace of past workloads as input, and follows the same logic of MAST's fast-path scheduler, excepts that it does not actually execute a workload, but instead assumes its running time is the same as recorded in the trace. For comparison, we modified the simulator to implement a federated approach, which dispatches a workload to the region with the lowest demand over supply rate of the required GPU type. When playing an 8-hour trace to the simulator, we find that for MAST, the rate of workloads violating SLOs stays below 1.3% all the time. For the federated approach, however, the SLO violating rate is much higher, especially when load is high. High-priority workloads suffer more, reaching a 50% SLO violating rate during busy hours. This is because high-priority workloads often need to preempt low-priority ones to guarantee SLOs, but with a simple heuristic, the federated approach may not be able to dispatch a high-priority workload to the right region where it can preempt others.

6 Discussions

Solutions suitable for smaller organizations. While MAST is designed for hyperscalers, some of its principles can be applied to smaller organizations. If a small organization only needs to run training workloads in a single cluster, implying one region, then it does not need any of MAST's advanced capabilities. However, in the event of a power or network outage affecting the region, it would be unable to run any ML training jobs due to the lack of disaster-recovery capability.

If an organization's infrastructure operates in at least two regions to be disaster ready, then it can leverage the key ideas in MAST. Without MAST, they would default to solution (1) in Table 1, leading to suboptimal resource allocation due to the isolated operation of the two regions. With insights from MAST, if the scale of their infrastructure is small enough to be handled by a single resource allocator, they could adopt solution (9), yielding optimal placement results. If they do not want to significantly modify their cluster manager like Kubernetes [27], they could at least adopt solution (3), which involves a relatively minor change to use a global job queue but still offers significant benefits in balancing the load across regions. Furthermore, if they cannot afford to replicate every table across every region, they would need a component similar to Tetris to intelligently determine data placement.

Future work. Currently, Tetris considers storage quota and network bandwidth as hard constraints. It is valuable to understand the impact of adding storage and cross-region network bandwidth to better utilize expensive GPUs.

For scalability, Tetris currently determines home regions (§3.2 to §3.4) and creates additional table replicas (§3.5) in separate steps. We plan to explore whether there exists an efficient MIP problem formulation that can simultaneously determine the optimal number of replicas and the home region for each table. It is important that such a problem formulation can be solved in a scalable manner.

Finally, we plan to leverage the fact that the slow path (Tetris) not only determines data placement but also, as a byproduct, calculates the placement of recurring jobs. The latter is currently overlooked by the fast path when placing jobs. Because unexpected one-time jobs may make some of the slow path's job placement decisions unfeasible or suboptimal, future research is needed to better connect the fast and slow paths. This would allow us to benefit from the job-placement decisions that the slow path has more time to compute.

7 Related Work

Scheduling within a cluster. There are many prior works about scheduling within a single cluster, including general-purpose schedulers [14,16,19,22,38,47,56] and those specific to ML training [1,2,5–7,9,13,17,21,23–25,30,31,33–35,39,40,45,49,51–54,57,59]. The latter often considers specific characteristics of ML training, such as model accuracy, gang scheduling, sensitivity to network topology, heterogeneity of compute resources, elasticity, etc. Cluster-level scheduling is largely orthogonal to the design principles of MAST, though some of their ideas may be applicable in the cluster-level placement algorithm of MAST.

Scheduling across clusters. Among existing systems, Yugong [20] and Hydra [10] are the closest to MAST as they can schedule jobs across cluster or datacenters, but they still differ from MAST since 1) they perform early-binding at cluster scope (§4.1), 2) they don't take GPU into consideration (§1 and §3), and 3) they rely on simple heuristics to dispatch jobs to clusters. Finally, in the data warehouse hierarchy (hundreds of namespaces→millions of tables→ billions of data partitions), Yugong places data at the namespace level (called "projects" in the paper), whereas Tetris places data at the partition level, which provides more opportunities for fine-grained optimization. However, Tetris' fine-grained placement makes the optimization problem about 10^6 times larger.

Singularity [43] is the only ML-specific global-scale scheduler we are aware of. However, the article does not disclose details about the scheduling part, but focuses more on how to provide elasticity to ML training jobs, and thus it's impossible for us to provide a concrete comparison.

8 Conclusion

This paper demonstrates that by utilizing the three design principles of *temporal decoupling*, *scope decoupling*, and *exhaustive search*, we can build a global ML training scheduler that can 1) scale to tens of regions and hundreds of thousands of machines and 2) provide high-quality data and job placement to achieve almost 100% allocation of GPUs.

Acknowledgments

This paper presents years of work by past and current members of several teams at Meta, including MAST, Tetris, and Twine [44]. In particular, we would like to call out some current team members not on the author list: Mike Begic, Rick Chang, Cheng Cheng, Fuat Geleri, Ankit Gureja, Christian Guirguis, Hamid Jahanjou, Johan Jatko, Jonathan Kaldor, Norbert Koscielniak, Alexander Kramarov, Mengda Liu, Running Lu, Duy Nguyen, Ivan Obraztsov, Colin Ovens, Marcin Pawlowski, Nader Riad, Derek Shao, Ahmed Sharif, Mihails Smolins, Bhushan Sonawane, Peeyush Taneja, Yingjie Tang, Hai Dang Tran, Tom Wan, Anthony Wang, Runmin Wang, Wenbang Wang, and Lukasz Wesolowski. We thank all reviewers, and especially our shepherd, Ryan Huang, for their insightful comments.

References

[1] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 348–363, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[3] AWS Regions, 2023. https://aws.amazon.com/about-aws/global-infrastructure/.

[4] Azure Regions, 2023. https://azure.microsoft.com/en-us/explore/global-infrastructure/products-by-region/.

[5] Yixin Bao, Yanghua Peng, and Chuan Wu. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 505–513, 2019.

[6] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 495–503, 2018.

[7] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. Online Evolutionary Batch Size Orchestration for Scheduling Deep Learning Workloads in GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.

[9] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.

[11] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[12] FICO Xpress Optimization. https://www.fico.com/en/products/fico-xpress-optimization.

[13] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A Novel Deadline-Aware Scheduler for Deep Learning Training Jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 609–623, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[15] Google Cloud Regions, 2023. https://cloud.google.com/blog/products/infrastructure/introducing-new-google-cloud-regions.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, aug 2014.

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[18] Gurobi Optimization. https://www.gurobi.com/.

[19] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[20] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-Distributed Data and Job Placement at Scale. *Proc. VLDB Endow.*, 12(12):2155–2169, aug 2019.

[21] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739. USENIX Association, April 2021.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[23] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[25] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 463–479, 2020.

[26] Donald L Kreher and Douglas R Stinson. Combinatorial algorithms: generation, enumeration, and search. *ACM SIGACT News*, 30(1):33–35, 1999.

[27] Kubernetes, 2020. https://kubernetes.io/.

[28] Kubernetes Federation, 2020. https://github.com/kubernetes/community/tree/master/sig-multicluster.

[29] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, 2021.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[31] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association.

[32] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's Service Management Platform for Stream Processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE, 2020.

[33] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association.

[34] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 521–537, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[36] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James

Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[37] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[39] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[41] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient Queue Management for Cluster Scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[42] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019.

[43] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[44] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.

[45] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-Efficient Distributed Deep Learning: A Comprehensive Survey. *arXiv preprint arXiv:2003.06307*, 2020.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in neural information processing systems*, 30, 2017.

[47] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[48] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[49] Shaoqi Wang, Oscar J. Gonzalez, Xiaobo Zhou, Thomas Williams, Brian D. Friedman, Martin Havemann, and Thomas Woo. An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[50] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM symposium on cloud computing*, pages 246–258, 2019.

[51] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages

945–960, Renton, WA, April 2022. USENIX Association.

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[53] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[54] Peifeng Yu and Mosharaf Chowdhury. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems*, volume 2, pages 98–111, 2020.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

[56] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *Proc. VLDB Endow.*, 7(13):1393–1404, aug 2014.

[57] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[58] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Kevin Wilfong, Sundaram Narayanan, Jack Langman, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1042–1057, 2022.

[59] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 331–347, New York, NY, USA, 2023. Association for Computing Machinery.

Automatically Reasoning About How Systems Code Uses the CPU Cache

Rishabh Iyer, Katerina Argyraki, George Candea
EPFL, Switzerland

Abstract

We present a technique, called CFAR, that developers can use to reason precisely about how their code, as well as third-party code, uses the CPU cache. Given a piece of systems code *P*, CFAR employs program analysis and binary instrumentation to automatically "distill" how *P* accesses memory, and uses "projectors" on top of the extracted distillates to answer specific questions about *P*'s cache usage. CFAR comes with three example projectors that report (1) how *P*'s cache footprint scales across unseen inputs; (2) the cache hits and misses incurred by *P* for each class of inputs; and (3) potential vulnerabilities in cryptographic code caused by secret-dependent cache-access patterns.

We implemented CFAR in an eponymous tool with which we analyze a performance-critical subset of four TCP stacks—two versions of the Linux stack, a stack used by the IX kernel-bypass OS, and the lwIP TCP stack for embedded systems—as well as 7 algorithm implementations from the OpenSSL cryptographic library, all 51 system calls of the Hyperkernel, and 2 hash-table implementations. We show how CFAR enables developers to not only identify performance bugs and security vulnerabilities in their own code but also understand the performance impact of incorporating third-party code into their systems without doing elaborate benchmarking.

CFAR is open-source and freely available at [58].

1 Introduction

System performance is important yet often poorly understood. Hence the recently proposed notion of a *performance interface* [30, 31, 47], defined by analogy to semantic interfaces (e.g., abstract classes, specifications, documentation) that have been used for many decades to succinctly describe a program's functionality. A performance interface describes a system's performance behavior in a manner that is simultaneously succinct, precise, and human-readable. The goal of performance interfaces is to help developers efficiently reason about the performance behavior of both their own and third-party code without having to delve into the code's implementation details—just like semantic interfaces help developers reason about functionality today.

Low-level systems code (e.g., operating system kernels, device drivers, network stacks, hypervisors) is special, because performance often critically depends on how the code interacts with the underlying micro-architecture. As a result, system developers spend a lot of time trying to understand this interaction, e.g., trying to understand whether the code's memory-access patterns are cache-friendly [2, 12–14, 50, 70] or whether the code's working set fits in cache [19, 23, 45, 67, 68, 77]. *Not* understanding this interaction can lead to performance bugs that are hard to diagnose, and can also result in unexpected performance behavior when using third-party code. For instance, a recent patch showed how the fast path of the Linux TCP stack had been experiencing a bloated cache footprint for over a decade, incurring slowdowns of up to 45% [42]; prior work has shown that applications may run up to 4× slower after calling into third-party code (e.g., a syscall) due to the callee's micro-architectural footprint [66, 74].

The goal of this work is to help system developers answer key questions about how their code and third-party code interacts with the underlying micro-architecture. We focus on interactions with the CPU caches (both data and instruction caches), since these often play a critical role in the performance of systems code [12–14, 19, 23, 38, 45, 57, 67, 68, 70, 77, 78]. We seek to answer frequently-asked questions about cache usage such as: "How does the code's cache usage scale as a function of the workload?" [6, 19, 23, 67, 68] and "Which workloads make the code's working set exceed the cache size?" [38, 57] without requiring developers to delve into the code's details or run elaborate, time-consuming benchmarks.

Answering the above questions requires visibility into how the code processes an *abstract* workload, so we look for abstractions that capture (in a succinct, precise, and human-readable manner) how the code interacts with the caches as a function of the workload. Our approach is in contrast to existing performance-analysis tools like profilers [10, 43, 59, 69] and cycle-accurate simulators [7, 9]: such tools can only provide insights into cache usage for the *concrete* workloads with which the code is profiled or simulated; they cannot provide visibility into how the code would behave for arbitrary, previously unseen workloads. As a result, when using these tools, developers are forced to manually reverse-engineer the answer to their questions. This process is both time-consuming and error-prone [29], particularly for code that the developers did not write themselves.

We present Cache Footprint AnalyzeR (CFAR), a technique for processing a piece of systems code into answers to developers' questions about how that code uses the cache. This processing consists of two phases: In the first phase, CFAR takes as input the target code and extracts from it an abstract representation (a "distillate") of how the code accesses memory. In the second phase, CFAR uses simple programs ("projectors") to transform the distillate into answers to specific questions about the code's cache usage. Since the distillate is a precise abstraction of the code's memory usage (i.e., it contains all the information relevant to how the code accesses memory), developers can use projectors to answer diverse questions about the code's cache usage. The eponymous tool that implements the CFAR technique relies on a combination of static analysis, symbolic execution, and binary instrumentation to automatically extract distillates. We chose these particular program-analysis techniques because, despite their scalability limitations (discussed in §4.3), they enable precisely the level of visibility a developer seeks, enabling her to reason about how the code processes an abstract workload.

The current CFAR prototype comes with three projectors that answer frequently-asked questions about cache usage: (1) $\mathcal{P}_{\text{scale}}$ computes how the amount of data the code brings into the cache (in bytes) varies as a function of the workload; (2) $\mathcal{P}_{\text{h/m}}$ computes, as a function of workload, whether memory accesses will hit or miss in the cache; and (3) $\mathcal{P}_{\text{crypt}}$ flags cryptographic code that branches on, or accesses memory in a way that depends on secret inputs, thereby flagging potential security vulnerabilities or proving their absence. $\mathcal{P}_{\text{crypt}}$ is an example that demonstrates the flexibility of CFAR's two-phased process: since the distillate contains all information relevant to how the code accesses memory, developers can write projectors to analyze more than just standard performance properties. We envision developers contributing more such projectors to the CFAR tool, making it more useful over time. In stable state, developers will likely just use whatever ships with CFAR, extending it only when they cannot get the answers they seek.

We use CFAR to analyze a performance-critical subset of the transport layer of 4 TCP stacks—2 versions of Linux's stack (i.e., before and after the recent reorganization for cache efficiency [42]), a TCP stack used by the IX kernel-bypass OS [6], and the lwIP TCP stack for embedded systems [20]—as well as 2 hash-table implementations [60, 73], all 51 of the Hyperkernel's system calls [51], and 7 algorithm implementations from the OpenSSL cryptographic library [54]. We use the results to demonstrate how distillates and projectors enable developers to understand the cache usage of both their own and others' code, for unseen workloads, without running elaborate benchmark suites. As part of the evaluation, we also uncover a cache-inefficient data layout in the kernel-bypass TCP stack, an error path in the Hyperkernel `mmap()` system call (which, despite looking innocuous, inadvertently pollutes 40% of the L1 d-cache), and a constant-time violation in OpenSSL 3.0's implementation of AES. For all the above code, CFAR's analysis completes in minutes, which means that extraction and analysis of distillates can be feasibly integrated into a real-world software-development cycle.

The rest of this paper is organized as follows: We first motivate CFAR using examples of cache-usage questions that existing tools cannot answer (§2), then provide an overview of the CFAR approach (§3) and detail its design (§4). We then evaluate the CFAR prototype experimentally (§5), discuss related work (§6), and conclude (§7).

2 Motivation

In this section, we give an example of the kind of questions that system developers ask about their code's cache usage (§2.1), and then describe why existing tools cannot answer such questions (§2.2).

2.1 Example

Consider a developer, Alice, who is building an in-memory key-value store that has to be fast. The key-value store uses a hash table to store the key-value pairs and runs atop a user-space, kernel-bypass transport stack. Alice has modified an existing hash-table implementation to suit her needs, and thus understands that part of the code well. However, she is using an off-the-shelf transport stack [20, 34, 75], of which she understands little beyond the semantic interface it exposes.

In such a system, throughput is often bottlenecked by the number of last-level cache (LLC) misses per request [41, 67, 79]; hence, to optimize throughput, Alice needs to know how the different parts of her code use the cache and how they affect the LLC misses as a function of the workload. For example, if her system fails to reach the expected throughput due to persistent LLC misses, what is the predominant cause? Is it that the hash table code touches too many cache lines per `put()` or `get()` request? Or is it that the transport stack's buffer-management code touches too many cache lines per connection [6]? In the former case, Alice should spend her time optimizing the memory layout of the hash table [12–14], whereas, in the latter, she should port her code to alternative stacks with smaller memory footprints [20, 67]. Finally, if both codebases were already highly optimized, she should avoid wasting time on code optimizations and replicate her key-value store across multiple machines [4].

2.2 Existing Tools Are Insufficient

Existing tools like profilers [10, 43, 59, 69] and cycle-accurate simulators [7, 9] are fundamentally ill-suited to answering Alice's questions. This is because profilers and simulators are designed to reason about what the code does to the micro-architecture *for a given* workload, whereas answering Alice's questions requires reasoning about what the code does to the micro-architecture *as a function of* workload. So, while profilers and simulators can provide visibility into the code's cache usage for a given workload, they do not have *predictive power*, and thus cannot provide Alice with visibility into cache usage for workloads beyond the ones that she herself provided to the tools.

As a result, developers like Alice are forced to guess the answers to their questions based on (incomplete) information derived from profiling. Between cycle-accurate simulators and profilers, performance engineers typically prefer profilers because they are orders of magnitude faster, even if less precise. So, Alice would typically profile her system with many workloads to measure micro-architectural events and then guess the predominant cause of LLC misses. In particular, she would try to identify the properties of workloads that led to low throughput: were they those that led to a large number of `put()` or `get()` accesses per request? Or those that led to a large number of concurrent connections? This is similar to what developers in industry do to answer such questions: they run their code for multiple workloads, use profilers to count the total number of unique cache lines touched, and then manually extrapolate how workload affects their code's cache footprint [5, 42].

Reasoning about cache usage in such a manner is not only time consuming but also error prone, particularly for third-party code. For instance, Alice (who knows little about the implementation of the transport stack) may not even think about running workloads that lead to different numbers of concurrent connections. In general, performance profiling suffers from the "large input problem" [48, 53]: unexpected performance behavior often manifests only when input size (e.g., the number of concurrent connections) exceeds some threshold that may seem arbitrary to those who are not intimately familiar with the code. So, designing a test suite that completely "covers" a system's performance behaviors is hard, and developers do not even have metrics for performance coverage. While line coverage is used as a proxy for what fraction of the code's semantic behaviors is covered by tests, performance profiling does not have even such an imperfect metric.

As a result, developers like Alice often fail to identify workload properties that significantly impact cache usage, causing performance cliffs to manifest in production. For instance, developers from Google recently showed that the fast path of the Linux TCP stack had been accessing 50% more cache lines than it needed to, for over a decade, which was leading to performance degradation of up to 45% [42]. Similarly, initial work on predicting the working set of network functions ignored the impact of different packet sizes [19], and a study of Linux's system-call performance showed how a newly introduced configuration parameter can destroy spatial locality and lead to increased LLC misses [62]. In practice, developers like Alice often use incomplete performance profiling to guesstimate their system's cache usage, and then they overprovision resources for their system, to mitigate unexpected performance degradation [24]. This leads to lower system efficiency and inflated costs, and is not always effective.

Summary. Existing tools like profilers and cycle-accurate simulators are ill-suited to answering frequently-asked questions about cache usage, because they do not have *predictive power* across the space of possible workloads. As a result, developers are forced to estimate the answers to their questions using incomplete information obtained via profiling. This process is not only time consuming but also error prone, particularly when applied to third-party code.

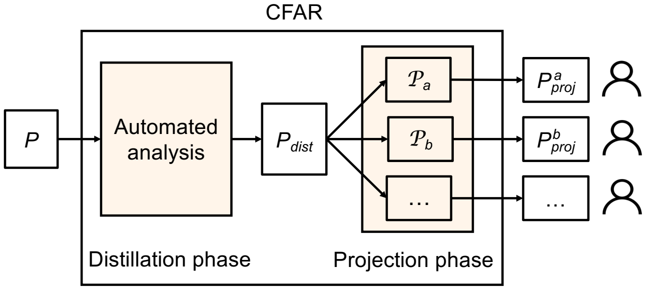


Figure 1: The CFAR workflow. P denotes a unit of systems code, P_{dist} denotes the corresponding distillate, $\mathcal{P}_i$ denotes the different projectors, and P^i_{proj} denotes the corresponding projections that provide answers to developers' questions about P's cache usage.

3 CFAR Overview

Answering questions about cache usage requires reasoning about the code. We therefore look for abstractions that precisely capture what the code does to memory (and thus the cache) as a function of its input (workload).

With this in mind, we propose two abstractions: *distillates* and *projections*. Let P be any well-defined part of a system that can be invoked individually, such as a system call in an OS kernel or a function in a library, or even a standalone program. A *distillate* P_{dist} is a program that specifies precisely and completely how P accesses memory. A *projection* P^{π}_{proj} is a much simpler program that answers a specific question π about P's cache usage. For any given P, there exists a unique distillate P_{dist}, but there can be as many projections as there are questions about P's cache usage.

We represent distillates and projections as programs—as opposed to denser, more mathematical representations (e.g., [22])—for two reasons. First, programs provide developers with a representation that they are familiar with, allowing them to quickly read and understand the answers to questions about cache usage. Second, programs can be executed, which makes it possible for tools to leverage distillates and projections for automated performance analysis; in §4 we show how CFAR executes a distillate against a cache model to reason about cache hits and misses.

Fig. 1 illustrates CFAR's workflow, which consists of two phases: The first phase takes as input a module P of a system's code and automatically extracts P's distillate. The second phase relies on simple programs ("projectors") that transform the distillate into projections that answer specific questions about P's cache usage, such as "How many unique cache lines does P touch as a function of the workload?" and "How does P's cache hit/miss profile vary as a function of the workload?" CFAR currently provides three such projectors, and we envision developers contributing more over time.

The two-phased workflow provides CFAR with the flexibility needed to answer diverse questions about cache usage. Since the distillate captures *all* information relevant to how the code accesses memory, it can always be transformed—

```
1  int sys_create(int fd, fn_t fn, uint64
       ftype, uint64 value, uint64 omode) {
2
3      if (ftype == FD_NONE)
4          return -EINVAL;
5      if (!is_fd_valid(fd))
6          return -EBADF;
7      if (&proc_tbl[pid]->ofile[fd] != 0)
8          return -EINVAL;
9      if (!is_fn_valid(fn))
10         return -EINVAL;
11
12     struct file = get_file(fn);
13     if (file->refcnt != 0)
14         return -EINVAL;
15     file->type = ftype;
16     file->value = value;
17     file->omode = omode;
18     file->refcnt = file->offset = 0;
19     set_fd(pid, fd, fn);
20     return 0;
21 }
```

```
1  def sys_create_dcache(fd, fn, ftype, value, omode):
2      # State: pid, proc_tbl, file_tbl
3
4      if ftype == FD_NONE: #6 accesses
5        return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]
6
7      if not(fd >=0 and fd < NOFILE): #6 accesses
8          return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]
9
10     if [proc_tbl+256*pid+64+8*fd]: #7 accesses
11         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
12
13     if not(fn >=0 and fn < NOFILE): #7 accesses
14         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
15
16     if [file_tbl+40*fn+8]: #9 accesses
17         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)..,(r,file_tbl+40*
       fn+8),..,(r,rsp-8)]
18
19     # Succesful create. 17 accesses
20     return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,(r,file_tbl+40*fn
       +8),(w,file_tbl+40*fn),(w,file_tbl+40*fn+16),..,(w,proc_tbl+256*pid+64+8*fd)
       ,..,(r,rsp-8)]
```

Figure 2: Example program on left (Hyperkernel `sys_create` system call that creates a new file) and the corresponding data-accesses distillate.

```
def sys_create_icache(fd, fn, ftype, value, omode):
    # State: pid, proc_tbl, file_tbl
    # sys_create abbreviated as s

    if ftype == FD_NONE: # 10 instructions
      return [(r,s),..,(r,s+168),..,(r,s+176)]

    # Error paths elided for presentation clarity
    ......

    # Succesful create. 45 instructions
    return [(r,s),(r,s+8),..,(r,s+160),(r,s+168),(r,s+176)]
```

Figure 3: Instruction-accesses distillate for `sys_create`.

using a suitable projector—into a projection that answers a specific question about the code's cache usage. We demonstrate this flexibility by building a projector ($\mathcal{P}_{\text{crypt}}$) that goes beyond standard performance analysis and uses the distillate to identify potential cache-based security vulnerabilities.

CFAR does not make any assumptions about the kind of code that it takes as input. That said, in this work, we focus on systems code (e.g., operating systems, device drivers, network stacks, hypervisors), since it is code for which cache usage has a significant impact on performance.

We now define the three key components of CFAR, namely distillates (§3.1), projectors (§3.2), and projections (§3.3). Table 1 summarizes these definitions.

Notation	Description
P	A unit of code that can be invoked individually (e.g., system call, library function, standalone program). It takes as input I and has initial state S_0.
Ω	An ordered sequence of memory accesses (to symbolic addresses).
$\mathcal{P}_\pi$	A *projector*. It is a program that defines a function/property $\pi(\Omega)$ related to cache usage.
P_{dist}	The unique *distillate* of P. It is a program that takes as input I and computes Ω as a function of I and S_0, where Ω is P's memory-access sequence.
P^{π}_{proj}	A *projection* of P. It is a program that takes as input I and computes $\pi(\Omega)$ as a function of I and S_0, where Ω is P's memory-access sequence and $\pi(\Omega)$ is defined by a projector $\mathcal{P}_\pi$.

Table 1: Glossary.

3.1 Distillates

Consider a program (or function, or method) P, with input(s) I, and state S_0 at the time of invocation. S_0 consists of the values of P's objects in the heap and the stack up to `%rsp`.

P's *distillate* P_{dist} is another (typically simpler) program that takes the same input(s) I, and computes P's sequence of memory accesses Ω as a function of I and S_0. Since accessing data vs. instructions exhibits distinct patterns, we distinguish between a data-accesses distillate P^{data}_{dist} and an instruction-accesses distillate P^{instr}_{dist}. The former computes the sequence of data-memory accesses that would be observed if executing P with input I starting from state S_0, while P^{instr}_{dist} computes the corresponding instruction-memory accesses.

We illustrate what a distillate looks like with the example of the `sys_create` system call (Fig. 2, left) in the Hyperkernel [51]. First, each memory access in Ω is a tuple $<type, addr>$, where *type* can be read (`r`), write (`w`) or read-modify-write (`rmw`), and *addr* is a memory address. In a data-accesses distillate (Fig. 2, right), each memory address is a function of standard state components (e.g., the stack pointer `%rsp`), as well as components that are specific to P. For example, line 11 in the distillate describes accesses that are a function of `proc_tbl`, `pid`, and `fd`, which arise from executing line 7 in `sys_create`. If a memory address is independent of I and S_0 (e.g., the address of a `struct` allocated by P in the heap and then freed before returning), it is represented as a named constant (e.g., `mallocRetVal@file.c:342`). In an instruction-accesses distillate (Fig. 3), each memory address is represented as an aligned offset relative to the address of the first instruction in P. In our particular example, the compiler inlines all helper functions, hence there is only one base address `s`.

The distillate P_{dist} is a precise and complete representation of P's memory usage. It is *precise* because it provides the exact sequence of memory accesses for any execution of P. The symbolic expressions for data- and instruction-memory accesses as a function of I and S_0 are precise by construction, and therefore correct for any concrete instantiation of I and S_0. The distillate is *complete* in that it contains all information on P's memory accesses that can be found in P. No matter what the concrete values of I and S_0, how the address space is randomized [1], or where in memory the code is loaded, P_{dist} will always be able to produce the exact sequence of memory accesses that P makes when executing from S_0 with input I.

3.2 Projectors

A *projector* $\mathcal{P}_\pi$ is a program that defines a function $\pi(\Omega,\ldots)$ related to cache usage. For example, a projector may define $\pi(\Omega) = |\Omega|$, i.e., the number of memory accesses in Ω, while another projector may define $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$, i.e., the number of unique 64-byte cache lines accessed by Ω.

We think of a function π as a question about cache usage (e.g., "How many memory accesses does this piece of code perform?" or "How many unique cache lines does the code access?"). CFAR enables developers to write their own projectors, such that they can formulate their own questions.

A key property of projectors is that they are *code-agnostic*: A function $\pi(\Omega, \ldots)$ defined by projector $\mathcal{P}_\pi$ is independent of the semantics of the code that produced Ω (or any of the other arguments). This code-agnostic nature makes projectors easy to express; for example, a developer can write the simple projector that defines $\pi(\Omega) = |\Omega|$ (mentioned above) to query the number of memory accesses performed by *any* P, without having to understand P's details.

The function π may take inputs beyond just Ω. For example, it may take as additional input a cache model, and compute the number of hits and misses incurred by Ω in the L1 data cache given that particular cache model. Since Ω is independent of where and how the code that produced it was executed, such a generalized function π can precisely characterize the impact of running a piece of code on different micro-architectures and/or with different OS configurations (e.g., for different memory-page sizes that P may use).

3.3 Projections

A *projection* P^{π}_{proj} of P is another (typically simpler) program that takes the same input(s) I as P, and computes the value of function π for P, as a function of I and S_0. Said differently, if we think of π as a specific question about cache usage, then P^{π}_{proj} is a program that provides the answer to that question for P. Fig. 4 shows a projection of `sys_create` that computes the number of data memory accesses performed by the system call as a function of its input and the OS state. CFAR produces P^{π}_{proj} by applying the projector $\mathcal{P}_\pi$ to P's distillate P_{dist}.

```
def sys_create_dcache_num_accesses(fd, fn, ftype, value, omode):
    # State: pid, proc_tbl, file_tbl

    if ftype == FD_NONE:
      return 6

    if not(fd >=0 and fd < NOFILE):
        return 6

    if [proc_tbl+256*pid+64+8*fd]:
        return 7

    if not(fn >=0 and fn < NOFILE):
        return 7

    if [file_tbl+40*fn+8]:
        return 9

    # Succesful create.
    return 17
```

Figure 4: Projection of `sys_create` that describes the number of data memory accesses.

Summary. CFAR provides abstractions to reason about what a program does to the memory hierarchy. If a program is a function P: $<I, S_o> \rightarrow$ semantic outputs, we say that the distillate is a function P_{dist}: $<I, S_o> \rightarrow \Omega$ that abstracts away everything that has to do with program semantics and preserves information about memory accesses. The projection is a function P^{π}_{proj}: $<I, S_o> \rightarrow \pi(\Omega)$ that computes an answer and abstracts away all unrelated information. This is a progression of abstraction steps, starting from the original program and arriving at the final projection. Each step takes the same arguments $<I, S_o>$, but returns a result that is increasingly more focused on the cache-usage question at hand.

4 CFAR Design

We now provide more details on the two phases of CFAR: distillation (§4.1) and projection (§4.2).

4.1 Phase #1: Distillation

CFAR automatically distills an input program P into its corresponding distillate P_{dist} using a four-step process, shown in Fig. 5: it ① enumerates all feasible executions paths in P using automated program analysis; then ② obtains a binary execution trace for each such path; then ③ based on the results of these two steps, prepares an execution tree for the distillate; and lastly ④ optimizes this tree and produces P_{dist}.

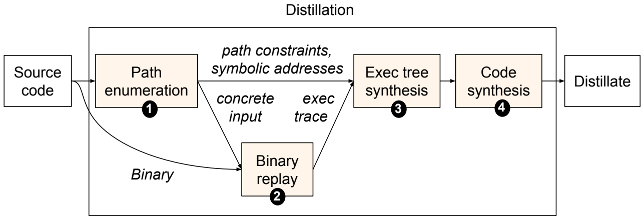


Figure 5: The four steps in CFAR's distillation process.

Our use of context-sensitive program analysis and binary replay ensures that CFAR can extract a precise distillate without requiring any effort on the part of the developer, but this also imposes limitations. Most notably, CFAR is subject to the scalability limitations of such program analysis and is thus not ideal for reasoning about multi-threaded code or about code with loops whose bounds cannot be statically computed. Additionally, CFAR is also limited by the proprietary nature of modern hardware. For instance, since the exact algorithms used to schedule instructions in an out-of-order processor are not publicly available, CFAR cannot reason about speculative memory accesses. We describe CFAR's limitations in detail in §4.3, but we note here that, despite these limitations, CFAR is able to provide valuable information about cache usage that is otherwise unavailable, and do so for a wide variety of systems code (§5).

4.1.1 Step ①: Path enumeration

To enumerate all the paths in P, CFAR uses *exhaustive* symbolic execution [11, 26, 36, 65]. This is a context-sensitive program-analysis technique that automatically traverses the feasible execution paths of a body of code, enabling a comprehensive analysis of its control flow. The technique is powerful but also faces challenges related to loops and pointers, which we discuss in §4.3. We use an exhaustive form of this technique, which yields *all* feasible paths in P.

For each enumerated path α, CFAR saves four key pieces of information: (1) the precise path constraint C_α that uniquely defines this path, i.e., the conjunction of the predicates of each conditional along α; (2) a concrete input I_α that exercises this path, obtained by asking an SMT constraint solver [18] for a satisfying assignment to the free variables in C_α; (3) the symbolic expression corresponding to the address of each data/instruction memory location accessed along the path, expressed as a function of P's inputs and/or state; and (4) a corresponding `file:linenum` identifier for each memory operation, to be used later. The sequence of these symbolic expressions is ω_α.

Our CFAR prototype uses KLEE [11] to perform the symbolic execution. KLEE, like many other symbolic-execution engines, analyzes the code at the IR level, which in KLEE's case is LLVM [16]. CFAR can therefore handle any code that is compiled to LLVM and can be handled by KLEE. Our KLEE modifications and additions total ~1,500 lines of C++.

4.1.2 Step ②: Binary replay

What actually executes on the hardware is not the source code or the IR. Compiler optimizations, such as link-time optimization, cause the executing machine code to not directly correspond to what is in the IR. Furthermore, many IRs are Static Single Assignment (SSA), in which each variable is assigned exactly once. This makes the data flow and dependencies among variables more explicit and easier for the compiler to analyze, but also implies an infinite register file. Processors do not have infinite register files so, during an actual execution, register values often need to be spilled to the stack. Since these pushes and pops to/from the stack are not present in an SSA IR, and thus not captured when analyzing P in its IR form, the corresponding memory accesses will not appear in ω_α.

Therefore, CFAR replays an *instrumented* version of the *P binary* for each I_α, to obtain a corresponding execution trace X_α for each path α in P. For each machine instruction executed in X_α, CFAR saves the program counter, the instruction opcode (e.g., `mov`, `push`, `pop`), the concrete memory addresses accessed, and the corresponding `file:linenum` debug information inserted into the binary by the instrumentation.

We deliberately split the CFAR analysis into a source-based and a binary-based step. On the one hand, it is easier to extract symbolic expressions for memory operations by analyzing the source code or the IR. On the other hand, analyzing the binary enables CFAR to be fully precise with respect to compiler optimizations and which instructions lead to memory accesses and do not merely manipulate CPU registers. In theory, these two steps could be combined into a single one by directly executing the binary symbolically (e.g., with S2E [15]). To answer with certainty whether this is possible, one would need to assess how CFAR is affected by the loss of type information when going from source code to binaries.

The CFAR prototype uses Intel PIN [46] to instrument binaries. Our Pintool consists of ~350 lines of C++.

4.1.3 Step ③: Execution tree synthesis for P_{dist}

In this step, CFAR combines the information extracted in the previous two steps. For each path α in P, it combines the symbolic memory trace ω_α with the corresponding binary execution trace X_α. To produce a *data*-access trace, CFAR takes the sequence of concrete addresses from X_α and replaces (using debug information) all input- and state-dependent accesses with the corresponding symbolic expressions from ω_α, resulting in Ω_α^{data}. To produce an *instruction*-access trace, CFAR uses the program-counter values and the call stack in X_α to compute the symbolic offset of each instruction from the start of P (e.g., from its entry point, if P is a function or a system call) to produce Ω_α^{instr}. The call stack gives CFAR information on which function the instruction belongs to, so that it can compute the function-specific offset.

Next, CFAR assembles an execution tree using the path constraints C_α. It arranges all the paths into a tree based on their common prefixes; for every path α there exists a path from root to leaf in the tree, and vice-versa. Each internal node of the tree contains the predicate corresponding to the original branch in P. The conjunction of the predicates for all internal nodes along a root-to-leaf path forms the corresponding path constraint C_α.

4.1.4 Step ④: Synthesis of the P_{dist} distillate

The final step in CFAR's distillation process consists of summarizing loop-related memory access patterns, along with other improvements for human readability of P_{dist}. Symbolic execution, by default, unrolls loops and thus produces a separate execution path for each loop iteration. This leads to bloated distillates that are hard to read and contain redundant information, particularly if the code's memory-access pattern does not change meaningfully across loop iterations.

Summarizing a loop entails representing the effects of that loop without representing all its iterations explicitly. Conceptually, the goal of this step is to eliminate from the execution tree the subtrees induced by loop unrolling. This step does not elide or lose any information contained in the distillate—it only optimizes the distillate's control flow for human readability. By definition, any projection derived from P_{dist} will retain P's control flow, as reflected in the distillate.

While automatically summarizing loops in general is undecidable [25], studies have shown that there exist four common categories of loops that relate to data locality issues in systems code [35]. Therefore, CFAR contains loop-summary templates for these four categories of loops—two that traverse array-like data structures, and two that traverse pointer-chasing data structures (e.g., linked lists, trees). All four categories require the loop body to not branch on the specific value of the iteration counter, and require the loop to have a maximum of two termination predicates, one in the loop definition and at most one `break` in the body. Each loop template has a corresponding loop summary.

For loops that do not match a template, the distillate presents them in unrolled form—still correct, just less human-readable. We thus call this "best-effort" loop summarization.

After loop summarization and a few other optimizations for readability, CFAR transforms the tree into a program that represents P_{dist}. This program takes the same input as P. Every internal tree node generates an `if` statement in the program, branching on the predicate contained in that node. Each path through the program ends with a `return` of the corresponding Ω_α. Depending on the memory-type of the distillate, the returned symbolic memory-access trace is either Ω_α^{instr} or Ω_α^{data}.

Fig. 6 shows a snippet for the P_{dist}^{data} distillate of the `memcmp` function in the C standard library. Our CFAR prototype uses Python to represent distillates, because it is one of the most widely used languages [52] and has an easy-to-understand syntax. The example illustrates CFAR's loop summarization for a loop that belongs to the first category of loops mentioned above. CFAR uses first-order logic to summarize loops with primitives from Z3's Python API [72]. The predicate that starts on line 3 identifies the smallest index `i` (bounded by `len`) at which the two strings differ. The distillate shows that the memory accessed by `memcmp` corresponds to every element of the two arrays up to `i`.

```
1 def memcmp_dcache(s1,s2,len):
2
3     if Exists(i,And(0<=i<len,[s1+i]!=[s2+i],
4                     ForAll(j, Implies(0<=j<i),[s1+j]==[s2+j]))):
5
6       return ForAll(k, Implies(0<=k<=i),[(r,s1+k),(r,s2+k)])
7     return ForAll(k, Implies(0<=k<=len),[(r,s1+k),(r,s2+k)])
```

Figure 6: P_{dist}^{data} for `memcmp` showing CFAR's loop summarization.

4.2 Phase #2: Projection

The distillate produced by the previous phase contains *all* the information on P's memory-access behavior. The answer to a developers' cache-usage question can therefore be found in the distillate, but it is buried among details that may not be relevant to that question. The projection phase turns distillates into focused, actionable answers that are not clouded by details irrelevant to the question being asked.

4.2.1 General overview

Developers write projectors in the form of programs that take as input a tuple $<\Omega, C>$, along with possibly other projector-specific parameters, such as the cache model mentioned in §3.2. Ω is a symbolic memory-access trace, and C is a constraint on the variables that appear in the symbolic addresses of Ω, in the form of a first-order logic expression.

We expect most projectors to ignore C and implement a function of just Ω, like $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$. We therefore did not mention the C parameter in §3.2, for clarity of presentation, but, to answer all cache-usage questions, the more general function $\pi(\Omega, C)$ is sometimes needed. For example, determining if all memory accesses are cache-line-aligned requires, in the general case, both Ω and C. The origin of this constraint C is the path constraint that causes the original program to execute the memory accesses in Ω. The branch predicates in distillates are such constraints. The constraint can be simple, like `0≤idx<128`, or more sophisticated, stating for example that the value stored at a particular memory location is non-zero: `[file_tbl+40*fn+8]`$\neq 0$. In our prototype, Ω and C are Python lists of Z3 expressions [72].

To produce a projection, CFAR takes a projector program $\mathcal{P}_\pi$ and a distillate P_{dist}, and synthesizes a new program P_{proj}^{π}. For each branch in P_{dist}, the P_{proj}^{π} program has the same branch as P_{dist} but, instead of returning Ω (as the distillate does), it returns the value of invoking $\mathcal{P}_\pi\ (\Omega, C, \ldots)$. In other words, the projection P_{proj}^{π} has the same control flow as the distillate but, instead of calculating a memory-access trace, it calculates a specific cache-usage property of that trace.

4.2.2 Example projectors: $\mathcal{P}_{\text{scale}}$, $\mathcal{P}_{\text{h/m}}$, and $\mathcal{P}_{\text{crypt}}$

Our CFAR prototype comes with three example projectors: (1) $\mathcal{P}_{\text{scale}}$ computes how the cache footprint (in bytes) varies across an entire range of previously unseen inputs (e.g., how it scales with the number of active network connections); (2) $\mathcal{P}_{\text{h/m}}$ computes the cache hit and miss profiles per class of inputs, as opposed to per specific, concrete input; and (3) $\mathcal{P}_{\text{crypt}}$ flags cryptographic code that accesses the cache in a way that depends on secret inputs. This projector can be used to find potential security vulnerabilities or prove their absence.

While the questions answered by these projectors are non-trivial, the projectors themselves are straightforward to write. For example, we express the functionality of both $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{crypt}}$ in less than 100 lines of Python. While $\mathcal{P}_{\text{h/m}}$ requires ~800 lines, almost 600 of those are a Python translation of the cache model from the `gem5` cycle-accurate simulator [7]. In §5.2, we show how a simple 5-line projector helped us identify a performance bug in a TCP stack used by IX [6].

$\mathcal{P}_{\text{scale}}$ computes cache footprint based on the symbolic Ω for a given input class. $\mathcal{P}_{\text{scale}}$ first determines which addresses in Ω change if the value of the input changes. It then uses an SMT solver to check the alignment of these addresses and determine the number of unique bytes touched by the accesses to those addresses, and produces the result as a human-readable formula. For instance, applying $\mathcal{P}_{\text{scale}}$ to the `sys_create` distillate in Fig. 2, for the input class that corresponds to successful creation (line 20), yields the formula `8*fd + 32*fn`. This formula says that, in a sequence of successful `sys_create` calls, the cache footprint will increase by 8 bytes for each distinct `fd` argument value and by 32 bytes per distinct `fn` argument value. This is how `fd` and `fn` influence the cache footprint, and this is exactly the information that Alice wanted to know for keys and network connections in §2. $\mathcal{P}_{\text{scale}}$ can be used, for instance, to quickly determine when the code's working set will overflow the cache.

$\mathcal{P}_{\text{h/m}}$ is more sophisticated and takes four projector-specific parameters: a workload size W, an input set cardinality N, a probability mass function PMF, and a cache model. Its goal is to compute the number of hits and misses experienced by a workload of W inputs that can take on any of N distinct types, distributed within the workload according to the PMF, when using a cache that works according to the model. In the scenario we will explore in §5 for a TCP network stack, W would be the number of packets in a trace, and N the number of unique connections—what distinguishes packet types is which connection they belong to. The PMF would be the relative distribution of packets among the N connections.

$\mathcal{P}_{\text{h/m}}$ first produces a workload of W symbolic inputs satisfying the PMF. For example, if W=5 and N=2 and the PMF is $< 0.4, 0.6 >$, the workload would be $< \lambda_1, \lambda_2, \lambda_1, \lambda_2, \lambda_2 >$ or some other variant that satisfies the PMF. Then, for each symbolic input in the workload, $\mathcal{P}_{\text{h/m}}$ iterates through Ω and sends each memory access to the cache model; the access may be a function of the λ_i symbolic input, or independent of it. $\mathcal{P}_{\text{h/m}}$ records, for each access, whether it is a hit or a miss.

Since there are multiple workload variants that satisfy the PMF, the process above repeats, with alternate variants, until the resulting hits/misses counts are statistically significant.

$\mathcal{P}_{\text{h/m}}$ can be thought of as a symbolic, trace-based cache simulator. A major challenge is, when dealing with symbolic addresses, $\mathcal{P}_{\text{h/m}}$ cannot compute set-associativity conflicts precisely. Instead, $\mathcal{P}_{\text{h/m}}$ *approximates* them by allocating an unconstrained, symbolic memory address (typically the base of a data structure) randomly to a set in the cache, and then mapping all relative addresses as offsets from that base. For example, if the address γ is randomly mapped to set μ, then the address $\gamma + 64$ would be deterministically mapped to set $(\mu + 1)$ `mod` # of sets in the cache. In §5 we will see that this approximation works well when compared to real hardware.

By default, $\mathcal{P}_{\text{h/m}}$ provides a 3-level inclusive cache with a next-line prefetcher whose size and set associativity at each level is configurable. The default PMF is uniform, and W=100N. $\mathcal{P}_{\text{h/m}}$ assumes that the memory trace Ω does not update initial program state S_0 that influences addresses in Ω.

$\mathcal{P}_{\text{crypt}}$ is an example of a projector that also takes the path constraint C into account. Its goal is to answer the question of whether there are any data accesses to secret-dependent memory addresses or secret-dependent branches, both of which are known sources of side channels [3]. $\mathcal{P}_{\text{crypt}}$ takes, as a projector-specific input, a list of program inputs that constitute secrets. It then uses an SMT solver to determine which (if any) of the memory addresses in Ω are influenced by secrets. It then checks whether any of the secrets appear in the path constraint C. If it finds any, then $\mathcal{P}_{\text{crypt}}$ returns `file:linenum` debug information for the corresponding branch or memory access, as well as the path constraint that leads to it. If none found, then $\mathcal{P}_{\text{crypt}}$ states that the code does not have secret-dependent branch instructions or data accesses.

$\mathcal{P}_{\text{crypt}}$ cannot check for all types of cache-based leakage. For example, leakages due to speculatively executed instructions [44] are out of scope for $\mathcal{P}_{\text{crypt}}$.

4.3 Limitations and Assumptions

Scalability limitations of symbolic execution: CFAR's reliance on symbolic execution (SE) makes it subject to SE's own limitations. Depending on which SE engine is used, certain kinds of loops, or symbolic pointers, or multi-threading could prevent CFAR from obtaining all execution paths [8]. However, there is active research on this topic, and recent SE engines have brought various enhancements that overcome these challenges, such as state merging [39], loop-extended symbolic execution [64], loop summaries [27, 71], loop invariants [33], and symbolic abstract transformers [37].

Any CFAR prototype will ultimately inherit the power of its underlying SE engine. Since our prototype relies on KLEE [11], code whose loops do not have statically computable bounds, or that is multi-threaded, or that has arbitrary symbolic pointers is not an ideal match, because path exploration may take too long. This makes our current prototype a poor fit for analyzing entire systems, like the Memcached or Redis key-value stores. Nevertheless, we show in §5 that CFAR extracts useful distillates for key components of complex systems code (e.g., for data structures whose cache footprint is a common source of performance problems).

Using CFAR for code that is not amenable to exhaustive symbolic execution: CFAR's reliance on exhaustive symbolic execution means that automatically extracting complete distillates is not always feasible. We now discuss how developers can obtain useful results with CFAR even in such cases.

A simple approach is to *constrain the input space*, e.g., constrain CFAR to inputs that trigger the "fast path" through the code, since that is a common target of performance analysis. For instance, if the code of interest is an IP-packet forwarding function, it is reasonable to constrain the distillate to packets without IP options. This dramatically reduces both the size of the distillate and the time required to obtain it (since it eliminates the part of the code that loops through the variable-length IP options), while still yielding practically useful results (since performance-sensitive traffic typically does not carry IP options). Focusing on the cache usage of the fast path is common practice today; for instance, the recent reorganization of the Linux TCP stack was based entirely on the requirements of the TCP fast path [42].

For constraining the input space, CFAR provides an interface similar to KLEE's [11], with which developers can provide constraints on arbitrary program variables. In our evaluation, we use this approach to analyze code that is not amenable to exhaustive symbolic execution (e.g., the Linux TCP stack), and the results are compelling despite the constrained input space. Constraining the input space requires the developer to have some knowledge of, for instance, what typical/fast-path inputs look like. However, it does not require knowledge of the code's internal details, because these are explored automatically by CFAR.

An alternate approach is to run CFAR with a specified time budget. When the time limit is reached, CFAR outputs a partial distillate that returns the exact sequence of symbolic memory accesses performed by the code along the explored execution paths. While this approach does not require developer knowledge, the downside is that CFAR may not explore all meaningful execution paths in the given time budget. The CFAR prototype offers this time-budget feature, but we did not use it in any of the experiments in our evaluation.

Prototype cannot aggregate across input classes: In the current projection model (§4.2), a projector instance gets to see only the Ω corresponding to one input class. Our CFAR prototype does not yet support sharing state across different instances of projectors, and thus does not support aggregating measures across multiple input classes. This support is simple to add, we just have not encountered the need for it yet.

Cannot account for inter-process interactions: Projectors cannot answer questions that span multiple processes. Also, we assume that P is small enough to not have its execution interrupted by preemption. The distillate P_{dist} is always correct with respect to P, but the predictions made based on this distillate alone will miss cache accesses performed by code other than P during a preemption. In other words, if the period of interest includes a preemption, when a projector looks at Ω, it does not get the full picture, because P is not the only code that interacts with the micro-architecture.

Limitations due to proprietary hardware details: CFAR employs binary instrumentation to obtain an execution trace. Such instrumentation can only reveal instructions that the processor retires (i.e., finishes executing); it does not reveal instructions that were executed as a result of incorrect speculation, such as a mispredicted branch. Speculated instructions nevertheless could impact the cache, even if their semantic effects are undone. Since CFAR does not see those accesses, the answers computed by projectors may not be fully accurate. We are not aware of any tool that can precisely report such mis-speculated instructions during an execution, since the scheduling algorithms used in the out-of-order pipelines of commercial processors are proprietary.

5 Evaluation

In this section, we evaluate the CFAR prototype by answering two main questions:

- **Does CFAR work?** We show that CFAR extracts 100%-accurate data- and instruction-accesses distillates, and that this extraction completes in minutes for various kinds of systems code (§5.1).
- **Is CFAR useful** to system developers? We describe four use cases that demonstrate how CFAR provides developers with visibility into cache usage in a way that profilers and simulators cannot (§5.2).

Evaluation targets. We used CFAR to analyze the fast path of the transport layer of 4 TCP stacks: 2 versions of Linux's stack (before and after the recent reorganization for cache efficiency [42]), a TCP stack used by the IX kernel-bypass OS [6], and the lwIP TCP stack for embedded systems [20]. We also analyzed 2 hash-table implementations [60, 73], all 51 system calls in the Hyperkernel [51], and 7 algorithm implementations in OpenSSL 3.0.0 [54]. For the Linux TCP stack, we analyzed the stable versions before and after the reorganization (`v6.5` and `v6.8`). For all other code, we analyzed the latest stable version. IX uses the lwIP stack as a starting point, but heavily modifies the internal data structures and timer management [6].

We demonstrate that CFAR provides actionable cache-usage information for a broad spectrum of systems code. At one end of the spectrum are the Hyperkernel and OpenSSL, both of which are amenable to automated program analysis. The hash-table implementations occupy the middle, since they are both amenable to manual (but not automated) program analysis. At the other end of the spectrum are the four transport-layer implementations, which were not written to be amenable to any form of program analysis.

Setup. All experiments ran on an Intel Xeon E5-2690 v2 CPU at 3.30GHz with 25.6MB of LLC and 252GB of DRAM, with Ubuntu 22.04 and Linux kernel v5.4. The CFAR prototype incorporates a modified version of KLEE 2.1.

5.1 Does CFAR Work?

There are two key aspects to this question: does CFAR obtain an accurate abstract representation of performance from the code (§5.1.1), and does it do so in reasonable time (§5.1.2).

5.1.1 Accuracy of distillates

To measure the accuracy of our prototype's distillates, we randomly picked half the execution paths of each target, constructed inputs that exercised each path, counted the number of instructions and memory accesses executed while running with each concrete input, and then compared this number to the one predicted by the target's distillates.

The error was always **zero**, across all programs and inputs. That is, the number of instructions counted during real execution always equaled the number of memory accesses predicted by the instruction-accesses distillate for the given input, and the number of memory accesses counted during real execution always equaled the number of memory accesses predicted by the data-accesses distillate for that input. Additionally, CFAR's distillates correctly predicted every instruction-memory and data-memory address accessed by the code.

5.1.2 Time to extract distillates

Table 2 shows how long CFAR takes to extract distillates: for all programs, the analysis completes in less than 30 min. The longest times are for the Vigor hash table and the `echde` key-generation algorithm in OpenSSL; they are the only ones that take more than 15 min. This is because, for these programs, symbolic execution needs to unroll long loops that iterate over the hash map and that compute co-prime numbers, respectively. For all programs, the binary replay, execution-tree synthesis, and code synthesis take approximately 2-3 min in total. The dominant component of CFAR's analysis time—and the one that varies across programs—is symbolic execution.

Program	P_{dist} extraction time
Linux TCP ingress	11 min
Linux TCP egress	14 min
IX TCP ingress	5 min
IX TCP egress	7 min
lwIP TCP ingress	4 min
lwIP TCP egress	5 min
Hyperkernel syscalls (51 total)	Avg: 4 min / Max: 7 min
OpenSSL primitives (7 total)	Avg: 9 min / Max: 22 min
Vigor hash table	28 min
Klint hash table	12 min

Table 2: Time taken by CFAR to extract distillates.

5.2 Is CFAR Useful for System Developers?

We demonstrate CFAR's usefulness by presenting four cases of CFAR answering important questions that developer Alice cannot readily answer with the state of the art: How does my code's working set vary with the workload (§5.2.1)? I want to use a third-party data-structure library, but how does it interact with my cache (§5.2.2)? Does my code lead to inefficient memory-access patterns (§5.2.3)? Can I prove/disprove the absence of secret-dependent memory accesses (§5.2.4)?

5.2.1 How does the working set vary with workload?

We used the $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors to analyze the cache usage of the fast path of the transport layer of the four TCP stacks. Recall that this is the question that Alice wanted to answer in §2, but could not.

We constrain the input space as discussed in §4.3: focus the analysis solely on packets processed in the TCP fast path, i.e., packets that belong to an established TCP connection, are received in order, and do not suffer hash collisions with packets from other connections. We pick this particular class of packets because it represents a large fraction of packets processed by the TCP stack, on the path for which performance matters the most. The recent re-organization of the Linux TCP stack was focused entirely on this fast path [42].

First, we used $\mathcal{P}_{\text{scale}}$ to figure out the number of unique cache lines touched by the TCP fast path, for symbolic packet contents. The answer was 4, 5, 8, and 12 unique cache lines for the lwIP, kernel-bypass, Linux stack `v6.8` and `v6.5`, respectively. $\mathcal{P}_{\text{scale}}$ provides this information automatically, whereas benchmarking or code inspection would have a hard time producing it, because it cannot be gleaned merely by observing the size of the connection-specific `struct`. For example, in Linux, the `struct tcp_sock` occupies 42 cache lines in total, but only a fraction of them are accessed on the fast path.

We then passed this information to $\mathcal{P}_{\text{h/m}}$ and used it to predict when incoming packets were likely to suffer consistent cache misses due to the working set overflowing the LLC. The answer was that this would occur at approximately 91K, 76K, 47K, and 28K concurrent connections for the lwIP, kernel-bypass, Linux stack `v6.8` and `v6.5`, respectively. The small differences between these predictions and simple capacity-based calculations (e.g., 25.6M LLC / (64*4) = 100K connections for lwIP) are due to $\mathcal{P}_{\text{h/m}}$ being able to account for conflict misses in addition to capacity misses.

To verify these predictions, we ran a set of experiments where the transport layer receives and sends packets from/to a fixed set of established connections, and we varied the number of connections. To isolate just the transport layer (which is the code we analyzed), we wrote simple shims for the application and IP layers ourselves. In each experiment, we measured the average latency incurred by packets within the transport layer.

Fig. 7 plots packet-processing latency as a function of the number of connections. For each of the four stacks, there is a clear shift around the number of connections predicted by $\mathcal{P}_{\text{h/m}}$. For instance, the latency for the Linux stack `v6.5` increases by only 64ns from 1K to 26K connections, but increases by 211ns from 26K to 52K connections. Likewise, although less visible in the graph due to Linux's higher latency, the latency for the lwIP stack increases by only 13ns from 1K to 86K connections, and it increases by 50ns from 86K to 125K connections. The shift does not occur exactly at the predicted number of connections, but very close to it: compared to the predicted values of 28K, 47K, 76K, and 91K, we observed the shifts at 26K, 44K, 72K, and 86K, respectively. This difference is expected, because of $\mathcal{P}_{\text{h/m}}$'s set-associativity conflict approximation (§4.2.2) and because cache-mapping policies are proprietary, so $\mathcal{P}_{\text{h/m}}$'s cache model is imprecise.

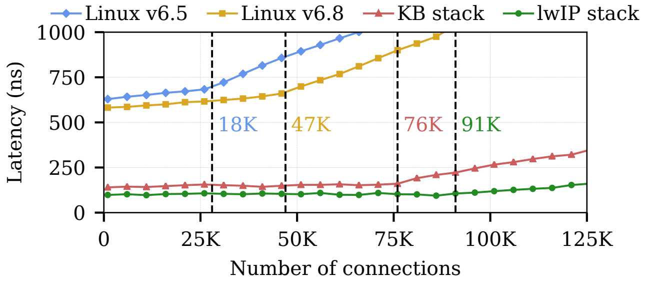


Figure 7: Measured latency for TCP packet processing as a function of the number of connections. CFAR predicted consistent LLC misses to start occurring at 28K, 47K, 76K and 91K connections for the Linux TCP stack `v6.5`, Linux TCP stack `v6.8`, the kernel-bypass (KB) stack and the lwIP stack, respectively.

Conclusion. Based on these results, we conclude that CFAR's $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors enable developers to accurately identify how the working set of third-party code or their own code changes as a function of the workload, without requiring them to run elaborate benchmarks. Given that CFAR can extract distillates in under 30 minutes, such extraction and analysis of distillates can become part of the regular development cycle (e.g., be made part of a continuous-integration pipeline), enabling developers to identify surprising performance behavior early and with relatively little effort.

5.2.2 How does third-party data-structure library code interact with my cache?

System developers often want to use third-party data-structure implementations, but are anxious about how that code will interfere with their own usage of the cache. We used the $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors to show how one can get cache-usage information for the hash-table implementations from Vigor [73] and Klint [60]. The analysis we show here can drive the choice between using one library vs. the other. We did not write the two libraries, but we read their code and *thought* we understood it fairly well.

For the CFAR analysis, we constrained the input space by fixing the maximum capacity of the hash tables to 64K entries. The resulting distillates had a different branch for each possible number of hash collisions, which is bound by the maximum capacity (thus, 64K cases). The conclusion of the CFAR analysis is therefore formally correct only for this maximum capacity. However, both hash-table implementations take the capacity as a configurable parameter, and the resulting memory-access pattern is independent of table capacity. We validated this through code inspection, we just cannot prove it formally using symbolic execution. Thus, like a developer, we proceeded to use $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ assuming that the distillates for the two hash tables provide valid predictions for any capacity, as long as the number of collisions does not exceed 64K. (The experiments validated this assumption.)

The projections proved our expectations about the performance of the two hash tables *wrong*. The two hash tables organize keys, values, and 4 metadata fields in slightly different ways: Vigor stores them as 6 distinct arrays, while Klint packs all 6 fields into a single 64B `struct` and maintains a single array with elements of this `struct` type. At first glance, it appears that the latter always leads to better locality and thus improved performance. However, it turned out that this is not always true.

Applying $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ to the `put()`, `get()`, and `delete()` operations of the two implementations predicts the following: For a `put()` or `get()`, both implementations bring 64B of data into the cache, but Klint does so in 1 cache line, while Vigor does so across 6 cache lines. When the table does not fit in the LLC, Klint suffers 1 LLC miss, while Vigor suffers 6. On the other hand, for a `delete()` call, both implementations touch the same 32B. However, Klint packs them together with other fields into a cache-line-aligned 64B `struct`, so it must bring the full 64B into the LLC, then update the 32B for invalidating the entry; the remaining 32B belonging to the deleted entry will never be reused. For Vigor, even though it brings a full 64B into the LLC, the other 32B belong to a still-valid entry and are likely to be reused, and thus the cost is amortized. As a result, for a range of table occupancies, Klint overflows the LLC and suffers 1 miss, while Vigor fits in the LLC and suffers none. $\mathcal{P}_{\text{h/m}}$ predicts that this range begins at approximately 400K keys and ends at approximately 800K keys, at which point both implementations overflow the LLC.

To verify these predictions, we measured the latency and LLC misses incurred by the `put()` and `delete()` calls of the two implementations. We configured a capacity of 2M entries for both hash tables. Fig. 8 plots Vigor's latency overhead relative to Klint, as a function of table occupancy. As predicted, Klint `put()` is consistently faster, due to better locality. Yet, for occupancies of 400K-800K keys, Klint `delete()` has 30% worse latency than Vigor. As predicted, Vigor incurs no misses in this range, while Klint incurs 1 per `delete()` call. There was one discrepancy between $\mathcal{P}_{\text{h/m}}$ predictions and the outcome of our experiments: for occupancies above 860K, $\mathcal{P}_{\text{h/m}}$ predicted 3 misses per `delete()` for Vigor, whereas we measured only 1 per call. We believe this to be due to Intel's stride prefetcher, which our current cache model does not take into account.

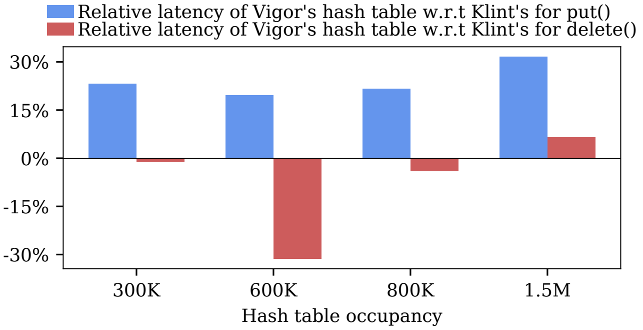


Figure 8: Relative latency (measured) of the Vigor hash table as compared to Klint's, for `put()` and `delete()` calls. Positive numbers indicate that the Vigor table is slower, and vice-versa.

Conclusion. Data-structure libraries often tailor their memory layout to different workloads [12–14]. Those who use the data structures need to understand these choices and the differences between different implementations. Using benchmarks can be tedious (e.g., in the present example, measuring the performance of `put()` and `delete()` for hash-table sizes up to 2M). Instead, CFAR projectors can quickly reveal the differences between how the implementations affect the cache across the entire range of inputs, allowing developers to pick the implementation best suited for their expected workload.

5.2.3 Does my code lead to inefficient access patterns?

We now describe how CFAR's projections helped us uncover two inefficient cache access patterns in the kernel-bypass TCP stack and the Hyperkernel's `mmap()` system call.

Kernel-bypass (KB) TCP stack: Motivated by the recent re-organization of the Linux TCP stack for cache efficiency—in particular, the `struct` that stores connection-specific data—we decided to check if CFAR's projections could help us improve the performance of the kernel-bypass stack as well. To understand how the fast path of the kernel-bypass stack was accessing different fields of the connection-specific `struct` (named `struct pcb` in this stack), we wrote a simple projector that returned the offset (in cache lines) of each access within the `struct pcb` from the base address of the `struct` (Fig. 9).

```
def pcb_offset(seq):
    pcb = sympy.Symbol('pcb')
    # if address is an offset from only the pcb
    # return (address-pcb)/64
    return [(x-pcb)//64 for x in seq
            if sympy.is_constant(x-pcb)]
```

Figure 9: Projector to compute the offset within the `pcb` structure.

Applying this projector to the fast path's `rcv()` and `snd()` calls revealed that there was only a single access to the 5th cache line in the `struct pcb`. Fig. 10 shows the list of cache-line accesses returned for `rcv()` and `snd()`, respectively.

```
# Receive fast path: KB stack
# Only one access to 5th cache line
[1,1,0,0,2,2,3,4,1,2,2,3]
# Send fast path: KB stack
# No access to 5th cache line
[2,3,3,1,1,3,3,3,3,1,2,3,2,2,1,1,1,1,0,0,2,1,2,2,1,0,2]
```

Figure 10: Cache-line accesses for `rcv()` and `snd()` on fast path.

Using the `file:linenum` information that CFAR logs during symbolic execution (§4.1.1), we realized that the field being accessed was `keep_cnt_sent`, which was being updated on the `rcv()` path to indicate that the connection was still live. To optimize this, we re-organized the `struct pcb` by moving `keep_cnt_sent` into the first 4 cache lines and moved some of the timer fields (primarily used during retransmissions) to the 5th line. Fig. 11 shows the list returned by the `pcb`-offset projector after this change, which confirmed that the fast path only accessed the first 4 cache lines.

```
# Receive fast path: KB stack
# No access to 5th cache line
[0,0,0,0,1,1,2,1,0,1,1,2]
# Send fast path: KB stack (updated)
# No access to 5th cache line
[1,2,2,0,0,2,2,2,2,0,1,2,1,1,3,3,3,3,3,3,1,3,1,1,0,0,1]
```

Figure 11: Cache-line accesses after our `pcb` optimization.

We evaluated the impact of this change by running the same experiment we ran in §5.2.1, where we measured the latency of the fast path as a function of the number of connections. Fig. 12 shows the results. Our optimization has a significant impact on the fast path's connection scalability: touching one less cache line enables the TCP stack to support 88K concurrent connections (instead of only 72K) before suffering from a latency increase due to LLC misses.

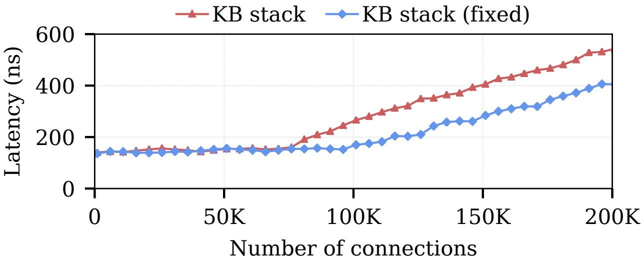


Figure 12: Before-and-after optimization: Latency as a function of the number of connections, for the kernel-bypass (KB) TCP stack.

Hyperkernel `mmap()`: CFAR enabled us to uncover and fix a subtle performance issue in Hyperkernel's `mmap()` implementation: The `mmap()` code performs a four-level page walk, checking for permissions only before it allocates the final page. So, if it is called with invalid permissions, it performs significant unnecessary work (allocates and zeroes out up to 3 new page-table pages, depending on where the walk stops), even if it does not exhibit incorrect behavior (i.e., does not allocate the final page). This brings up to 12KB of data into the L1 cache, which is more than 37% of the 32KB L1 cache in a modern server, so doing this unnecessarily pollutes the cache.

Fig. 13 shows part of the projection for `mmap()` resulting from a $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$ projector. Line 6 corresponds to the scenario where the permissions are invalid, and the walk fails at level 1 (i.e., no page-table page is allocated at that level for the target address). Line 12 corresponds to the scenario where the permissions are valid, and the walk fails at level 2. In the former case, the code touches `201` cache lines, whereas in the latter it touches `202`. So, even though the code need not allocate any pages in the former case, it touches almost identical numbers of cache lines in both cases.

```
 1 def mmap_dcache_num_cache_lines(va,perm):
 2     #State: pid, proc_tbl, pages
 3
 4     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>39)&511)]:
 5         if not (perm & PTE_PERM_MASK):
 6             return 201
 7         return 265
 8
 9     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>30)&511)]:
10         if not (perm & PTE_PERM_MASK):
11             return 138
12         return 202
13     ....
```

Figure 13: Unique data cache lines accessed by `mmap()`.

We fixed the code and ran CFAR again. Fig. 14 shows the new projection: in the invalid-permissions scenario (line 4), the code now touches 3 (instead of >200) cache lines.

Conclusion. The results show that the CFAR distillate, coupled with simple projectors, enables developers to efficiently (i.e., without benchmarking) inspect systems code and identify performance bugs that are otherwise hard to diagnose.

```
def mmap_optimized_dcache_num_cache_lines(va,perm):
    #State: pid, proc_tbl, pages

    if not (perm & PTE_PERM_MASK):
        return 3
    if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>39)&511)]:
        return 265
    if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>30)&511)]:
        return 202
    ...
```

Figure 14: `mmap` projection after fix.

5.2.4 Can I prove/disprove the absence of side channels caused by secret-dependent memory accesses?

Finally, we used CFAR's $\mathcal{P}_{\text{crypt}}$ projector to analyze the 8 OpenSSL algorithms listed in Table 3. The first 7 are the ones mentioned in the beginning of §5, while the last one is from a previous version of OpenSSL (`v1.1`). We included the latter because it is known to exhibit cache-based leakage (CVE-2018-0737 [55]), and we wanted to test CFAR's ability to identify this behavior (none of the algorithms we analyzed in the latest version of OpenSSL exhibit it). The $\mathcal{P}_{\text{crypt}}$ projector indeed confirmed the cache-based leakage in OpenSSL `v1.1`.

Program	Results
OpenSSL 3.0 AES	Identified *previously unknown branch-based leak*
OpenSSL 3.0 ChaCha	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 3.0 ECDHE	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 3.0 MD5	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 3.0 MD4	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 3.0 Poly1305	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 3.0 SHA-256	*Proved absence* of secret-dependent branches/mem accesses
OpenSSL 1.1 RSA	Reproduced *known cache-based leak* (CVE-2018-0737)

Table 3: OpenSSL programs analyzed using CFAR's $\mathcal{P}_{\text{crypt}}$.

We also uncovered a previously-unknown branch-based side channel in OpenSSL `v3.0.0`. The $\mathcal{P}_{\text{crypt}}$ projection revealed that the cipher-block unpadding function used by AES had secret input in the path constraint. To further investigate, we wrote another projector that counts the number of executed instructions. This revealed (Fig. 15) that the number of instructions executed by the function in question depends on the length of the input buffer's padding, making the code vulnerable to padding oracles.

```
def ossl_cipher_unpadblock_num_insns(buf, buf_len, block_size):

  if buf.padding_len == 0:
    return 44
  if buf.padding_len > block_size:
    return 48
  return 57 + 19*buf.padding_len
```

Figure 15: Instruction-count projection reveals that the number of instructions executed by AES's cipher unpadding is influenced by `buffer.padding_length`, which is a secret input.

We reported this to the maintainers, who confirmed it [56]. We submitted a fix, which has undergone multiple rounds of review and is now in the final stages of getting merged.

The instruction-count projection after the fix shows that the number of instructions is now independent of input (Fig. 15), thus proving that it achieves constant-time execution [3].

```
def ossl_cipher_unpadblock_num_insns(buf, buf_len, block_size):
  return 2985
```

Figure 16: AES instruction count after our fix.

Our experience with OpenSSL suggests that incorporating CFAR and its projectors into the development cycle would be beneficial. As it turns out, the side channel we found had been latent in OpenSSL since `v1.1.1`, which was released in 2019. It persisted despite the thorough code reviews that OpenSSL undergoes. Yet, a quick glance at the projection before the fix would have immediately revealed the problem. Perhaps, if distillates and projections were extracted regularly, e.g., as part of continuous integration, more side channels could be detected before making their way into production.

Conclusion. Since the distillate captures all information relevant to how a piece of code accesses memory, CFAR can help developers efficiently reason about more than just performance properties. $\mathcal{P}_{\text{crypt}}$ helps identify both branch- and cache-based leakage in cryptographic code (or prove their absence).

Evaluation summary. CFAR-extracted distillates are 100% accurate. They are useful to system developers because, together with projectors, they give visibility into cache usage in a way that profilers and simulators cannot. Our evaluation shows four concrete instances of such visibility and shows how CFAR enables developers to (1) reason precisely about the cache usage of code they or others wrote, without having to run elaborate benchmarks; (2) quickly identify cache-inefficient access patterns that are otherwise hard to diagnose; and (3) analyze code not only for performance bugs but also for cache-based security vulnerabilities.

6 Related Work

Performance interfaces. CFAR is part of an ongoing effort to augment systems with *performance interfaces* [30–32, 47]; these are meant to enable developers to efficiently reason about performance, just like semantic interfaces (abstract classes, specifications, documentation) enable reasoning about functionality. Some of this work [30, 31] provides visibility into the latency of software network functions, assuming a simple cache model that is appropriate for that particular domain. By providing visibility into the usage of shared micro-architectural structures (namely, the data and instruction caches), CFAR goes a step further: it enables developers to reason about the performance of a broader class of systems code, but also about the performance *side-effects* that a callee can have on a caller due to shared micro-architecture.

CFAR leverages two key ideas from prior work on performance interfaces, particularly PIX [30]. First, just like PIX (and Freud [63]), CFAR represents performance properties as programs that are both human-readable and executable. Second, CFAR's two-phased approach is similar to the separation between the PIX front- and back-end, which separates the performance properties of the code from the environment it runs in. PIX's two-phased approach has also been used by other recent work: `ltc` [47] uses it to provide visibility into the performance of hardware accelerators, while Performal [76] uses a similar approach to verify the performance of distributed systems. That said, CFAR's key technical contributions are the abstractions of the distillates and projections, which are specific to CFAR's focus on cache usage.

Using automated program analysis to reason about performance properties of systems code. Given the recent advances in automated program analysis techniques [27, 33, 37, 39, 64, 71], there is work that leverages such analysis to reason about various performance-related properties of systems code. Violet [28] uses it to find configuration bugs in large cloud applications, Bolt [31] and Castan [57] use it to analyze the latency of software network functions, and Clara [61] proposes using it to analyze the performance impact of offloading programs onto SmartNICs. However, in each of these cases, symbolic execution is coupled with an analysis framework that is specific to the property of interest. In CFAR, we use symbolic execution to extract all the information about how a piece of code uses memory, and we enable developers to write projectors that transform this information into the answers that the developers need.

Understanding the cache usage of systems code. Given the ever growing gap between processor and memory speeds, understanding how systems code uses the cache has been extensively studied. However, we are not aware of any tool that, like CFAR, possesses predictive power across unseen workloads. All prior tools we know of are limited to providing insights about the specific workloads that the tool was run on.

We drew significant inspiration from work in the 90s on abstract execution [40] and memory tracing [21]. Both these efforts aimed to replay the memory trace of a piece of systems code (just like CFAR's distillates), but only for concrete inputs. This is because their goal was to avoid having to store large memory traces required for computer architecture simulations, so they sought to generate this trace on the fly instead. CFAR's distillate thus represents a generalized version of their work, and builds on advances in automated program analysis.

More recent work has focused on building better profilers [10, 17, 35, 43, 49, 59, 69] to help developers fix performance issues that are caused by poor cache utilization. Such systems involve a fundamental trade-off between ease of use, performance overhead, and the level of detail at which they can analyze the execution of the given input workload. The most detailed memory profiler we know of is Memspy [49]: It uses a system simulator to execute an application, which allows it to interpose on all memory accesses and build a complete map of the cache. Thus, it can account for and explain every single cache miss and—using a processor-accurate model—can approximate memory-access latencies. However, Memspy requires porting applications to its simulator, which can be a painstaking task. Additionally, its high performance overhead restricts it to profiling a limited number of input workloads. At the other end of the spectrum are profilers like DMon [35]: These work off-the-shelf for almost any systems code, and have low enough overhead to run continuously in production. Their downside is that they can only be used to monitor a specific subset of events and cannot provide the visibility that MemSpy does.

We see profilers as complementary to CFAR. Distillates and projectors allow developers to quickly understand which workloads might be of interest and cause unexpected cache behavior. Once they narrow this search space, they can use state-of-the-art profilers to study these workloads in greater detail for specific, concrete inputs.

7 Conclusion

Developers need better *abstractions* to reason precisely about the expected performance behavior of their systems. Developers today are forced to manually inspect or profile the system *implementation* directly, which is both time-consuming and error-prone, since most systems today rely on a lot of third-party code. This is in contrast to how developers reason about functionality, where abstractions such as specifications, interfaces and documentation have been widely used for decades.

In this work, we focused on helping developers reason precisely about how systems code interacts with the underlying micro-architecture, specifically the CPU cache. We presented CFAR, a technique that introduces an abstraction that precisely captures what a piece of code does to the micro-architecture as a function of its inputs (the distillate) and provides a simple means of "querying" this abstraction, to help developers efficiently answer diverse questions about cache usage of their own, as well as third-party code, without having to delve into the code's details or run time-consuming benchmarks. We see CFAR as a key step towards augmenting systems with *performance interfaces* that describe the system's performance behavior in a manner that is simultaneously succinct, precise, and human-readable, just like semantic interfaces describe functionality.

We used CFAR to analyze different types of systems code and demonstrated that it can help developers identify performance bugs and security vulnerabilities, as well as understand the performance impact of using third-party code in their systems. CFAR's analysis completes in minutes, making it feasible to integrate CFAR into the software development cycle.

CFAR is publicly available as open-source software at [58].

8 Acknowledgements

We are grateful to colleagues who provided helpful feedback on drafts of the paper at various stages: Ed Bugnion, Marios Kogias, James Larus, and Sylvia Ratnasamy. We also thank our shepherd, Michael Stumm, and the anonymous reviewers for their feedback that significantly improved the paper.

References

[1] Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization. [Last accessed on 2024-05-23].

[2] Ainsworth, S., and Jones, T. M. Software prefetching for indirect memory accesses. In *International Symposium on Code Generation and Optimization* (2017).

[3] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *USENIX Security Symposium* (2016).

[4] AWS Elasticache. https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/AutoScaling.html. [Last accessed on 2024-05-23].

[5] Ayers, G., Nagendra, N. P., August, D. I., Cho, H. K., Kanev, S., Kozyrakis, C., Krishnamurthy, T., Litz, H., Moseley, T., and Ranganathan, P. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Internation Symposium on Computer Architecture* (2019).

[6] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Symposium on Operating Systems Design and Implementation* (2014).

[7] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. In *ACM SIGARCH Computer Architecture News* (2011).

[8] Boonstoppel, P., Cadar, C., and Engler, D. R. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[9] Burger, D., and Austin, T. M. The simplescalar tool set, version 2.0. In *ACM SIGARCH Computer Architecture News* (1997).

[10] Cachegrind: A Cache and Branch-Prediction Profiler. https://valgrind.org/docs/manual/cg-manual.html. [Last accessed on 2024-05-23].

[11] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation* (2008).

[12] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-Conscious Structure Definition. In *International Conference on Programming Language Design and Implementation* (1999).

[13] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-Conscious Structure Layout. In *International Conference on Programming Language Design and Implementation* (1999).

[14] Chilimbi, T. M., and Larus, J. R. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *International Symposium on Memory Management* (1998).

[15] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2011).

[16] The Clang compiler. https://clang.llvm.org.

[17] Curtsinger, C., and Berger, E. D. Coz: Finding Code that Counts with Causal Profiling. *Commun. ACM* (2018).

[18] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[19] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward Predictable Performance in Software Packet-Processing Platforms. In *Symposium on Networked Systems Design and Implementation* (2012).

[20] Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. Tech. Rep. 2:77, Swedish Institute of Computer Science, 2001.

[21] Eggers, S. J., Keppel, D. R., Koldinger, E. J., and Levy, H. M. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *ACM SIGMETRICS Conference* (1990).

[22] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A Performance Counter Architecture for Computing Accurate CPI Components. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2006).

[23] Farshin, A., Roozbeh, A., Jr., G. Q. M., and Kostic, D. Make the Most out of Last Level Cache in Intel Processors. In *ACM European Conference on Computer Systems* (2019).

[24] Fuerst, A., Novakovic, S., Goiri, I., Chaudhry, G. I., Sharma, P., Arya, K., Broas, K., Bak, E., Iyigun, M., and

Bianchini, R. Memory-Harvesting VMs in Cloud Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[25] Furia, C. A., Meyer, B., and Velder, S. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Survey* (2014).

[26] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *International Conference on Programming Language Design and Implementation* (2005).

[27] Godefroid, P., and Luchaup, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *International Symposium on Software Testing and Analysis* (2011).

[28] Hu, Y., Huang, G., and Huang, P. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Symposium on Operating Systems Design and Implementation* (2020).

[29] Hundt, R., Raman, E., Thuresson, M., and Vachharajani, N. MAO — An Extensible Micro-Architectural Optimizer. In *International Symposium on Code Generation and Optimization* (2011).

[30] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symposium on Networked Systems Design and Implementation* (2022).

[31] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance Contracts for Software Network Functions. In *Symposium on Networked Systems Design and Implementation* (2019).

[32] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The Case for Performance Interfaces for Hardware Accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[33] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. TRACER: A Symbolic Execution Tool for Verification. In *International Conference on Computer Aided Verification* (2012).

[34] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. E. TAS: TCP acceleration as an OS service. In *ACM European Conference on Computer Systems* (2019).

[35] Khan, T. A., Neal, I., Pokam, G., Mozafari, B., and Kasikci, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Symposium on Operating Systems Design and Implementation* (2021).

[36] King, J. C. Symbolic Execution and Program Testing. *Journal of the ACM 19*, 7 (1976).

[37] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. M. Loop Summarization Using Abstract Transformers. In *Automated Technology for Verification and Analysis* (2008).

[38] Kumar, P., Dukkipati, N., Lewis, N., Cui, Y., Wang, Y., Li, C., Valancius, V., Adriaens, J., Gribble, S., Foster, N., and Vahdat, A. PicNIC: Predictable Virtualized NIC. In *ACM SIGCOMM Conference* (2019).

[39] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient State Merging in Symbolic Execution. In *International Conference on Programming Language Design and Implementation* (2012).

[40] Larus, J. R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Softw. Pract. Exp.* (1990).

[41] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Symposium on Networked Systems Design and Implementation* (2014).

[42] Analye and reorganize core networking structs to optimize cacheline consumption. Linux Kernel mailing list. https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/. [Last accessed on 2024-05-23].

[43] The Linux Perf Tool. https://perf.wiki.kernel.org. [Last accessed on 2024-05-23].

[44] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium* (2018).

[45] Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., and Ngai, T. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques* (2009).

[46] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. PIN: building customized program analysis tools with dynamic instrumentation. In *International Conference on Programming Language Design and Implementation* (2005).

[47] Ma, J., Iyer, R., Kashani, S., Emami, M., Bourgeat, T., and Candea, G. Performance interfaces for hardware accelerators. In *Symposium on Operating Systems Design and Implementation* (2024).

[48] Marinov, D., and Khurshid, S. TestEra: A Novel Framework for Automated Testing of Java Programs. In *International Conference on Automated Software Engineering* (2001).

[49] Martonosi, M., Gupta, A., and Anderson, T. E. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *ACM SIGMETRICS Conference* (1992).

[50] Mowry, T. C., Lam, M. S., and Gupta, A. Design and Evaluation of a Compiler Algorithm for Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (1992).

[51] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Symposium on Operating Systems Principles* (2017).

[52] Github: State of the Octoverse 2022 - Programming Languages. https://octoverse.github.com/2022/top-programming-languages. [Last accessed on 2024-05-23].

[53] Olivo, O., Dillig, I., and Lin, C. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *International Conference on Programming Language Design and Implementation* (2015).

[54] OpenSSL. https://github.com/openssl/openssl. [Last accessed on 2024-05-23].

[55] OpenSSL CVE-2018-0737. https://github.com/advisories/GHSA-rj52-j648-hww8. [Last accessed on 2024-05-23].

[56] Pull request to fix constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/pull/16323. [Last accessed on 2024-05-23].

[57] Pedrosa, L., Iyer, R., Zaostrovnykh, A., Fietz, J., and Argyraki, K. Automated Synthesis of Adversarial Workloads for Network Functions. In *ACM SIGCOMM Conference* (2018).

[58] CFAR project website. https://dslab.epfl.ch/research/perf, 2024.

[59] Pesterev, A., Zeldovich, N., and Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In *ACM European Conference on Computer Systems* (2010).

[60] Pirelli, S., Valentukonytė, A., Argyraki, K., and Candea, G. Automated Verification of Network Function Binaries. In *Symposium on Networked Systems Design and Implementation* (2022).

[61] Qiu, Y., Kang, Q., Liu, M., and Chen, A. Clara: Performance clarity for smartnic offloading. In *ACM Workshop on Hot Topics in Networks* (2020).

[62] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. An Analysis of Performance Evolution of Linux's Core Operations. In *Symposium on Operating Systems Principles* (2019).

[63] Rogora, D., Carzaniga, A., Diwan, A., Hauswirth, M., and Soulé, R. Analyzing System Performance with Probabilistic Performance Annotations. In *ACM European Conference on Computer Systems* (2020).

[64] Saxena, P., Poosankam, P., McCamant, S., and Song, D. Loop-Extended Symbolic Execution on Binary Programs. In *International Symposium on Software Testing and Analysis* (2009).

[65] Sen, K., Marinov, D., and Agha, G. CUTE: a Concolic Unit Testing Engine for C. In *Symposium on the Foundations of Software Engineering* (2005).

[66] Soares, L., and Stumm, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Symposium on Operating Systems Design and Implementation* (2010).

[67] Sutherland, M., Gupta, S., Falsafi, B., Marathe, V., Pnevmatikatos, D., and Daglis, A. The NeBuLa RPC-Optimized Architecture. In *Internation Symposium on Computer Architecture* (2020).

[68] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K. J., Ratnasamy, S., and Shenker, S. ResQ: Enabling SLOs in Network Function Virtualization. In *Symposium on Networked Systems Design and Implementation* (2018).

[69] Valgrind. https://valgrind.org. [Last accessed on 2024-05-23].

[70] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup Graph Processing by Graph Ordering. In *ACM SIGMOD Conference* (2016).

[71] Xie, X., Chen, B., Liu, Y., Le, W., and Li, X. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Symposium on the Foundations of Software Engineering* (2016).

[72] Z3 Python API. https://github.com/Z3Prover/z3/blob/master/src/api/python/z3/z3.py. [Last accessed on 2024-05-23].

[73] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying Software Network Functions with No Verification Expertise. In *Symposium on Operating Systems Principles* (2019).

[74] Zarandi, A. P., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *International Symposium on Microarchitecture* (2021).

[75] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Symposium on Operating Systems Principles* (2021).

[76] Zhang, T. N., Sharma, U., and Kapritsos, M. Performal: Formal Verification of Latency Properties for Distributed Systems. In *International Conference on Programming Language Design and Implementation* (2023).

[77] Zhang, Y., Ding, W., Kandemir, M. T., Liu, J., and Jang, O. A Data Layout Optimization Framework for NUCA-Based Multicores. In *International Symposium on Microarchitecture* (2011).

[78] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. In *International Conference on Programming Language Design and Implementation* (2004).

[79] Zhou, D., Yu, H., Kaminsky, M., and Andersen, D. G. Fast Software Cache Design for Network Appliances. In *USENIX Annual Technical Conference* (2020).

VeriSMo: A Verified Security Module for Confidential VMs

Ziqiao Zhou[*] Anjali[†] Weiteng Chen[*] Sishuai Gong[‡] Chris Hawblitzel[*] Weidong Cui[*]
[*]*Microsoft Research* [†]*University of Wisconsin-Madison* [‡]*Purdue University*

Abstract

Hardware vendors have introduced confidential VM architectures (e.g., AMD SEV-SNP, Intel TDX and Arm CCA) in recent years. They eliminate the trust in the hypervisor and lead to the need for security modules such as AMD Secure VM Service Module (SVSM). These security modules aim to provide a guest with security features that previously were offered by the hypervisor. Since the security of such modules is critical, Rust is used to implement them for its known memory safety features. However, using Rust for implementation does not guarantee correctness, and the use of unsafe Rust compromises the memory safety guarantee.

In this paper, we introduce VeriSMo, the first verified security module for confidential VMs on AMD SEV-SNP. VeriSMo is fully functional and provides security features such as code integrity, runtime measurement, and secret management. More importantly, as a Rust-based implementation, VeriSMo is fully verified for functional correctness, secure information flow, and VM confidentiality and integrity. The key challenge in verifying VeriSMo is that the untrusted hypervisor can interrupt VeriSMo's execution and modify the hardware state at any time. We address this challenge by dividing verification into two layers. The upper layer handles the concurrent hypervisor execution, while the lower layer handles VeriSMo's own concurrent execution. When compared with a C-based implementation, VeriSMo achieves similar performance. When verifying VeriSMo, we identified a subtle requirement for VM confidentiality and found that it was overlooked by AMD SVSM. This demonstrates the necessity for formal verification.

1 Introduction

Confidential computing has been adopted by major cloud providers with the aim of removing the cloud provider out of the Trusted Computing Base (TCB). This is achieved by leveraging hardware-based Trusted Execution Environments (TEEs), which are encrypted and isolated from the rest of the software stack managed by the cloud provider. In recent years, hardware vendors have introduced confidential VM architectures (e.g., AMD SEV-SNP [1], Intel TDX [19], and Arm CCA [6]) that can run a full VM inside a TEE.

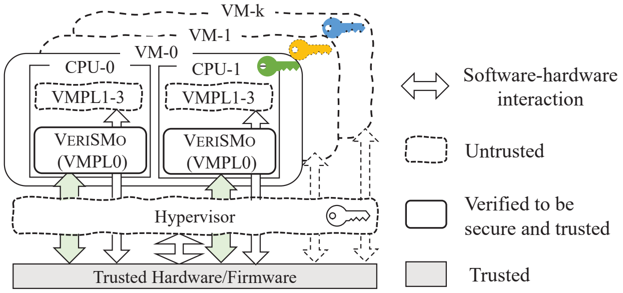


Figure 1: VeriSMo in AMD SEV-SNP architecture

While a confidential VM's confidentiality and integrity are protected from the untrusted hypervisor, it also means that a confidential VM cannot use security features that previously were offered by the hypervisor. To fill this gap, security modules such as AMD Secure VM Service Module (SVSM) were introduced to provide the missing security features in a privileged layer inside a confidential VM. Given the importance of the security of such modules [4, 39], Rust is used to implement them for its known memory safety features. However, using Rust for implementation does not guarantee correctness, and the use of unsafe Rust compromises the memory safety guarantee.

In this paper, we present VeriSMo[1], the first verified security module for confidential VMs on AMD SEV-SNP. Similar to other security modules like AMD SVSM [2], VeriSMo is a privileged software layer that runs inside a confidential VM and provides security features such as code integrity, runtime measurement, and secret management. The isolation between the security module and the guest OS is based on a new privilege dimension called Virtual Machine Privi-

[1] VeriSMo is derived from realism in the arts, particularly late 19th-century Italian opera. Its pronunciation reflects its small size.

lege Levels (VMPLs) on AMD SEV-SNP. VERISMO runs at the highest-privileged VMPL. Unlike other security modules, VERISMO is fully verified for functional correctness, secure information flow, and memory safety. Specifically, VERISMO is implemented in Rust and verified using Verus [22], a program verificaton tool designed for Rust.

AMD SEV-SNP provides confidential VMs with confidentiality and integrity. The former is achieved by encrypting the memory of a confidential VM and the latter is achieved by tracking the ownership of memory pages based on a new mechanism called the Reverse Map Table (RMP). While VERISMO can directly control memory encryption, it has to interact with the hypervisor to maintain the integrity of memory pages. This is because the hypervisor controls the nested page table and the owernship of memory pages in the RMP, while VERISMO is responsible for updating the RMP to validate memory pages assigned to a confidential VM.

The key challenge in verifying VERISMO is that the untrusted hypervisor can interrupt VERISMO's execution and modify the hardware state at any time. This concurrent interference makes it unwieldy to use standard Floyd-Hoare reasoning when verifying that VERISMO enforces the confidentiality and integrity of the VMs. To address this challenge, we divide verification into two layers. The upper layer handles the concurrent hypervisor execution, while the lower layer handles the VERISMO implementation, which is itself concurrent. This allows us to reason about these two different forms of concurrency (hypervisor interference and VERISMO's internal concurrency) using two different techniques:

1. For the upper layer, which we call the "machine-model layer", we define an abstract machine model that represents various physical hardware resources and hypervisor operations. We then prove that steps taken by this abstract machine preserve the confidentiality and integrity of the VMs.

2. For the lower layer, which we call the "implementation layer", we use Rust's ownership checking and Verus's permissions to reason about VERISMO's internal resources as the resources are accessed concurrently by different CPUs.

The interaction between the two layers is managed by preconditions that the VERISMO implementation must satisfy when performing hardware operations and postconditions that the VERISMO implementation can assume after hardware operations. For example VERISMO must satisfy a particular precondition when writing to a page table, and VERISMO can assume a postcondition about a memory page after executing the `pvalidate` instruction on the page. The upper layer can assume that the preconditions are satisfied, so that we can use these preconditions to verify that the abstract machine preserves the confidentiality and integrity of the VMs.

To make the implementation layer's verification scalable, especially with concurrent CPU access, we adopt permission-based reasoning, as suggested by previous research [8, 22, 31, 37]. This method combines ideas from Linear Logic [13] and Separation Logic [36], using access permissions as abstract capabilities for operations like reading and writing. Our approach applies these permissions to create type-safe interfaces for hardware resources, ensuring consistent maintenance of correct permissions during software interactions with these resources. Moreover, these interfaces, verified at the machine model level, guarantee memory safety and operational correctness in concurrent environments.

To enforce security information flow, we introduce a security type that carries possible value sets and security labels for each primitive type. The key concept here is to track a security level to each variable at every privilege level and ensure the proper relationship between the security level in value and the proper access permission in memory.

We built VERISMO mostly from the ground up, with the exception of integrating a verified cryptographic library [35], which we trust completely to avoid unnecessary duplication of verification efforts. We compared VERISMO with a C-based implementation and observed similar performance. It takes roughly 6 minutes to verify VERISMO on a 32-core machine, which shows the efficient proof time achieved through our optimized verification design and the use of Verus which is highly optimized for SMT solving.

In summary, our work makes the following contributions:

- VERISMO is the first verified security module operating within a confidential VM.
- We demonstrate how to verify VM integrity and confidentiality in the presence of a potentially malicious concurrent hypervisor, decomposing the verification into two layers to handle two levels of concurrency.
- We utilize the state-of-the-art Rust-based verification framework, showcasing the feasibility of constructing a verified real-world system using permission-based reasoning in Rust.
- We encode security flow policies using a type system and define safe casting to ensure the confidentiality of secret data while allowing all flexible accesses to secrets. (Section 8.4.1).

2 Background

2.1 AMD Confidential VMs

AMD Secure Encrypted Virtualization (SEV) is a confidential VM architecture. The latest version of AMD SEV, known as SEV-SNP, offers enhanced integrity and confidentiality protections for VMs.

Memory Encryption AMD SEV-SNP encrypts memory using a VM-specific encryption key, and secures the virtual

CPU (vCPU) state by encrypting and storing it in a VM Saving Area (VMSA) when the vCPU is trapped into the hypervisor. To support communication with the outside world (hypervisor or traditional IO devices), SEV allows VMs to selectively control encryption for memory pages by either setting an encryption bit in the guest page table or configuring a special MSR called vTOM.

Reverse Map Table AMD SEV-SNP introduces the Reverse Map Table (RMP) for memory integrity. It is located in the reserved system memory and is updated only with special CPU instructions – `rmpupdate` by the hypervisor or `rmpadjust` and `pvalidate` by the VM. The RMP is indexed by System Physical Addresses (SPAs), and each entry includes a Guest Physical Address (GPA) (as the reverse mapping of the nested page table), the assigned security domain (the hypervisor or a VM), as well as a validation bit to indicate whether the VM has accepted the memory assignment via `pvalidate`. To ensure the memory confidentiality and integrity, a confidential VM must correctly manage its page tables and the RMP.

VM Privilege Levels SEV-SNP introduces VM Privilege Levels (VMPLs) to isolate software running within a confidential VM. VERISMO runs in highest-privilege level—VMPL0, and we use VMPL3 to denote the level for running other softwares inside the VM. A vCPU's VMPL is stored in its VMSA. Different VMPLs share the same guest physical memory but have different permissions. By default, only VMPL0 has full permissions enabled to all guest memory pages. A VMPL can grant a subset of its permissions to a lower-privileged VMPL via the `rmpadjust` instruction. Those permissions are stored in the RMP and are part of the RMP check.

VM Platform Communication Key In AMD SEV-SNP, confidential VMs rely on the hypervisor to forward their messages to the Platform Security Processor (PSP) for tasks such as deriving new keys and generating attestation reports. To prevent attacks from a malicious hypervisor, The PSP uses VM Platform Communication Keys (VMPCKs) to establish secure channels with a confidential VM. These keys are passed to a confidential VM at launch time. VMPL0 has access to all keys and can choose to release some keys to other VMPLs.

VM Secure Interrupts A malicious hypervisor may inject arbitrary interrupts to change the data/control flow of the VM. Without secure interrupts, shared memory might be exploited by the hypervisor to leak sensitive data. For example, a recent research [38] demonstrates that #VC interrupts can leak sensitive data via the shared guest-hypervisor communication block (GHCB). To prevent the hypervisor from injecting arbitrary interrupts into a VM, AMD SEV-SNP introduces two secure interrupt injection modes: restricted interrupts and alternative interrupts. Each VMPL can have its own interrupt mode specified in the VMSA. When restricted interrupts are enabled, the hypervisor can only inject one interrupt type introduced by AMD called #HV. When a #HV arrives at a VMPL, the guest code at that VMPL can refer to a shared #HV doorbell page to check the interrupt type instead of directly jumping to an arbitrary interrupt handler. When alternative interrupts are enabled, the hypervisor cannot inject any interrupts into the VMPL, and the interrupts are always controlled by a higher-privileged VMPL. Thus, VMPL0 must use the restricted interrupt mode for security, while other VMPLs can use either the restricted or alternative interrupt mode.

2.2 Rust and Verus

Rust is a modern programming language that offers high performance and memory safety without requiring a garbage collector. Rust's ownership system enforces memory safety in a way conceptually similar to linear logic or separation logic. Rust is safe by default, meaning the compiler enforces memory safety guarantees. However, for scenarios where assembly code or direct control of memory is needed, Rust provides 'unsafe' blocks, which can cause bugs and memory safety issues [28].

Verus [22] is a verification tool designed for Rust. Verus extends Rust with verification features such as preconditions, postconditions, and loop invariants. For specifying and proving properties of Rust programs, Verus allows Rust developers to define three types of variables—executable, ghost, and tracked variables as well as three types of functions—executable, proof, and specification (spec) functions. The non-executable functions and variables are used by Verus during verification but are erased during compilation.

Ghost variables, which are used in proofs to represent mathematical abstractions such as sets or maps, are not checked by Rust's ownership checker. Tracked variables (referred to as "proof variables" in earlier versions of Verus [22]), on the other hand, are used to represent owned resources or permissions, and are checked by Rust's ownership checker. VERISMO uses tracked variables to represent permissions to access hardware resources such as memory and registers, in a style similar to separation logic or linear logic, but checked with Rust's ownership checker rather than with a dedicated separation logic or linear logic checker.

3 System Design

In this section, we present the system design of VERISMO.

3.1 Threat Model

VERISMO follows the threat model assumed by confidential computing. It only trusts the CPU and assumes that every-

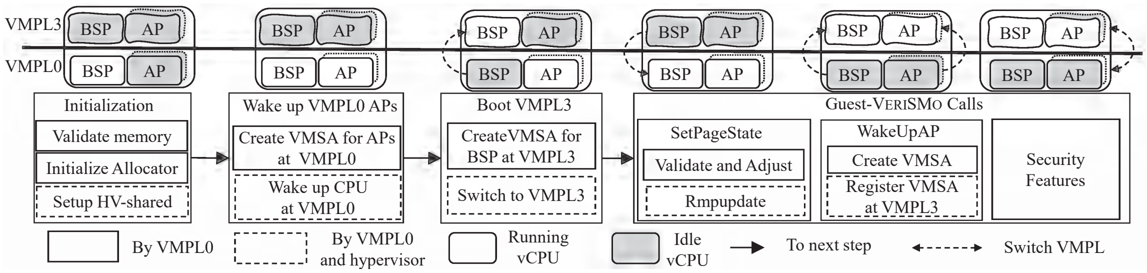


Figure 2: VeriSMo work flow

thing outside of a confidential VM is entirely controlled by an adversary, including the hypervisor. Although the hypervisor can interrupt a VM at any time, it can only inject the #HV interrupts because VeriSMo uses the restricted interrupt mode to prevent malicious interrupt injections. Furthermore, VeriSMo does not trust the guest OS running in the confidential VM. Denial-of-service attacks are possible, as the untrusted hypervisor manages the host and can either shut down the physical machine or opt not to schedule a vCPU for VeriSMo to run. Physical attacks are out of scope, as they are orthogonal to our work.

3.2 Architecture

VeriSMo runs in VMPL0, while the guest operating system runs in VMPL3 (VMPL1 or VMPL2 could also be used, but we choose VMPL3 in this paper). Both VeriSMo and the guest run in the retricted interrupt mode [2], and are thus not vulnerable to the interrupt injection attacks (e.g., [38]).

The work flow of VeriSMo is shown in Figure 2. When a confidential VM is launched, VeriSMo executes first. It reserves private memory for itself and then launches the guest OS. Afterwards, VeriSMo runs in a loop on each processor, waiting for calls from the guest OS.

3.3 Guest-VeriSMo Communication

VeriSMo and the guest OS running on a processor can transition execution to each other by issuing a hypercall to the hypervisor. Furthermore, a per-CPU memory page is shared between VeriSMo and the guest OS so that they can communicate with each other. The hypervisor does not have access to this memory page.

3.4 VeriSMo Guest APIs

The guest OS in VMPL3 must rely on VeriSMo to wake up its application processors (APs) and to validate memory pages, as it lacks these capabilities. Additionally, the guest OS can use VeriSMo-provided security features.

Waking up APs. During the boot time, the guest OS on the bootstrap processor (BSP) calls VeriSMo to activate APs. Upon receiving the request, VeriSMo's code running on the BSP notifies code running on APs. Once receiving the notification, VeriSMo's code running on an AP sets up a per-CPU VMSA page for the guest OS and transitions execution to the guest OS.

Guest Memory Management. While both VeriSMo and the guest OS are capable of sharing memory pages with the hypervisor, only VeriSMo can make memory pages private by validating them in the RMP. To track the state of memory pages (e.g., private/validated or shared/invalidated), VeriSMo requires the guest OS to use VeriSMo-provided APIs to share memory pages with the hypervisor. If the guest OS chooses not to follow this requirement, these shared pages will not be validated by VeriSMo anymore.

Guest Kernel Code Integrity. To assist the guest OS in preventing unauthorized code execution in kernel mode, VeriSMo offers the `LockKernel` API. The guest OS can invoke this API with a list of memory ranges corresponding to its kernel-mode code. VeriSMo will then remove from the guest OS the write permission to the kernel code pages and the supervisor-execution permission to other memory pages. VeriSMo also ensures that this API can be called only once.

Runtime Measurement. To facilitate runtime measurement for the guest OS, VeriSMo provides two APIs, `ExtendPCR` and `Attest`, based on a hash chain. The hash chain's initial value is set to the measurement of the guest OS's starting code and configuration. The guest OS can invoke `ExtendPCR` to extend the hash chain and call `Attest` with a nonce to request an attestation report. VeriSMo assembles the attestation report to include a hardware-attested report for VeriSMo's identity and a VeriSMo-attested report for the hash chain.

[2] The mainstream Linux (v6.8) does not support restricted interrupt injection in either KVM or the guest. We used the Hyper-V hypervisor and our modified guest Linux to enable restricted interrupts with #HV doorbell implementation.

Secret Management. VERISMO provides three APIs to support the guest OS for secret management: `DeriveKey`, `Encrypt`, and `Decrypt`. `DeriveKey` generates an encryption key derived from the current guest runtime measurement. This key is kept in VERISMO and is never discloses to the guest OS. The guest OS can invoke `Encrypt` or `Decrypt` to use this derived key to encrypt or decrypt data.

4 Verification Overview

4.1 Motivation

Traditional software testing can only partially check correctness for certain inputs and cannot formally ensure correctness. Formal verification is the only solution that provides a formal guarantee for the correctness. Below, we demonstrate the need for formally verifying three properties: functional correctness, secure information flow, and VM confidentiality and integrity.

Functional Correctness. Functional correctness defines the desired outcome (i.e., the postcondition) of a function when an input meets certain requirements (i.e., the precondition). In the following code, a key generation function contains a bug that results in a violation of the desired specification.

Listing 1: Incorrect functionality

```
fn GenPrivKey() -> (key: Key)
ensures key.is_random()
{
   return 123; // a constant is not random
}
```

Secure Information Flow. Secure Information Flow defines a safety problem by considering whether the information flow in a system is managed in a way that prevents unauthorized access or leakage of sensitive data. A program is said to be secure if and only if its memory trace and the values of low-security variables are independent of the initial values of its high-security variables. For example, the codes below show the security violation via data flow (left) and control flow (right).

```
low = high % 2
```

```
if high % 2 == 1 {a()} else {b()}
```

VM Confidentiality and Integrity. When a program *P* operating at a certain privilege level accesses memory *M* on a CPU, it is possible that *M* is concurrently accessed by *P* on a different CPU, or by another program at a different privilege level (e.g., the hypervisor). It is important to note that confidentiality and integrity violations within a program *P* can be eliminated through verification of *P* itself. However, the unexpected memory value due to concurrent updates from untrusted programs cannot be prevented. Thus, strict correctness cannot be verified against a specification relying on values from mutable shared memory, which is concurrently accessible by the hypervisor or other VMPLs.

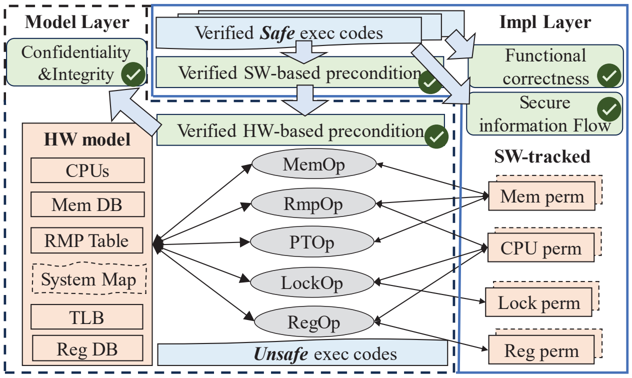


Figure 3: Two layer verification

Listing 2 illustrates an example where an incorrect modification of the page table can lead to the leakage of secrets to the untrusted hypervisor (confidentiality violation) and leave unintended effect in the software (integrity violation). Listing 3 shows a similar violation due to an incorrect RMP change.

Listing 2: Integrity/confidentiality violation via page table change

```
page_table_set_encryption(a_addr, false);
*a = ret_sensitive(); // Leaked result;
do_critical(&a); // Unintended result;
```

Listing 3: Integrity/confidentiality violation via RMP change

```
rmp_adjust(a_addr, READ, VMPL3);
*a = ret_sensitive(); // Leaked result;
rmp_adjust(a_addr, WRITE, VMPL3);
do_critical(&a); // Unintended result;
```

4.2 Verification Design

While we can adapt existing verification techniques to verify functional correctness and secure information flow, verifying VM confidentiality and integrity has its own challenge. The challenge comes from the fact that the untrusted hypervisor can interrupt VERISMO's execution and modify the hardware state at any time. This concurrent interference makes it hard to verify that VERISMO enforces VM confidentiality and integrity. Furthermore, there is another source of concurrency: VERISMO itself is a concurrent program. To handle these two different forms of concurrency separately, we divide verification into two layers: the machine-model layer and the implementation layer (see Figure 3).

In the machine-model layer, we define an abstract machine model that represents various hardware resources. Then we prove that steps taken by this abstract machine preserve VERISMO's confidentiality and integrity. In the implementation layer, we use Rust's ownership checking and Verus's permissions to reason about VERISMO's internal resources as the resources are accessed concurrently by different CPUs. The interaction between the two layers is managed by preconditions that the VeriSMo implementation must satisfy when performing hardware operations and postconditions that it can assume after hardware operations.

5 Machine-Model Layer

In this section, we describe the verification process at the machine-model layer. The goal of this layer is to prove that steps taken by an abstract machine ensures VeriSMo's confidentiality and integrity. Specifically, we need to ensure the following security properties required by VeriSMo.

(1) Ensuring the integrity and confidentiality of VM data in private memory. The hypervisor is unable to alter or explicitly read VM-private memory in the hardware through any sequence of operations by the hypervisor entity.

(2) Maintaining VMPL isolation. VMPL3 is unable to explicitly read VMPL0-private data, and whenever VMPL0 reads its own private data, it obtains the correct data, not data tampered with by VMPL3.

5.1 Abstract Machine Model

Our abstract machine model, Ψ, represents the contents and attributes of hardware resources such as registers, memory, page tables, and the RMP. This model is updated through some transition operations initiated by different entities. Since the model defines the interactions between these entities, we can formally check the preconditions for each operation required for ensuring the desired security properties.

5.1.1 Entities

Our abstract machine model has three entities.

E_0 represents VeriSMo executing at VMPL0.

E_3 represents the guest OS running at VMPL3. E_0 and E_3 share the same memory encryption key.

E_{hv} represents the hypervisor running in the hypervisor mode. E_{hv} does not have access to the memory encryption key of a confidential VM. Sibling guest VMs are ignored because the hypervisor's capabilities are their super set.

Both E_{hv} and E_3 are untrusted and can execute arbitrary code. Therefore, the value read out of the memory shared with E_{hv} or E_3 is treated as unconstrained.

5.1.2 Primitive Operations

Our abstract machine model defines a set of primitive operations (see Figure 3) that can be initiated by different entities to read or modify hardware resources. Each operation represents a single machine instruction, and its behavior is formally defined based on the AMD manual [3]. For instance, we define how the hardware model returns a memory value after nested page table walks and RMP checks.

For each operation, we define a trusted exec function with a single line of unsafe assembly in Rust with its pre- and post-condition. The postcondition reflects the operations's effect and are fully trusted. For instance, the postcondition for `pvalidate` is that the RMP entry is marked as validated. The preconditions of trusted functions are checked in the verification process to prove that, by enforcing the preconditions, the abstract machine state ensures the security properties when E_0's operation is constrained by the preconditions. The completeness of the operation model is important to our verification. Since VeriSMo is not as large as a guest OS, we currently only model critical memory and cache operations under some assumptions. For example, VeriSMo directly uses the guest-hypervisor communication to replace code that may trigger #VC, and always forces a VM termination when a #VC or other unexpected interrupt is triggered. Thus, we do not need to model the potential #VC events when accessing a memory.

5.2 Top-level Security Property Specifications

When proving the security properties (1) and (2), we prove both the confidentiality and integrity theorem outlined in Listing 4 and Listing 5.

Listing 4: VMPL0 confidentiality

```
proof fn proof_confidentiality(Ψ: Machine, e0: Entity,
      e: Entity, va1: Addr, va2: Addr)
requires
  E_0.contains(e0), e0 ≠ e,
  m_inv(Ψ),
  m_read(Ψ, va1, e0).is_Ok(),
  m_read_ret(Ψ, va1, e0).is_Secret(),
  m_to_spa(Ψ, va1, e0) ≡ m_to_spa(Ψ, va2, e),
  m_read(Ψ, va2, e).is_Ok(),
ensures
  m_read_ret(Ψ, va2, e).is_Encrypted();
```

Listing 5: VMPL0 integrity

```
proof fn proof_integrity(Ψ: Machine, Ψ': Machine, e0:
     Entity, va: Addr)
requires
  E_0.contains(e0),
  m_inv(Ψ, e0),
  attack_model(Ψ, Ψ'),
  m_read(Ψ, va, E_0).is_Ok(),
  m_read(Ψ', va, E_0).is_Ok(),
ensures
  m_read_ret(Ψ, va, e0) ≡ m_read_ret(Ψ', va, e0);
```

To prove confidentiality, a critical specification is the SNP machine invariant (`m_inv`) representing whether a machine state is valid. For the integrity proof, we additionally rely on a specification (`model_attack`) that determines whether a state Ψ' is reachable from Ψ under attack. Thus, it is necessary to prove the correctness of both the machine invariant specification (Listing 6) and the attack model specification (Listing 7). This entails consideration of machine model modifications stemming from all possible operations, where only $\mathtt{E}_0$'s operation is constrained by preconditions (`Op::sw_requires`).

Listing 6: Correctness of machine state invariant

```
proof fn proof_machine_inv(Ψ: Machine, Ψ′: Machine, op:
     Op, e0: Entity, e: Entity)
requires
  E0.contains(e0),
  (e0 ≡ e)  ⟹  Op::sw_requires(e, op),
  Ψ′ ≡ m_op(Ψ, e, op),
  m_inv(Ψ, e0),
ensures
  m_inv(Ψ′, e0);
```

Listing 7: Correctness of attack model

```
proof fn proof_attack_model(Ψ: Machine, Ψ′: Machine,
     Ψ″: Machine, op: Op, e0: Entity, e: Entity)
requires
  E0.contains(e0), e ≠ e0,
  m_inv(Ψ, e0),
ensures
  Ψ′ ≡ Machine:op(Ψ, e, op)  ⟹  attack_model(e0, Ψ,
     Ψ′),
  (attack_model(e0, Ψ, Ψ′) && Ψ″ ≡ m_op(Ψ′, e, op))
     ⟹  attack_model(e0, Ψ, Ψ″);
```

5.3 Security Property Proof Sketches

In this section, we describe five critical lemmas and provide a sketch of their proofs, in order to prove the two top theorems and the correctness of critical specifications. It is worth noting that they are fully proved with Verus.

In VERISMO, we classify the guest memory into three sets: VMPL0-Private , VMPL3-Private, and Hypervisor-Shared. The divided memory sets allow us define the security properties for different memory types.

5.3.1 VM-Private Memory

Lemma 1. *Let Ψ represent a machine state in which M is a guest physical memory block that stores value D. Suppose Ψ' is a future state reachable through modifications made by $\mathtt{E}_{\mathtt{hv}}$. Then, if M is VM-private in Ψ, VM's read operation on M in Ψ' either fails or returns the original value D.*

A key invariant property of the RMP is that, once a RMP entry is validated by a VM for a VM-private memory page, the guest physical address (GPA) of this memory page will be either bound to the system physical address (SPA) of the RMP entry or nothing at all, regardless of any operations by $\mathtt{E}_{\mathtt{hv}}$, as long as $\mathtt{E}_0$ does *not* validate the GPA again. It is straightforward to prove this property. If $\mathtt{E}_{\mathtt{hv}}$ makes any changes to the RMP entry, the entry will become invalidated, thus the GPA is not bound to any SPA. If $\mathtt{E}_{\mathtt{hv}}$ does not change the RMP entry, then the GPA remains bound to the same SPA as long as $\mathtt{E}_0$ does not validate the GPA again.

With this property, we can prove the Lemma 1. If *M* is VM-private and validated in Ψ, then the GPA of *M* is bound to an SPA. This implies that the read operation on *M* by either $\mathtt{E}_0$ or $\mathtt{E}_3$ in Ψ' either returns the original value *D* or fails.

This lemma essentially requires that a valid state of Ψ will always ensure the VM-private *M* has unique bound from a GPA to an SPA no matter how the hypervisor changes the nested page mapping, as we discussed in Section 2.1. Such invariant property requires that $\mathtt{E}_0$ does not validate a GPA when it is validated and bound to an SPA in the RMP.

Lemma 2. *Given a machine state Ψ, if a guest physical memory block in VM-private is mapped to a system physical memory M that stores a VM's secret S, then in any future hypervisor-reachable machine state Ψ', $\mathtt{E}_{\mathtt{hv}}$'s read operation on M will return an encrypted version of S.*

At first glance, one may assume that the VM-private memory page requires both the encryption bit in the guest page table and the validation bit in the RMP. However, our verification indicates that the validation bit in the RMP cannot be reliably guaranteed. When proving the lemma, we confirmed that holding the validation bit in the hardware state is not necessary. This implies that a requirement for a 'C' bit in the page table suffices to prove the lemma.

5.3.2 VMPL0-Private Memory

Lemma 3. *Let Ψ represent a machine state in which M is a VMPL0-private guest physical memory block that stores value D. Suppose Ψ' is a future reachable state through modifications made by $\mathtt{E}_{\mathtt{hv}}$ and $\mathtt{E}_3$. Then the $\mathtt{E}_0$'s read operation on M in Ψ' either fails or returns the original value D, and M cannot be read by $\mathtt{E}_3$.*

Since VMPL0-private memory is a subset of VM-private memory, Lemma 1 and Lemma 2 implies that $\mathtt{E}_{\mathtt{hv}}$ cannot read it or tamper its value. Here we focus on $\mathtt{E}_3$. An RMP entry contains access permissions for each VMPL. These permissions control whether a VMPL can read, write, and execute on the memory. Furthermore, the hardware restricts $\mathtt{E}_3$ from modifying access permissions for its own VMPL. To ensure that $\mathtt{E}_3$ cannot access VMPL0-private memory, the verification process requires a precondition to `rmpadjust` that $\mathtt{E}_0$ cannot grant access permission to VMPL3 if the memory is in VMPL0-private.

5.3.3 Correct Guest Address Translation

In addition to RMP updates, updating the page table is also critical for safe memory translation. We must ensure the integrity of the memory translation by considering all possible changes from all entities. We verify it by proving the following lemma. The correctness of our guest address translation helps to prove the top theorem when considering accesses via guest virtual addresses instead of guest physical addresses.

Lemma 4. *Let Ψ represent a machine state in which a guest virtual address GVA is successfully translated to a system physical address SPA by a VMPL0's memory access. Suppose Ψ' is a future state reachable through modifications made by* $\mathtt{E_{hv}}$ *and* $\mathtt{E_3}$*. Then, in Ψ', VMPL0's access to the GVA succeeds with the same translation to SPA or fails.*

Lemma 3 establishes that a GPA for VMPL0-private memory is either bound to a specific SPA or not bound to any SPA. When a GVA is successfully translated to a SPA in Ψ, it implies that the GPA that the GVA is mapped to is bound to the SPA. Therefore, in Ψ', the GPA is either still bound to the same SPA or not bound to any SPA. Since $\mathtt{E_{hv}}$ and $\mathtt{E_3}$ cannot change $\mathtt{E_0}$'s page table, the translation from the GVA to the GPA remains the same. This guarantees that the GVA is either translated to the same SPA or fails.

To simply the implementation layer verification, we also prove the following lemma to ensure the mapping from GVAs to SPAs is one-to-one.

Lemma 5. *For each reachable Ψ, the mapping from a guest virtual address to a system physical address is a one-to-one mapping.*

Since Lemma 3 implies that the mapping from GPAs to SPAs for VM-private memory is one-to-one, we only need to ensure that the mapping from GVAs to GPAs is one-to-one. We prove it by separating memory writes into two categories: normal memory (`mem_write`) and page table memory (`pt_write`). To simplify the proof, we set aside a set of guest physical pages for $\mathtt{E_0}$'s page table (referred to as the PT memory), and enforce that the PT memory is always VMPL0-Private (by updating the precondition to `rmpadjust`). This allows us to define a precondition for `mem_write` and `pt_write` to check that a memory write falls into their respective categories.

A trusted initial assumption we make is that the initial page table for $\mathtt{E_0}$ at launch time is correct in the sense that the page table pages are in the PT memory and the page table enforces a one-to-one mapping from GVAs to GPAs. Then to prove it is true for any reachable Ψ, we prove that any `pt_write` operation preserves this property by adding a precondition to check that the memory write would keep the page table pages in the PT memory and the one-to-one mapping from GVAs to GPAs.

5.3.4 Connecting Machine Model to Implementation

The confidentiality and integrity of the VM-private and VMPL0-private memory, together with the correct page table translation, ensure that the memory content accessed by $\mathtt{E_0}$ through a guest virtual address remains consistent with the content stored in the hardware state. This consistency allows the implementation layer verification to focus on the software-tracked state, eliminating the complexity of having to worry about the actual hardware state. To convert preconditions for primitive operations from hardware-based to software-based in implementation verification, we prove that if an operation succeeds and the operation's software-based constraint is true, the corresponding hardware-based one must be true.

6 Implementation Verification

In this section, we describe the verification at the implementation layer. For simplicity, software in this section refers to the implementation of VeriSMo. We first describe how we use permission-based verification to handle concurrency and scale verification to a large codebase. We then describe how we use information-flow verification to prevent secret leakage.

6.1 Permission-based Verification

In VeriSMo, we incorporate the software constraints derived from the machine model verification into "tracked" permissions defined by Verus[22]. Each resource permission includes an identifier and multiple fields that represent the value or attributes of the resource. To ease the proof process across various memory access scenarios, we opt for implementing fine-grained memory permissions. This approach helps avoid the complexities tied to a single large-size global state (e.g., the hardware abstract model used in Section 5), and simplify the concurrency reasoning using ownership. Moreover, to aid in safe memory sharing, we introduce a lock permission for shared memory. To ensure safe register access, we establish register permissions in accordance with their definitions.

6.1.1 Memory Access Permission

Object-based Memory Permission. We extend the definition of a basic memory permission described in Verus to make all memory access safe in the context of AMD SEV-SNP VMs. A memory permission is defined as a `tracked` variable (`SnpPointsTo`) without the ability to be copied or constructed. By incorporating an appropriate initial assumption to ensure the initial uniqueness of all memory permissions, we can guarantee the uniqueness of each permission throughout the program.

As shown in Listing 8, our extended SNP memory permission consists of three elements: the guest virtual address (`addr`) as the permission identifier, the `value` stored at that address by the software, and the memory attributes (`swattr` and `hwattr`) as seen by both software and hardware. These memory attributes include RMP (`rmp`) and page table (`pte`) values tied to the memory. Furthermore, considering the specific use of page tables, we have added an attribute (`is_pt`) to denote whether the memory serves as a page table. These improvements facilitate efficient management and enforcement of memory safety within the SEV-SNP VM context.

Listing 8: SNP object-based memory permission definition

```
pub ghost struct SnpMemAttr
{ rmp: RmpEntry, pte: PTAttr, is_pt: bool}

pub ghost struct SnpPointsToData<T> {
  addr: int, value: Option<T>,
  swattr: SnpMemAttr, hwattr: SnpMemAttr,
}

pub tracked struct SnpPointsTo<V>
{ _p: marker::PhantomData<V>, _ncopy: NoCopy }

impl<T> SnpPointsTo<T>
{ pub spec fn view(&self) -> SnpPointsToData<T>; }
```

Raw Memory Permission While object-based access permissions provide a user-friendly approach to object-oriented programming, VeriSMo operates as a low-level security module, involving a significant number of raw memory operations.

To effectively support raw memory, it is necessary to establish additional foundational information concerning size, value casting, memory splitting, and merging. This essential ground-truth information is not provided by Verus and is defined by VeriSMo as trusted specifications or axioms:

Object Size Specification. We assume that an object's size is equivalent to its actual memory usage. The precise size value should only matter when an operation has a specific size requirement (e.g., pvalidate requires page-sized) or when size comparisons are necessary (e.g., memory splitting or joining).

Casting between Objects and Bytes. Our trusted proof for casting aims to enforce unique bindings and ensure consistent sizing between objects and their corresponding byte representations.

Raw Memory Split and Merging. During memory splitting and merging operations, byte values and memory ranges are divided or combined, respectively, while memory attributes remain consistent with the original state.

Examples to Convert Unsafe Rust to Safe Verus. To illustrate how to use memory permission to convert Rust's unsafe memory access into Verus's safe memory access operation, we provide two dummy examples using memory primitive functions defined in Listing 9. The examples demonstrate how unsafe accesses can be identified through either verification (ϟ) or Rust's borrow checker (⊗).

The first example (in Listing 10) takes a memory permission reference pointing to a VMPL0-private memory at 0x1000, and thus it can borrow a value at address 0x1000. However, Line 8 cannot change content, since the permission is borrowed as immutable; Line 9 cannot access raw memory at 0x2000 due to the mismatched memory identifier.

Another example provided in Listing 11 demonstrates how the verification process detects unsafe RMP updates, ensuring the valid memory state. It initializes an non-validated memory permission and then assign it to VMPL1 at the end to render the memory accessible to VMPL0. After `pvalidate`, the memory permission remains not ready for other RMP memory operations until the operation is confirmed and the memory content is cleared. The strict requirement leads to a failed assertion at Line 6. Additionally, Line 11 fails since the pvalidate primitive function requires no double validation.

Listing 9: A selective primitive memory-related functions

```
fn borrow<'a>(vaddr: usize, Tracked(mperm): Tracked<&'a
      SnpPointsTo<V>>) -> (v: &'a V)
requires
  mperm@.wf_borrow(vaddr as int),
ensures
  mperm@.spec_read_rel(*v),
{...}
fn replace(vaddr: usize, in_v: V, Tracked(mperm):
     Tracked<&mut SnpPointsTo<V>>)
requires
  old(mperm)@.wf_replace(vaddr as int, in_v),
ensures
  mperm@.spec_write_rel(old(mperm)@, Some(in_v)),
{...}
fn pvalidate(vaddr: u64, psize: u64, val: bool, rflags:
      &mut u64, Tracked(mperm): Tracked<&mut
     SnpPointsToRaw>) -> (ret: u64)
requires
  spec_pval_requires(vaddr as int, psize, val, old(
     mperm)@),
ensures
  spec_arch_pval(vaddr as int, psize, val, old(mperm)@,
      mperm@, *old(rflags), *rflags, ret),
{...}
fn rmpadjust(vaddr: u64, psize: u64, attr: RmpAttr,
     Tracked(mperm): Tracked<&mut SnpPointsToRaw>) -> (
     ret: u64)
requires
  spec_rmpadjust_requires(vaddr as int, psize as int,
     attr, old(mperm)@),
ensures
  spec_arch_rmpadjust(old(mperm)@, mperm@, vaddr as int
     , psize as int, attr),
{...}
```

Listing 10: Secure access to memory

```
1  fn access_private(Tracked(mperm): Tracked<&SnpPointsTo<
        u64>>)
2  requires
3    mperm@.wf_not_null_at(0x1000),
4    mperm@.is_vmpl0_private() {
5  ✓let val1 = *borrow(0x1000, Tracked(mperm));
6  ✓let val2 = *borrow(0x1000, Tracked(mperm));
7  ✓assert(val2 == val1);
8  ⊗replace(0x1000, 0x1234, Tracked(mperm));
9  ϟ let _val3 = *borrow(0x2000, Tracked(mperm));
10 }
```

Listing 11: Safe RMP table updates

```
 pub fn init_page(Tracked(mperm): Tracked<SnpPointsToRaw
     >)
 requires
   mperm@.wf_range((0x1000, PAGE_SIZE)) && mperm@.
      is_init() {
   let mut u64 rflags = 0;
✓let ret = pvalidate(0x1000, 0, 1, &mut rflags,
      Tracked(&mut mperm));
⚡ assert(mperm@.wf());
   if ret == 0 && rflags != 0 { return false; }
✓mem_set(0x1000, PAGE_SIZE, 0, Tracked(&mut mperm));
✓assert(mperm@.wf());
⚡ pvalidate(0x1000, 0, 1, &mut rflags, Tracked(&mut
      mperm));
   let rmpattr = RmpAttr::empty().set_vmpl(1).set_read
      (1).set_write(1);
✓rmpadjust(0x1000, rmpattr, Tracked(&mut mperm));
   if ret  != 0 { return false; }
✓assert(!mperm@.is_vmpl0_private());
   return true;
 }
```

6.1.2 Lock Access Permission

The memory permission we previously described does not entirely address the issue of concurrency reasoning. This challenge emerges because memory permissions, in their current form, do not inherently facilitate the shared write permission for concurrent access. For example, when CPU-A moves a memory permission to CPU-B, CPU-A subsequently loses access to that memory. How to retrieve the memory permission back is unclear without introducing locking or atomic permission mechanisms.

To enable safe concurrent memory access in a relaxed memory model, we choose to implement locking permissions since much of our code relies on locks to protect shared resources without directly using atomic operations. Each shared resource starts with a lock permission which stores a memory permission. When the software acquires a lock, the lock permission is converted to a locked state and returns the stored memory permission so the software can use the memory permission to access the shared resource. When the lock is released, the memory permission is returned back to the lock permission whose state is converted back to an unlock state.

Shared objects typically require an **invariant** to constrain their values. Without proving the invariant, for all verifications necessitating such a constraint, programmers may incur extra execution costs to check if the value meets the requirements. To maintain the invariant for values read after acquiring a lock, we have introduced a precondition in the release API. This precondition mandates that the object associated with the memory permission upholds the invariant whenever the release method is invoked. As a result, the value of the global variable read by a CPU is always in compliance with the invariant. The lock mechanism in VERISMO extends beyond shared memory (see Section 7.2). Listing 12 shows an example to use lock to protect a global variable `gvar` while keeping its invariant (`spec_gvar_inv`). The verified code guarantees that the data borrowed satisfies invariant (Line 12) after acquiring the lock, although the data could be different in two lock transactions (Line 13). Our trusted lock APIs can detect the violation of invariant for `gvar` since it fails the invariant at Line 15.

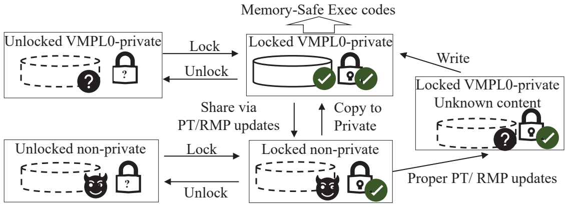


Figure 4: Safe memory access under concurrency

Listing 12: Lock protection

```
1  spec fn spec_gvar_inv() -> spec_fn(u64) -> bool
2  { |v: u64| 0 <= v <= 0xff_ffff}
3
4  fn access_global(Tracked(core): Tracked(SnpCore),
       Tracked(lperm): Tracked(LockPerm<u64>))
5  requires
6    lperm.is_unlocked(spec_gvar) {
7  ✓let (vaddr, Tracked(pt_mperm)) = gvar().acquire(
       Tracked(&mut lperm), Tracked(&core));
8  ✓let val1 = borrow(vaddr, Tracked(&pt_mperm));
9  ✓gvar().release(Tracked(&mut lperm), Tracked(&core),
       Tracked(pt_mperm));
10 ✓let (_, Tracked(pt_mperm)) = gvar().acquire(Tracked(&
       mut lperm), Tracked(&core));
11 ✓let val2 = borrow(vaddr, Tracked(&pt_mperm));
12 ✓assert(spec_gvar_inv()(val1) && spec_gvar_inv()(val2)
       );
13 ⚡ assert(val1 == val2);
14 ✓replace(vaddr, 0x1000_0000, Tracked(&pt_mperm));
15 ⚡ gvar().release(Tracked(&mut lperm), Tracked(&core),
       Tracked(pt_mperm));
16 }
```

6.1.3 VMPL0's Memory Safety

By utilizing the memory permission and the lock permission reasoning, we ensure that our implementation always has safe memory access. Here we describe ownership for VERISMO at a high level utilizing the permissions we described in Section 6.1.1 and Section 6.1.2.

Figure 4 illustrates safe operations for handling memory and lock permissions. When VMPL0-private memory is locked by the software on a CPU, no other entity can concurrently modify it. Therefore, the memory can be considered as private to VMPL0, and the acquire operation will safely grant a memory permission for unrestricted memory access. For memory shared with either the hypervisor or the guest OS, the memory permission obtained upon locking only permits copying and writing, but not borrowing references. For unrestricted operations, developers must either copy the memory to VMPL0-private memory or use rmpadjust/pvalidate operations to transition it to VMPL0-private memory.

6.1.4 Register Access Permission

Similar to memory permissions, we define register permissions for registers. Most registers are privately controlled by the software and thus we usually can maintain a stable invariant property after proper booting steps. Since a CPU has a fixed number of registers, each CPU uses a single permission representing all the CPU's registers together. Some registers need special permission designs.

In the SEV-SNP VM environment, the *GHCB MSR* differs from other registers. While all other registers are owned by the VM, the GHCB can be read and written by the hypervisor. To handle this difference, we introduce a "share" attribute for such registers, treating the values read from these shared registers as unconstrained.

For registers like *IDTR* and *GDTR*, which hold pointers to memory segments, we guarantee that memory ownership is transferred to the hardware upon writing to these registers.

Certain registers, such as the instruction pointer (*RIP*) and stack pointer (*SP*), pose a risk of inadvertently influencing software behavior in dangerous ways. We prevent explicit use of these registers by removing permissions to access them.

CR3 points to the top-level page table. A write operation to *CR3* is required to have tracked mutable permissions for both the page table and all usable virtual memory addresses. This safeguards against incorrect modifications to CR3, as consolidating all memory permissions used by the CPU is complex. VeriSMo does not have a user mode, and has a single page table for its kernel mode code. Therefore its CR3 register is only updated once at the boot time.

6.2 Information-Flow Verification

In addition to guaranteeing safe memory access, it is important to prevent secret leakage through explicit or implicit information flows. In VeriSMo, we define a precise tracking policy to maintain secure information flow by monitoring potential secret guessing spaces for variables.

Types carrying secret guessing space. We define security types to match Rust's primitive types. Each security type includes a value and a set of possible values (`valset`) that an entity can guess from. If the set is complete, the variable is considered a secret to the entity; if the set is singleton, the variable is public to the entity.

VeriSMo only takes three kinds of secrets: the VM communication keys generated by the hardware (Section 2.1), VeriSMo's own private/public key pair (Section 3.4), and symmetric encryption keys for the guest OS (Section 3.4). Since these secrets are exclusively used by trusted cryptographic functions in VeriSMo, we only encounter a limited number of proofs about the set size when invoking cryptograhic functions. With a trusted and formally verified crypto library, our security checking is highly simplified to check whether a variable is fully public or confidential to an entity, similar to taint tracking without over/under-tainting concerns.

We define a security trait that assigns security levels to different data types based on the size of their possible value sets. Primitive types in Rust have security levels equivalent to constants. To use security types like primitive types, we implement standard operator traits and ensure the correct propagation of the guessing space by applying the relevant set operations to the possible value sets. For example, each binary operation 'op' performed between variables a and b ensures the following constraints for the returned result.

$$(a \text{ op } b).\text{valset} \equiv \left\{ \text{val} | \exists (v,u) : \begin{array}{r} v \in \text{valset}_a \wedge \\ u \in \text{valset}_b \wedge \text{val} \equiv a \text{ op } b \end{array} \right\}$$
$$\wedge (a \text{ op } b).\text{val} \equiv (a.\text{val op } b.\text{val})$$

A comparison operation may lead to secret leakage via control flow. Therefore, the comparison operator includes a precondition that requires both variables to be public to all entities. We support the secret downgrade through a trusted function when needed. After a downgrade, a variable's possible value set becomes a singleton and thus can no longer be used as a secret. For instance, if a downgrade operation is applied to a secret key, the key no longer meets a precondition for trusted cryptographic functions, which requires cryptographic key to be a high-security variable, i.e., the possible value set if full.

Re-visit the memory permission for confidentiality. To maintain the confidentiality of secret variables, we impose a requirement that the software must not share them with other entities. To enforce this property, we ensure that every valid memory permission maintains a consistent relationship between the memory's confidentiality attribute (defined by the page table and the RMP) and the security level of the value it carries, as shown below:

$$\forall \text{ entity } \neg\text{memperm.swattr.is\_confidential\_to(entity)}$$
$$\implies \text{memperm.val.is\_constant\_to(entity)}$$

In addition, we cannot use secret data as address for memory access; otherwise, the hypervisor can use control flow to infer the secret. We enforce this precondition for all trusted functions related to memory access.

7 Implementation

We implemented VeriSMo in Rust and verified it with Verus, ensuring that the trusted functions are always used safely, and our implementation is correct. Our implementation is available at `https://github.com/microsoft/verismo`.

7.1 TCB in VERISMO

VERISMO fully trusts a small number of primitive functions, ground-truth axioms, the hardware model specification, the model-based specification defining the safety properties, as well as the specification for functional correctness. These trusted functions wrap unsafe code for interacting with the hardware for calling trusted external libraries (e.g., Rust core, HACL[35]). The size of the trusted code will be reported in the evaluation section. Additionally, VERISMO places trust in Verus and the Rust compiler.

7.2 An example with simplified verification

The permission-based verification not only guarantees memory confidentiality and integrity but also ensures correctness for certain functionalities without introducing additional specifications. Here, we use an example in VERISMO to explain how a lock can automatically protect associated resources and how memory permissions automate functional correctness for a Guest-VERISMO call to update a memory page's attributes.

VERISMO assigns certain memory ranges to VMPL3 and stores their *ranges* (for execution) along with a zero-sized tracked permission *map* (for proof) in a global variable (`OSMEM`). The `OSMEM` is shared by multiple cores and is protected by a lock that guards an invariant, ensuring that a memory page has its tracked permission inside the *map* if and only if the page is within the memory *ranges*.

Extended lock protection. When VMPL3 sends a request to VERISMO to grant or revoke permission for a page, VERISMO needs to acquire the lock to access the tracked permission from `OSMEM`. This automatically prevents concurrent changes to the RMP table for VMPL3's memory without introducing new locks. Here, a single lock protects both the `OSMEM` variable and the memory assigned to VMPL3.

Extended Functional Correctness. Additionally, since the tracked map only stores page permissions from the `OSMEM`-defined ranges, VERISMO must check whether the requested page is within the range to obtain the page permission required for updating the memory's attributes. This automatically ensures that a correct handler will conduct the necessary checks before any update happens.

8 Evaluation

Our performance evaluation is based on the Hyper-V hypervisor. Our experimental machine has an AMD EPYC 7543P 32-Core Processor, which supports SEV-SNP features. We allocated 8 dedicated cores to the host domain using minroot Hyper-V, allowing the hypervisor to assign the remaining 24 CPUs to VMs. This setup prevents unpredictable competition for CPU resources between different domains.

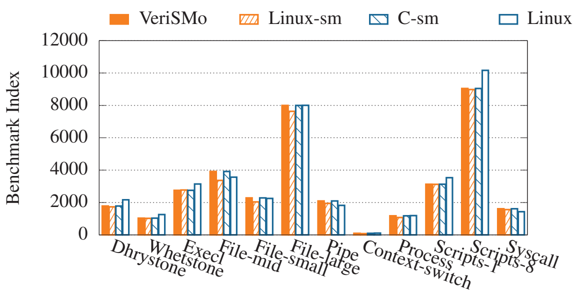


Figure 5: Unix benchmark results

8.1 Performance

To measure the performance of guest-VERISMO APIs, we created a kernel driver that sends these requests to VERISMO. We ran each call 100 times and recorded the average cost per call in Table 1.

Table 1: Microbenchmarking guest-VERISMO APIs

Request	Cycle (*1k)	Time (ms)	STD (ms)
`Switch`	34	0.012	0.002
`ExtendPCR`	37	0.013	0.002
`SetPageShared`	77	0.027	0.002
`SetPagePrivate`	82	0.029	0.002
`AttestPCR`	17633	6.298	4.270
`LockKernel`(8GiB)	4260969	1522	151
`LockKernel`(4GiB)	2201565	786	58

The `Switch` call serves as a test to complete a switch (VMPL3 → hypervisor → VMPL0 → hypervisor → VMPL3), and its cost prevails over some less expensive GVCA calls. The operation to extend a measurement (`ExtendPCR`) is quick, as it only involves adding a value from VMPL3 to the previous one and updating it with a cryptographic hash. `SetPageShared` is relatively more expensive since VERISMO needs to invalidate it and call the hypervisor to make a page shared which triggers additional context switch. `SetPagePrivate` is marginally more expensive than `SetPageShared`, as the former requires an additional rmp change via `rmpadjust`. Attestation is much more resource-intensive than others due to the need for hardware-based cryptograhic signing. Kernel code integrity protection (`LockKernel`), operating on all guest memory, is costly, but it is performed only once per VM session.

8.2 Performance Impact on Guest OS

We used UnixBench to measure the performance of a Linux kernel running with the VERISMO security module at VMPL3. The benchmarking results are presented in Figure 5. The security module introduces almost zero overhead. In contrast, Hecate [12], a compatibility-oriented L1 hypervisor in

Table 2: Lines of code.

	Exec	Spec	Proof	Axiom	Trusted
Verus	906	1361	2388	803	101
Model	0	2194	3188	67	0
HACL	107	7	0	0	90
Macro	2093	-	-	-	-
Common	606	603	2101	55	10
RawMem	640	635	324	105	11
Reg	341	162	22	0	12
Lock	183	145	105	0	7
PageTable	354	261	330	0	1
Alloc	386	164	1003	24	30
Core	3205	1013	3884	14	0

VMPL0, can incur up to a 40% slowdown. This is because our security module does not trap any guest OS's #VC Exception.

In some case, formal verification can be simplified in a less-efficient implementation or with extra runtime checks. Here, we compared VERISMO with two additional prototypes that we implemented, demonstrating no performance trade-off introduced by verification. While those prototypes—'linux-sm' (a modified Linux kernel as security module) and 'c-sm' (C-based security module)—only supports `WakeUpAP`, `SetPageShared`, and `SetPagePrivate` APIs, they serve as baselines for the fully-featured VERISMO. The comparable performance of VERISMO demonstrates that we did not sacrifice performance for the sake of verification.

8.3 Verification Size and Performance

Table 2 shows the number of lines of codes for all used libraries and modules in our code. 'Verus' represents modules VERISMO used from Verus's basic library, 'HACL' wraps 3 functions for encryption and hashing from an external formally verified HACL crate written in C and assembly. 'Core' represents code for VERISMO's core functionalities. The trusted executable code (majorly unsafe code) are from three categories. We only used 31 LOC unsafe Rust for hardware primitive operations. 'RawMem', 'Reg', 'Lock' and 'PageTable' include the trusted primitive operations for memory and registers. 'HACL', 'Common' includes external safe functions with a trusted postcondition. 'Alloc' includes 1 unsafe block to trust the global allocator interface. Although this allocator interface must be trusted, as the Rust compiler depends on it, we have verified the correctness of the implementation of our global allocator. On average, the proof and specification annotations are approximately twice the size of the executable code for the implementation layer. The machine-model layer is reusable if we do not change primitive functions. Verification of both model and implementation layers takes around 6 minutes with a multi-threaded Verus backed by Z3-4.11.2 as the solver running on our test machine with 32 cores. The verification efficiency is due to Verus's optmization for Z3 solver, Rust's ownership checking for automated memory reasoning, and the modularity of our proof code with Verus.

8.4 Security Improvement

Concurrently with our project, AMD SVSM [4] which is now replaced by COCONUT SVSM [39], is an ongoing Rust project to provide a security module in VMPL0 for AMD SEV-SNP VMs. However, they do not apply formal verification and heavily use unsafe Rust features. COCONUT uses 235 unsafe blocks, and AMD SVSM uses 150 unsafe blocks. In contrast, VERISMO uses only 32 unsafe blocks. We did not compare our performance with them since VERISMO runs on a different hypervisor. Since they share the same hardware model as VERISMO, our verification design can potentially be applied to their code.

8.4.1 An Example Bug Detected in VERISMO

Here, we showcase the importance of formal verification, using an unsafe memory update in Listing 13 that we detected via verification. This code handles the `SetPageShared` request from VMPL3. Before modifying the RMP entry for a memory page used by VMPL3, a traditional approach is to check the security of the change by examining whether the content carried in the guest physical page can be released externally. It uses a lock to protect VMPL3's memory to prevent concurrent updates. However, these measures are not sufficient for ensuring VERISMO's confidentiality when taking into account a malicious hypervisor.

Listing 13: Handling a memory state change request

```
fn SetPagePrivate(gpn: u64, attr: RmpAttr, lperm:
      Tracked<LockPerm>) -> Tracked<LockPerm> {
  let gpn = vn_to_pn(gvn);
  let tracked mut lperm = lperm;
✓let osmem = OSMEM.acquire(Tracked(&mut lperm));
✓match osmem_check(osmem, gpn, attr) {
  Ok(i) => {
    let Tracked(mut pperms) = osmem[i].pperms;
  ✓let tracked mut pperm = pperms.tracked_remove(gvn);
  ✓pvalidate(gvn, true, Tracked(&mut pperm), ..);
  ⚡ rmpadjust(gvn, 1, attr, Tracked(&mut pperm), ..);
```

Consider two system physical memory pages: SPN$_0$ and SPN$_3$. SPN$_0$ is mapped to GPN$_0$, while SPN$_3$ is mapped to GPN$_3$. The memory at SPN$_0$ holds VMPL0's secret key, which must remain confidential to VMPL3 and the hypervisor. Meanwhile, the memory at GPN$_3$ is allocated to VMPL3, and thus its access permissions can be adjusted upon VMPL3's requests. VMPL3 could gain access to the secret key at SPN$_0$ using the following steps (with the hypervisor's help):

1. VMPL3 requests VERISMO to transition GPN$_3$ to a shared status. VERISMO fulfills the request by invalidating the target memory and recording that GPN$_3$ is now invalidated since it is assigned to VMPL3.

2. To perform the attack, a malicious hypervisor binds SPN_0 with GPN_3 by executing `rmpupdate` and updating the nested page table to map GPN_3 to SPN_0.

3. When VMPL3 asks VeriSMo to revert GPN_3 back to private, VeriSMo validates the memory page at GPN_3 and adjusts permissions to permit VMPL3 access to the memory. This occurs because VeriSMo incorrectly assumes that GPN_3 was shared and does not contain VMPL0's secret.

As a result of the attack, VMPL3 obtains access to SPN_0 via GPN_3, which violates VeriSMo's confidentiality. The hardware's *failed-to-be-secure* design only detects the attack when there is an attempt to access the memory at GPN_0, as the memory binding for GPN_0 becomes invalid.

Without verification, detecting such a security vulnerability can be challenging. Its solution is straightforward: simply clear a memory page after validating it but before assigning it to VMPL3.

We reported this vulnerability to AMD in early May 2023, which resulted in a security fix [5]. However, COCONUT SVSM [39], which reuses a significant amount of code from AMD SVSM, still had the old buggy code at its early stage.

9 Related Work

Trusted execution environments. Traditional hypervisor-enforced VM isolation no longer meets the minimal trust requirements. This has led to a recent trend of adding an extra software layer as a trusted security monitor to deal with hypervisor-based attacks. Examples include Keystone [23], Komodo [11], and the Intel TDX module [18], which enforce enclave isolation and provide security features. Arm CCA [6] goes a step further by relying on two external layers for VMs: a Realm Management Monitor (RMM) for managing realms and a separate monitor for facilitating interactions between the RMM and the hypervisor. However, the issue of separating security domains within an enclave remains largely unaddressed. Different from these solutions, the security module [2, 4, 39] for AMD SEV VM [1] is isolated from both hypervisor and other in-TEE softwares in a VM.

Model checking. Formal methods can be applied to both model-level (i.e., model checking) and implementation-level verification. Model checking tools (e.g., SPIN[17], Tamarin[30]) formally checks some properties based on an abstract model of a system design (e.g., [15, 20]) to prove the correctness. Consequently, the correctness is proved without directly connecting to the implementation itself.

Software verification. Software projects (e.g., [11, 16, 29, 35, 40]) can be implemented in verification-friendly languages, such as Dafny [25] and F*[34], to enable end-to-end verification for both model and implementation. Many OS projects (such as [7, 10, 14, 21, 24, 26, 32]), based on unsafe assembly or C , are verified in proof languages. The implementation-level verification is made possible through a trusted language transformer from unsafe C. Due to the nature of unsafe C, however, their memory safety proofs require more effort than proofs about Rust. A recent study [9] uses verification to check secure information flow for confidential computing but assumes memory safety by requiring no unsafe Rust in code, which is not applicable to OS-level code.

Security model for verified OS. OS-level verification efforts ([7, 11, 14, 21, 26, 33]) typically focus on the traditional security model, either with a trusted hypervisor or without one. Some recent efforts, such as [27], have verified the security of the Arm CCA, ensuring the proper implementation of the RMM firmware for isolating multiple enclaves or VMs. Different from those works, We use the permission-based method to verify proper memory access, encoding both the state of the memory and the security level of its content. This approach simplifies our proof and streamlines our verification process, eliminating the need for additional abstract layers for proving concurrency and information flow. In addition, we verify memory accesses without extra proof efforts for the safe portion of the Rust code.

10 Conclusion

We developed and implemented VeriSMo, the first verified security module for confidential VMs enabled by AMD SEV-SNP. Operating at the highest privilege level, the security module provides protection to the guest while maintaining its own confidentiality and integrity, even in the presence of an untrusted concurrent hypervisor. Our verification process validated the security and correctness of the security module software, building upon our specifications of the AMD hardware primitives. Utilizing Rust's ownership and borrowing, the verification showcased the application of concepts from Verus's permission model on a large scale, encompassing thousands of lines of verified concurrent executable code. Our evaluation demonstrated that our security module is efficient and our verification is scalable.

Acknowledgments

We thank Verus team for sharing their insights into permission-based reasoning and providing desired features for verifying VeriSMo. The detailed reviews we received from OSDI'24 and the feedback from our shepherd Nickolai Zeldovich helped us greatly improve the paper.

References

[1] AMD. Strengthening VM isolation with integrity protection and more, 2020. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[2] AMD. Secure VM service module for SEV-SNP guests. *White Paper, August*, 2022. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf.

[3] AMD. *AMD64 Architecture Programmer's Manual (v4.07)*. Volume 3: General-purpose and system instructions edition, 2023. https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5.

[4] AMD. Secure VM service module for SEV-SNP guests. 2023. https://github.com/AMDESE/linux-svsm.

[5] AMD. Linux secure VM service module security fix for hypervisor-based attacks, 2023. https://github.com/AMDESE/linux-svsm/commit/0de111e9b85a340203759a3ab217a3e2f2be4b0b.

[6] Arm. Arm confidential compute architecture (Arm CCA), 2021. https://www.arm.com/products/security/arm-confidential-compute-architecture.

[7] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium*, volume 152, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[8] J. Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003. doi: 10.5555/1760267.1760273.

[9] H. Chen, H. H. Chen, M. Sun, K. Li, Z. Chen, and X. Wang. A verified confidential computing as a service framework for privacy preservation. In *32nd USENIX Security Symposium*, pages 4733–4750, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/chen-hongbo.

[10] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics: 22nd International Conference*, pages 23–42. Springer, 2009. doi: 10.1007/978-3-642-03359-9_2.

[11] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *26th ACM Symposium on Operating Systems Principles*, pages 287–305, 2017. doi: 10.1145/3132747.3132782.

[12] X. Ge, H.-C. Kuo, and W. Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *29th ACM Conference on Computer and Communications Security*, pages 1231—-1242, New York, NY, USA, 2022. ISBN 9781450394505. doi: 10.1145/3548606.3560592.

[13] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987. doi: 10.1016/0304-3975(87)90045-4.

[14] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, volume 16, pages 653–669, 2016. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[15] R. Guanciale, M. Balliu, and M. Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *27th ACM Conference on Computer and Communications Security*, pages 1853–1869, 2020. doi: 10.1145/3372297.3417246.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *25th ACM Symposium on Operating Systems Principles*, pages 1–17, 2015. doi: 10.1145/2815400.2815428.

[17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi: 10.1109/32.588521.

[18] Intel. Intel TDX module v1.5 base architecture specification, 2023. https://cdrdv2.intel.com/v1/dl/getContent/733575.

[19] Intel. Intel trust domain extensions (Intel TDX) module TD partitioning architecture specification, 2023. https://cdrdv2.intel.com/v1/dl/getContent/773039.

[20] M. K. Jangid, G. Chen, Y. Zhang, and Z. Lin. Towards formal verification of state continuity for enclave programs. In *30th USENIX Security Symposium*, pages 573–590, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/jangid.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification

of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009. doi: 10.1145/1629575.1629596.

[22] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *38th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2023.

[23] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *15th European Conference on Computer Systems*, pages 1–16, 2020. doi: 10.1145/3342195.3387532.

[24] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Formal Methods: Second World Congress*, pages 806–809. Springer, 2009. doi: 10.1007/978-3-642-05089-3_51.

[25] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: In 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

[26] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A secure and formally verified linux kvm hypervisor. In *42nd IEEE Symposium on Security and Privacy*, pages 1782–1799. IEEE, 2021. doi: 10.1109/SP40001.2021.00049.

[27] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the Arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 465–484, 2022. `https://www.usenix.org/conference/osdi22/presentation/li`.

[28] Z. Li, J. Wang, M. Sun, and J. C. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *2021 ACM Conference on Computer and Communications Security*, page 2183–2196, New York, NY, USA, 2021. ISBN 9781450384544. doi: 10.1145/3460120.3484541.

[29] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–304, 2013. doi: 10.1145/2451116.2451148.

[30] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *25th International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013. doi: 10.1007/978-3-642-39799-8_48.

[31] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2.

[32] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy*, pages 415–429, 2013. doi: 10.1109/SP.2013.35.

[33] S. Peters, A. Danis, K. Elphinstone, and G. Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–7, 2015. doi: 10.1145/2797022.2797042.

[34] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. doi: 10.1145/3110261.

[35] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *41st IEEE Symposium on Security and Privacy*, pages 983–1002, 2020. doi: 10.1109/SP40000.2020.00114.

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

[37] A. Sadiq, Y.-F. Li, and S. Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, 159:110450, 2020. doi: 10.1016/j.jss.2019.110450.

[38] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde. WeSee: Using malicious #VC interrupts to break AMD SEV-SNP. In *45th IEEE Symposium on Security and Privacy*, 2024. `https://ahoi-attacks.github.io/wesee`.

[39] SUSE. COCONUT-SVSM, 2023. `https://github.com/coconut-svsm/svsm`.

[40] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *24th ACM Conference on Computer and Communications Security*, pages 1789–1806, 2017. doi: 10.1145/3133956.3134043.

Validating the eBPF Verifier via State Embedding

Hao Sun
ETH Zurich

Zhendong Su
ETH Zurich

Abstract

This paper introduces *state embedding*, a novel and highly effective technique for validating the correctness of the eBPF verifier, a critical component for Linux kernel security. To check whether a program is safe to execute, the verifier must track over-approximated program states along each potential control-flow path; any concrete state not contained in the tracked approximation may invalidate the verifier's conclusion. Our key insight is that one can effectively detect logic bugs in the verifier by embedding a program with certain approximation-correctness checks expected to be validated by the verifier. Indeed, for a program deemed safe by the verifier, our approach embeds concrete states via eBPF program constructs as correctness checks. By construction, the resulting state-embedded program allows the verifier to validate whether the embedded concrete states are correctly approximated by itself; any validation failure therefore reveals a logic bug in the verifier. We realize *state embedding* as a practical tool and apply it to test the eBPF verifier. Our evaluation results highlight its effectiveness. Despite the extensive scrutiny and testing undertaken on the eBPF verifier, our approach, within one month, uncovered 15 previously unknown logic bugs, 10 of which have already been fixed. Many of the detected bugs are severe, *e.g.,* two are exploitable and can lead to local privilege escalation.

1 Introduction

The Extended Berkeley Package Filter (eBPF) [32, 37] allows untrusted user space extensions to be executed in kernel space. This mechanism has been broadly adopted by modern operating system kernels to flexibly implement various specialized tasks, including filtering [43], profiling [28], and security monitoring [15], among others [23, 54]. To ensure the safety of the untrusted extensions, a static checker [8] (verifier) is utilized to rigorously validate their integrity. In this work, our primary focus is on the eBPF verifier in the Linux kernel, which is mature and has successfully been applied in various contexts. The eBPF verifier employs abstract interpretation [20], a process where it traverses the program and gathers approximations across different abstract domains to identify potentially invalid behaviors. Due to its intricate checking mechanism, the verifier has evolved into one of the most complex components within the eBPF subsystem.

The correctness of the eBPF verifier is of utmost significance. The eBPF subsystem provides extensibility by granting restricted kernel space code execution capability to user space, which is enforced by the verifier. These restrictions are crucial as they limit memory access and control flow in programs, thereby preventing the kernel from being impacted by potentially harmful extensions. However, logic bugs in the verifier can compromise these restrictions, leading to unsafe programs being loaded. Indeed, the verifier's vulnerabilities are attractive to attackers as these bugs have a higher likelihood of being exploited to inject malicious programs into the kernel [1–3]. We will demonstrate, in Section 2.2, the exploitation of one such bug we found, a simple incorrect type cast in the verifier, to achieve local privilege escalation. Therefore, detecting and rectifying logic bugs in the eBPF verifier is critical to the overall kernel security.

Given its importance, existing work applies formal verification to several components of the verifier. For example, Agni [45] generates verification conditions for the range analysis of the verifier, and other work [44, 52] aims to verify the `tnum` domain [35]. These efforts have provided strong guarantees for the correctly verified components. Nevertheless, given its complexity, obtaining specifications, either manually or automatically, is intrinsically challenging even for a portion of the verifier [18]. Consequently, these checked specifications may be incomplete [9] or diverge from the implementation [4], *e.g.,* we still uncovered logic bugs in the verified range analysis. Moreover, these components are relatively small, and the verifier is constantly evolving with new algorithmic enhancements and features. Previous work has also applied automatic testing on eBPF [25, 26, 42]. For instance, Syzkaller [47] has been incorporated into the eBPF upstream and has identified many memory errors in the eBPF system call. Yet, they encounter challenges in detecting logic bugs due to the lack of effective test oracles [24], namely methods to automatically determine whether a program should be accepted or rejected by the verifier.

Observation. In essence, the verifier checks eBPF programs by tracking states at different locations along each possible execution path within its abstract domains, *i.e.,* the verifier state (approximation). The checking procedure on each execution path can be modeled as verifier state transitions:

$$A_0 \rightarrow A_1 \rightarrow ... \rightarrow A_{n-1} \rightarrow A_n$$

The verifier state transition corresponds to a set of concrete state transitions on the corresponding execution path:

$$S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{n-1} \rightarrow S_n$$

Each concrete state S_i must be contained in the corresponding approximation A_i; otherwise, it is a verifier bug since attackers can manipulate such a concrete state, thereby breaking the verifier's conclusion. The key observation is that, to ensure the conclusion is correct, the verifier must over-approximate all possible concrete states at each program point. In other words, any concrete state not contained in the approximation can invalidate the conclusion. The property is fundamental and can be harnessed as an effective test oracle to validate the correctness of the verifier without requiring specifications. The ensuing challenge is how to determine whether or not concrete states are contained in the approximation.

State Embedding. This paper introduces state embedding, a novel and effective mechanism for validating the eBPF verifier. Our key insight is: one can effectively detect logic bugs by embedding a program with the aforementioned approximation-correctness checks that are expected to be validated by the verifier itself. State embedding contrasts pairs of programs, P and P'. Initially, a program P, accepted by the verifier, is executed to profile its concrete states. Next, P' is crafted by embedding sinks that contradict these observed states into P, challenging the verifier to validate it. A correct verifier should reject P' since a valid approximation must include the observed concrete states, thus accepting P' reveals a logic bug. More concretely, given an accepted program P and a profiled concrete state S. Corresponding to S is a variable A representing the verifier's approximation. P' is constructed by embedding the following program construct:

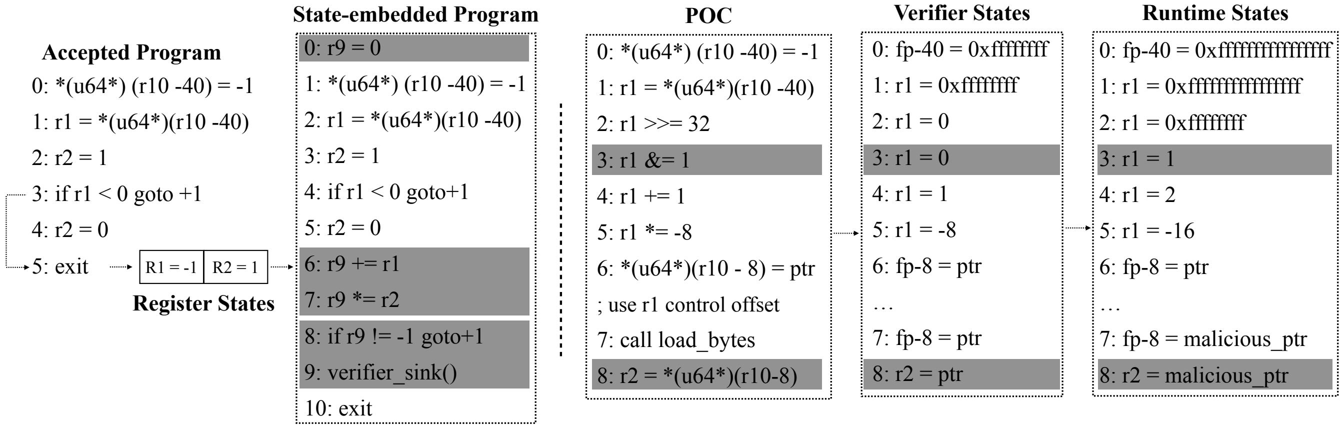
$$\textbf{if } S \in A \textbf{ then } \textit{verifier\_sink}()$$

The condition directs the verifier to check if the concrete state S is indeed contained within its approximation A, and *verifier_sink*() refers to any incorrect operation; encountering this during validation signals an error, indicating that S is correctly contained in A. One can easily realize the construct by utilizing the if-condition statement with equality comparison; we defer to Section 3.1 for concrete examples. Thus, P' can be applied to validate the verifier V: $\mathcal{P} \rightarrow \{\textit{safe}, \textit{unsafe}\}$, where marking P' as *unsafe* due to the triggered sink confirms the inclusion of S in A; conversely, deeming P' safe indicates a failure in capturing S within A, *i.e.,* a verifier's logic bug.

Realization. We realized state embedding as a practical tool, which we call SEV, and applied it in validating the eBPF verifier. First, for an eBPF program accepted by the verifier, SEV executes and profiles the program to gather its register states at each basic block. Second, to efficiently embed each state, we utilize the following optimization (which will be further elaborated in Section 3.1). At each basic block, a folding function is generated to fold the corresponding concrete register states into a global variable. A state-embedded program is synthesized by inserting the folding functions and embedding the concrete values of the global variables. Finally, the resulting program is used to validate the verifier; indeed, any failure to detect the sink indicates a logic bug in the verifier. Our evaluation results show that state embedding is highly effective. Within one month, we discovered 15 logic bugs exclusively in the verifier. This is a significant result considering that the verifier is primarily around 20,000 lines of code and, as aforementioned, has been partially verified. In addition, the verifier has gone through extensive security scrutiny and testing [7, 46]. Moreover, most of the bugs found are critical. For instance, two bugs are exploitable, where one allows users with `CAP_BPF` [41] to obtain root privilege and has existed for four years, and the other enables users with `CAP_PERFMON` [19] to obtain root privilege, both affecting kernel v5.10.33 and later.

State embedding significantly complements existing work. In comparison, our approach has several distinct advantages: (1) SEV treats the verifier as a grey-box rather than a black-box by inspecting the verifier states, which allows fine-grained detection of logic bugs; (2) by transforming and taking the state-embedded programs as input, the verifier automatically checks if the concrete states are contained, thus the approach requires little domain knowledge and is practical; (3) in general, one can easily embed rich concrete states, yet to detect all the sinks, the verifier must correctly collect approximations encompassing all the embedded states, thus being effective; and (4) state embedding can detect diverse logic bugs that lead to discrepancies between concrete states and approximations, *e.g.,* the bugs we found are located in various components including range analysis, stack access validation, *etc.* The key contributions of our work are:

- We propose state embedding, a novel and highly effective mechanism for detecting logic bugs in the eBPF verifier.
- We present SEV, a practical realization of state embedding, and apply it to stress test the eBPF verifier.
- We demonstrate state embedding's effectiveness by uncovering 15 previously unknown logic bugs in the eBPF verifier with 10 already fixed and many being critical.



Accepted Program

```
0: *(u64*) (r10 -40) = -1
1: r1 = *(u64*)(r10 -40)
2: r2 = 1
3: if r1 < 0 goto +1
4: r2 = 0
5: exit
```

State-embedded Program

```
0: r9 = 0
1: *(u64*) (r10 -40) = -1
2: r1 = *(u64*)(r10 -40)
3: r2 = 1
4: if r1 < 0 goto+1
5: r2 = 0
6: r9 += r1
7: r9 *= r2
8: if r9 != -1 goto+1
9: verifier_sink()
10: exit
```

POC

```
0: *(u64*) (r10 -40) = -1
1: r1 = *(u64*)(r10 -40)
2: r1 >>= 32
3: r1 &= 1
4: r1 += 1
5: r1 *= -8
6: *(u64*)(r10 - 8) = ptr
; use r1 control offset
7: call load_bytes
8: r2 = *(u64*)(r10-8)
```

Verifier States

```
0: fp-40 = 0xffffffff
1: r1 = 0xffffffff
2: r1 = 0
3: r1 = 0
4: r1 = 1
5: r1 = -8
6: fp-8 = ptr
...
7: fp-8 = ptr
8: r2 = ptr
```

Runtime States

```
0: fp-40 = 0xffffffffffffffff
1: r1 = 0xffffffffffffffff
2: r1 = 0xffffffff
3: r1 = 1
4: r1 = 2
5: r1 = -16
6: fp-8 = ptr
...
7: fp-8 = malicious_ptr
8: r2 = malicious_ptr
```

Figure 1: SEV performs state embedding by (1) profiling the concrete states of registers at each basic block, *e.g.,* R1 and R2; (2) folding R1 and R2 to R9 (#6 and #7); (3) embedding the concrete state of R9 (#8) and the verifier sink (#9). The instructions #8 and #9 implement the program construct and are the approximation-correctness check. The verifier interprets the if-condition #8 by validating if the runtime value -1 is within the approximation of R9, *i.e.,* determining if R9 could be -1, in which case the sink would be reported; otherwise, the verifier jumps from #8 to #10 and skips the sink. During validating the state-embedded program, the verifier skips the sink due to the logic bug. The root cause is sign information loss at #1. The POC program manipulates the state of R1 (#2 and #3), making the verifier believe R1 equals zero, whereas at runtime, it equals one. Consequently, the program overwrites the valid pointer stored on the stack with a malicious pointer (#6 and #7), thereby achieving arbitrary access.

2 Background and Illustrative Example

2.1 eBPF

eBPF is a register-based virtual machine that enables user space to extend the kernel dynamically. The user space first writes programs consisting of a sequence of eBPF instructions and loads the program with the `bpf()` system call. Programs operate on 11 registers (R0 to R10) and a fixed-size stack with four major types of instructions, namely load, store, arithmetic, and branch. An example program is presented in Figure 1. eBPF is adopted across different privilege levels, from unprivileged users [13] to those with certain capabilities [19, 41], and to fully privileged users. For instance, the `CAP_BPF` capability allows using eBPF with minimal privilege, widely applied in container scenarios. Consequently, the extensions are untrusted and a verifier is employed to validate their safety.

In a nutshell, the verifier traverses each execution path, interpreting every instruction in its abstract domains. Programs exhibiting any form of invalid behaviors, such as infinite loops and out-of-bounds access, are rejected. To strive for both soundness and precision, the verifier employs sophisticated algorithms to track program states. For instance, it gathers pointer types, register liveness, scalar ranges, *etc.* Scalar ranges are tracked using five abstract domains: four interval domains for different signs and bit-sizes, and the tristate number (`tnum`) domain [35] to model bit-wise operations. Scalar ranges are derived by combining information across these domains. The verifier models pointers based on region types and offsets, categorizing the former into more than twenty types, while tracking the latter using a variable. Furthermore, the verifier undergoes continuous updates by maintainers with new features and algorithms. The above features make the verifier the most complex component within the eBPF subsystem.

2.2 Illustrative Example

The key idea of our approach is to embed concrete states in programs such that when taking the state-embedded programs as input, we leverage the verifier to check whether the concrete states are contained in the approximation, thereby detecting logic bugs. In this section, we use an example to showcase how state embedding enables the detection of a subtle logic bug caused by a simple incorrect type cast. We also demonstrate the exploitation of the bug to highlight the significance of the verifier's correctness.

The Bug. The left segment of Figure 1 shows a valid eBPF program accepted by the verifier. Without effective test oracles, existing approaches would simply drop the case and proceed to the next iteration, since the program does not contain invalid behaviors. In comparison, SEV further utilizes state embedding to validate the verifier's approximation of the program, thereby uncovering this bug. Figure 2 shows the root cause and the patch proposed by us to fix the bug.

eBPF programs can spill registers or immediate values to the stack, and the verifier tracks the state of the stack accordingly. For the instruction `*(u64*)(r10-40)=-1` shown in the accepted program, the verifier marks the state of the accessed

```
diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index 857d76694517..44af69ce1301 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -4674,7 +4674,7 @@ static int check_stack_write_fixed_off(…)
        insn->imm != 0 && env->bpf_capable) {
      struct bpf_reg_state fake_reg = {};

-     __mark_reg_known(&fake_reg, (u32)insn->imm);
+     __mark_reg_known(&fake_reg, insn->imm);
      fake_reg.type = SCALAR_VALUE;
      save_register_state(state, spi, &fake_reg, size);
   } else if (reg && is_spillable_regtype(reg->type)) {
```

Figure 2: A logic bug detected by SEV. During spilling immediate values on the stack, the verifier incorrectly casts the `i32` immediate value to `u32` type, thus losing sign information. We proposed this patch to drop the cast, since `__mark_reg_known()` accepts `u64` type, the compiler would correctly promote integer type and propagate sign information. The patch has been accepted in the upstream and back-ported to the stable kernels.

stack slot as a known scalar value. However, during this process, the verifier incorrectly casts the immediate value from `i32` to `u32` and then sets the state of the stack slot, which is `u64`, to the casted value, causing lost sign information and the state of the stack slot being updated to an incorrect value. As depicted in Figure 1, when storing `-1` to the stack, the verifier incorrectly marks the corresponding stack slot as a value whose higher 32 bits are all zero, *i.e.,* losing sign information. Subsequently, when loading the same stack slot back, the verifier state of the destination register does not match the original register, *i.e.,* the concrete value `-1` is not contained in the approximation. Figure 2 demonstrates the patch to fix the bug, which has been merged to the upstream and back-ported to the stable kernels, given its security impact.

In general, detecting logic bugs in the verifier poses significant challenges due to the following characteristics:

- **Hard to notice:** The abstract domains utilized by the verifier are complex and challenging to comprehend thoroughly, and the cyclomatic complexity of its tracking logic is high. To pinpoint logic bugs, one needs to precisely understand the verifier states and inspect them following the tracking logic. Conducting such a process is difficult, *e.g.,* the aforementioned incorrect type cast is likely to be overlooked.

- **Hard to detect:** Existing work treats the program under testing as a black-box, yet logic bugs of the verifier are likely to be silent errors. For instance, the behavior of the program demonstrated is correct, which simply accesses the stack within bounds and operates the registers, and in this sense, the verifier's conclusion seems justified. However, as shown by our approach, the verifier's approximation contains a subtle flaw, which is challenging for existing approaches to detect.

State Embedding. To enforce the safety of the program, the verifier must track the over-approximation of program states on each execution path, yet a concrete state not contained in the approximation can invalidate the conclusion. Based on the observation, our approach systematically transforms the program to embed concrete states within certain program constructs and utilizes the verifier to validate whether the aforementioned property holds during the checking process.

Step 1: The first step of state embedding is to profile concrete states, as illustrated in Figure 1. For a program accepted by the verifier, we execute and profile concrete states at each basic block of the execution path. Since the if-condition at #3 holds at runtime, the program jumps to #5, where we collect the concrete states of registers. The concrete states of R1 and R2 must be contained in the verifier's approximation.

Step 2: The second step performs state embedding. In practice, the collected state information is rich, and directly embedding each state makes the verifier fork and explore paths multiple times, potentially leading to redundant checks. We adopt folding to efficiently conduct state embedding. First, we initialize R9 (#0), a reserved register that holds the folded concrete state. Then, we generate a folding function at each basic block, which consists of arithmetic instructions that fold the collected states into the single register R9. The folding function in Figure 1 has two instructions #6 and #7, which fold R1 and R2 into R9. The value of R9 is calculated during the generation by evaluating those arithmetic instructions with the concrete states, which, in the example, equals `-1`. Finally, the instruction #8 and #9 implement the program construct, which embeds the folded concrete state with the if-condition instruction that compares R9 with its value `-1` and the verifier sink. The condition makes the verifier compare the folded state with its approximation of R9. The sink would be detected if the verifier deems `-1` is within the approximation.

Step 3: Subsequently, we can take the state-embedded program as input to the verifier to detect the logic bug. During the checking process, the verifier initially collects the program states from #0 to #4, where it incorrectly tracks the approximation of R1 (#1 and #2). Such an issue would be captured as the concrete state of R1 is folded into R9. Since the verifier determines `r1<0` not hold, it proceeds from #4 to #5. At #6 and #7, the verifier folds R1 and R2 into R9, where it erroneously concludes that the only concrete value within the approximation of R9 is zero. Therefore, the folded concrete state `-1` would not be contained in the approximation of R9, causing the verifier to skip the sink at #9. Consequently, we have found the logic bug in the verifier.

Our approach embeds concrete states within certain program constructs and utilizes the verifier to validate the approximations. State embedding provides two distinct advantages:

- **Fine-grained:** State embedding views the verifier as a grey-box rather than a black-box since it performs fine-grained validation on the verifier states. For the aforementioned example, which is overlooked by the existing approaches and even by the experienced maintainers, our approach further validates the approximations with the profiled concrete states.

- **Practical:** Our approach validates if the aforementioned property holds by embedding concrete states and utilizing the verifier to compare the states against the approximations. Therefore, state embedding requires little domain knowledge and is more practical.

The Exploit. The right segment of Figure 1 depicts a proof-of-concept (POC) that exploits the bug to obtain root privilege. In essence, the POC uses the bug to manipulate the verifier's knowledge about the program. Since the verifier incorrectly believes that the higher 32 bits of R1 are zero, which are in fact all one at runtime, the POC first constructs an evil register by right-shift and logic AND operations (#2 and #3). These operations make the verifier conclude R1 equals zero, while at runtime it equals one. Then, the POC stores a valid pointer on the stack and invokes a helper function that allows eBPF programs to load user space data to their stack. By using the evil register as the length parameter, the POC makes the verifier think that only 8 bytes are stored on the stack, while in fact it stores 16 bytes, thus overwriting the pointer with the user-controlled pointer. Finally, the POC achieves arbitrary access with the malicious pointer, while the verifier erroneously believes the program is operating the original valid pointer.

Since eBPF programs are executed in kernel space, the POC can perform various malicious operations, *e.g.,* overwriting credentials of the current task struct for privilege escalation. Our full POC enables users with `CAP_BPF` to achieve root access, and kernels v5.10.33 and later are affected. After we submitted the patches that fix the bug, we also received positive feedback from the maintainers of the eBPF subsystem:

> *"...I owe you a big thanks as well since this helps with our internal process. So thank you in advance!"*

It is important to highlight that the aforementioned exploit accomplishes all the malicious operations through the subtle bug, a simple incorrect type cast in one line of the verifier's code. In addition to the presented bug, we also found another exploitable bug allowing users with `CAP_PERFMON` to obtain root privilege, arising from incorrect tracking of memory accesses with variable offsets. The capability mechanism in Linux grants users the minimum privileges for specialized tasks, yet these bugs undermine this rule, posing significant security concerns. Notably, in the POC programs for both bugs, the number of instructions used to manipulate the verifier's knowledge is less than ten, further demonstrating the severity of the bugs found. This illustrates that our approach can detect critical logic bugs by leveraging the verifier to perform fine-grained validations on its approximations.

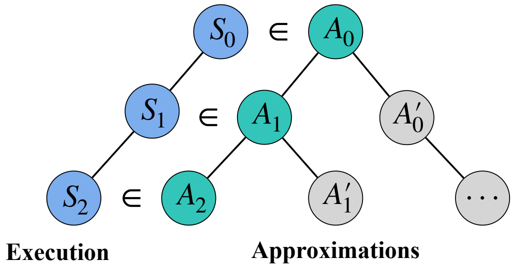


Figure 3: The verifier tracks program states on each possible path as shown in the right part, and the runtime execution corresponds to one of the paths. Each concrete state S_i must be contained in the approximation A_i; otherwise, operations on the non-contained states could be unsafe, *i.e.,* a logic bug.

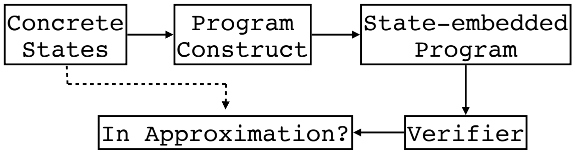


Figure 4: Our approach validates whether concrete states are contained in the approximation for logic bug detection by (1) embedding the concrete states in the program with the construct; (2) taking the state-embedded program as input to the verifier; and (3) leveraging the verifier to validate the states against the approximation.

3 State Embedding and SEV

In this section, we introduce state embedding, a folded variant, and describe our implementation of SEV.

3.1 State Embedding

State embedding is based on one fundamental observation: the concrete states must be contained in the corresponding approximations of the verifier, as shown in Figure 3. The goal of state embedding is to validate if this property holds during validation. As depicted in Figure 4, to achieve this, the approach executes and profiles an accepted program P, lacking inputs and external interactions, to collect concrete states, which remain consistent across multiple executions. Since those states are profiled from a real execution, they establish the ground truth that the corresponding approximations must properly contain them. Next, P is transformed to P' by embedding sinks that contradict those observed states. When taking P' as input, the verifier, following the same path as the real execution, compares the embedded concrete states against the approximations, thereby automatically validating its own correctness.

At the conceptual level, given a program P, a concrete state S, and the variable A holding the state (the approximation from the verifier's perspective), a state-embedded program P'

is synthesized by embedding the following program construct at the corresponding program location:

$$\textbf{if } S \in A \textbf{ then } \mathit{verifier\_sink}()$$

The $\in$ in the program construct is an abstraction of the operators in the program, for which the verifier interprets to check whether the concrete state S is contained in the approximation A or not. The $\mathit{verifier\_sink}()$ represents the operations, where the verifier reports errors. By embedding the program construct and taking P' as input, the verifier interprets the construct to check if the embedded state S is within A properly. The verifier sink in the construct is an indicator for the containment of the concrete state S.

Proposition 3.1 *State embedding does not introduce any invalid operations except for the sink.*

The aforementioned program construct does not introduce incorrect operations to the original accepted program except for the verifier sink. When interpreting the embedded construct, the verifier state A_i would be split into two states A_j and A_k, where the former follows the branch-taking path of the if-condition thus detecting the sink, and the latter continues the original execution path. Since the original program is considered safe under the approximation of A_i and $A_i = A_j \cup A_k$, *i.e.,* A_i is a superset of both A_j and A_k, the program is also considered safe with A_k. Therefore, the sink is the only expected error for a state-embedded program.

Corollary 3.1 *Failure to detect the sink indicates a logic bug in the verifier.*

More concretely, for a verifier V: $\mathcal{P} \rightarrow \{safe, unsafe\}$, since the original program P is considered safe by the verifier, $V(P') = unsafe$ with the sink being reported implies that the verifier correctly deems S is within the approximation A; on the contrary, $V(P') = safe$, *i.e.,* failure to detect the sink, indicates the concrete state S being missed from the approximation A, *i.e.,* a verifier logic bug.

Here we demonstrate the semantics of each part in the program construct with an example. In general, the verifier interprets concrete operators in the program in its abstract domain. Many concrete operators are available to implement the abstraction $\in$. For instance, as shown in Listing 1, one can utilize the if-statement with an == comparison to realize the construct; other comparisons, such as greater-equal (>=) or less-equal (<=), are also feasible, as per the implementation. The instruction writing to the read-only register R10 is an instance of the verifier sink in eBPF programs, and assertion failure can be used in other scenarios. After embedding, the verifier interprets the if-statement by validating if S is within the approximation of A to determine if the branch should be taken, in which case the sink would be reported.

```
...original program
The verifier checks if S is within A, when
interpreting the following if-statement.
if (S == A)
    verifier_sink();
...
```

Listing 1: Example of the program construct.

Folded Variant. A straightforward realization of state embedding is by profiling the program to collect concrete states and subsequently embedding the states with the corresponding program construct multiple times. However, directly embedding each state encounters two challenges: (1) the verifier may halt early at the first detected sink, leaving other sinks unchecked; and (2) the inserted sinks make the verifier fork exploring paths, introducing redundant checks. We propose a folded variant to embed the states in conjunction with one sink efficiently. For each concrete state in an execution:

$$S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_{n-1} \rightarrow S_n$$

we do not directly embed each S_i into the program. Instead, we generate a folding function f_i for each of them, which consists of simple computation operations, *e.g.,* ALU operations in eBPF programs, and hold the folded concrete state $\widehat{S}$ in a global variable $\widehat{A}$:

$$\widehat{S_1} = f_1(\widehat{S_0},\ S_1) \rightarrow \dots \rightarrow\ \widehat{S_n} = f_n(\widehat{S}_{n-1},\ S_n)$$

Finally, we embed the folded state $\widehat{S_n}$ once with the following program construct:

$$\textbf{if } \widehat{S_n} \in \widehat{A} \textbf{ then } \mathit{verifier\_sink}()$$

Since only the folded state is embedded, the verifier forks once. The bug-detecting capability of this variant is equivalent to the original form since incorrect approximations are likely to be propagated to the approximation of the global variable during the continuous testing campaign. The variant provides further benefits, *e.g.,* to detect the sink, the verifier needs to not only correctly track states for the original program, but also properly simulate the folding functions.

3.2 The SEV Implementation

We applied state embedding to detect logic bugs in the eBPF verifier. Algorithm 1 illustrates the major workflow. In each testing iteration, SEV first utilizes the program generator that we developed to obtain eBPF programs accepted by the verifier (lines 4-6), and then executes and profiles the program to collect the register states at each basic block (line 7). We follow established compiler testing methodologies [31] by utilizing deterministic, closed eBPF programs that require only a single round of profiling and the profiled states are consistent across executions. Based on the concrete states,

Algorithm 1: Workflow of SEV

```
1  Procedure Validate():
2      $LogicBugs \leftarrow \emptyset$
3      while not terminate do
4          $P \leftarrow$ NextProg()
5          if Verify($P$) = reject then
6              continue
           // Profile states at basic block
7          $S \leftarrow$ Profile($P$)
           // Embed the concrete states
8          $P' \leftarrow$ StateEmbedding($P$, $S$)
           // Validate the eBPF verifier
9          if Verify($P'$) = accept then
10             $LogicBugs \leftarrow LogicBugs \cup P'$
```

SEV transforms the program to inject folding functions at each basic block and embed the folded concrete state (line 8). Finally, the state-embedded program is used as input to the verifier (line 9), and a logic bug is detected if the program is accepted (lines 10-11), *i.e.,* the verifier is incapable of detecting the sink. The aforementioned loop continuously tests the verifier with state-embedded programs.

SEV first needs to obtain eBPF programs that can pass the verifier. We devise a program generator to facilitate state embedding, adhering to the established program generation approaches [50]. First, we ensure the instruction encoding and the control flow of a generated program are valid by following the instruction specification [6] and representing a program as a structured graph, thereby avoiding being rejected early. Next, we synthesize the program by combining several basic structures, *e.g.,* if-else block and back-edge, and leveraging lightweight global state information, such as register and stack slot states, to generate instructions reflecting realistic usage patterns. We categorize a register state into several types, including uninitialized, scalar value, and pointer, and synthesize operations accordingly, *e.g.,* generating pointer accesses or offset operations if a register stores a pointer. The generator continuously provides programs for the testing campaign.

We implemented a tracer based on the existing kernel infrastructures to profile programs. The tracer intercepts the execution of programs at each basic block and captures the instruction index and the register states, which are appended into an internal state buffer. The tracer interface is exposed via a virtual device so that the user space can utilize the functionality flexibly and access the buffer via `mmap()` for shared memory. For each accepted program, SEV executes it with the tracer enabled and decodes the buffer in user space.

Algorithm 2 presents SEV's implementation of state embedding. The inputs are the accepted program and the concrete states at each basic block collected with the tracer. We first initialize the folded state (line 2), and R9 is reserved to ensure its availability for storing this value. The algorithm maintains a map associating each basic block with its folding function (line 3). The collected states are essentially a basic block trace in conjunction with the states, where the basic block could appear multiple times due to loops. By using the map, the algorithm ensures the folding function is generated once for each basic block. The folding function is generated by synthesizing various eBPF arithmetic instructions for each collected register (lines 6-8). We consider all the ALU instructions the verifier can accurately track, excluding the division and all unary operations. Folding functions are generated by randomly selecting those operations applied to non-zero registers, preventing the state from being easily reduced to zero. The folded state is updated by calculating the folding function with the concrete states (line 9). Finally, the state-embedded program is generated by inserting each folding function in the corresponding basic block and embedding the folded state with the instructions (lines 10-11), as shown in Listing 2.

Algorithm 2: State Embedding

```
1  Function StateEmbedding(Program P, States S):
2      $FoldedState \leftarrow$ Initialize()
       // Basic block to folding function map
3      $FoldingFns \leftarrow \emptyset$
4      foreach $BB$, $Regs \in S$ do
5          $F \leftarrow FoldingFns[BB]$
6          if $F$ not exists then
               // Generate folding function
7              foreach $Reg \in Regs$ do
8                  $F \leftarrow$ GenALU($F$, $Reg$)
           // Update Folded State
9          $FoldedState \leftarrow F(Regs, FoldedState)$
       // Insert the folding functions
10     $P' \leftarrow$ InsertFoldingFns($P$, $FoldingFns$)
       // Embed the folded state
11     $P' \leftarrow$ EmbedFoldedState($P'$, $FoldedState$)
12     return $P'$
```

```
if r9 != FoldedState goto+1
r10 = 0
```

Listing 2: The instructions used for embedding the folded state and the verifier sink in eBPF programs.

The first instruction in Listing 2 makes the verifier validate whether the folded state is contained in the approximation of R9. The second instruction is the verifier sink, an illegal operation where the verifier reports a "writing to the read-only register R10" error. The sink is skipped if the folded state is not contained, indicating a logic bug. The state-embedded program can thus be used for validating the verifier.

4 Evaluation

In this section, we evaluate the effectiveness of state embedding by applying SEV to detect logic bugs in the eBPF verifier. Highlights of our results are as follows:

- **Considerable bugs:** We have found 15 previously unknown logic bugs solely in the eBPF verifier.
- **Diverse bug types:** The root causes of the bugs are various and located in different components of the verifier.
- **Critical severity:** Most of the bugs found by SEV are critical, posing various security implications.

We believe that the quantity and quality of the bugs found by our prototype SEV have demonstrated the effectiveness of state embedding in uncovering logic bugs in the verifier.

4.1 Evaluation Setup

Environment. All the bug-finding experiments were conducted on a Linux server with a 64-core AMD Ryzen Threadripper 3990X Processor, where each core has two threads, and the memory size of the server is 256 GiB. The version of the host Linux kernel is v5.15. To ensure that these experiments did not affect the host system, we conducted our testing within multiple virtual machine instances. These instances were created using QEMU version 6.2.0, with KVM employed to provide acceleration. The guest environment in each instance consisted of a minimal Debian distribution disk image, and the system was booted using the compiled kernels. In addition to incorporating common kernel configurations, we also enabled the eBPF subsystem-related options [5].

Kernel Version. We chose the eBPF upstream repository for testing, and the reasons are as follows: (1) the uncovered logic bugs in the upstream are likely to be previously known, and thus should be fixed immediately; (2) testing upstream enables the detection of bugs that may impact various past stable versions; and (3) testing the upstream kernel prevents newly introduced bugs being merged into subsequent releases. In addition to the built configuration previously mentioned, we also patched the kernel and enabled related options since we modified the eBPF interpreter `bpf/core.c` to intercept the execution of eBPF programs for state tracing.

Testing Process. SEV is designed to automatically execute the entire testing process. Initially, after configuring the disk image and the kernel for testing, SEV determines the appropriate QEMU command line and subsequently initiates the virtual machine using the specified kernel. Upon successful booting of the virtual machine, SEV proceeds to the testing phase. This involves the generation of eBPF programs and the validation of the verifier using state-embedded programs. A shared directory is established to transfer the testing results between the host and the guest. Subsequently, SEV checks the liveness of the virtual machine and restarts the whole campaign if it detects the system is not alive. We utilized SEV to test the eBPF upstream with the aforementioned testing process for one month.

Bug Triage. We triage and deduplicate all the bugs found based on their root causes. In principle, any failure to detect the sink in the state-embedded programs indicates a logic bug in the verifier. When SEV reports such a case, we further inspect it to locate the root cause. During the testing campaign, SEV retains the original program, the captured runtime states, and the corresponding state-embedded program when a logic bug is identified. We inspect the programs to pinpoint the instruction where the approximation of the verifier mismatches the runtime states, and then analyze the preceding instructions to collect related operations that produce the operands for the culprit instruction. The culprit instruction in conjunction with related operations enables us to locate the incorrect verifier logic. Finally, one can look into those parts of the verifier and analyze the root cause.

4.2 Quantitative Results

Bug Number. We applied SEV to test the eBPF verifier for one month and triaged the discovered bugs based on their root causes as mentioned in Section 4.1. Note that we only reported unique bugs to the eBPF mailing list, and only bugs with different root causes are counted in our evaluation. As a result, we have found 15 unique logic bugs in the eBPF verifier within one month, of which 10 have been fixed at the time of paper submission. The number of found bugs is significant considering: (1) the codebase of the eBPF verifier is relatively small compared to other subsystems, *e.g.,* it mainly contains 20,000 lines of code; (2) the verifier has undergone thorough security scrutiny by the community and is one of the most extensively tested components in the kernel, *e.g.,* eBPF self-tests [7] contain a large set of programs covering different corner cases to test the verifier; and (3) previous research efforts have applied verification on parts of the eBPF verifier [44, 45, 52]. The aforementioned results demonstrate that our approach is highly effective in uncovering logic bugs.

Furthermore, compared to manual testing, which requires substantial effort to keep up with the development of eBPF, our approach can continuously test the eBPF verifier along with its frequent updates and modifications. Existing testing tools like Syzkaller can uncover memory issues in the eBPF subsystem, yet they encounter difficulties in logic bug detection due to the lack of effective test oracles. While formal verification provides a strong guarantee, synthesizing specifications for the verifier is inherently complex, whereas SEV utilizes state embedding to automatically uncover logic bugs in the verifier without requiring specifications. Therefore, state embedding uniquely complements existing approaches.

Table 1: Number of bugs found by SEV in different locations.

Bug Location	Range Analysis	Memory Access	State Prune	CFG Check	Other
#	6	3	2	2	2

Bug Types. In general, logic bugs in the verifier can be classified into two categories: (1) incorrectly accepting unsafe programs, *i.e.,* soundness bugs, and (2) incorrectly rejecting safe, correct programs, *i.e.,* completeness bugs. The former introduces potential security issues, *e.g.,* allowing malicious programs to be loaded, while the latter forces developers to heavily refactor their programs to mitigate the verifier's imprecision. Overall, we have found 12 soundness bugs and three completeness bugs. Table 1 shows the bug details.

Our approach can detect various soundness bugs in the verifier for the following reasons. eBPF supports four major types of instructions, including load, store, ALU, and jump operations. For ALU and jump, the eBPF verifier mainly performs range analysis and pointer arithmetic checks. Since these operations are conducted on the registers, logic bugs in those components mainly result in incorrect approximation of register states. SEV can directly detect those logic bugs because our approach validates if the concrete states are contained in the approximation. For example, six of the bugs found are related to range analysis. For logic bugs in load and store checks, although we do not profile all the memory accessible to eBPF programs, SEV can still detect bugs in these areas because incorrect tracking on those states can be propagated to the approximation of registers. For instance, three of the bugs found are related to memory access validation, including the incorrect stack spill checking illustrated in Figure 1. Furthermore, we also uncovered logic bugs in other components, *e.g.,* the state pruning procedure.

SEV can also uncover completeness bugs as an additional design benefit. As detailed in Section 3, state embedding does not introduce any incorrect operations except for the sink. If the verifier rejects state-embedded programs for reasons other than the sink, it indicates the presence of a completeness bug. More broadly, the idea of state embedding can be specifically tailored for completeness bug detection. For example, replacing the sink with valid operations in the transformed programs would ensure their correctness, thereby highlighting any inaccuracies in the verifier's rejection of these programs. Completeness bugs not only interfere with development but also reflect potential implementation issues in the verifier. SEV uncovered three such bugs, where two are related to control flow graph checking, and the other is inconsistent stack access validation. While these bugs may not directly pose security issues, they significantly impact the usability of eBPF. For instance, one bug we discovered causes the verifier to erroneously reject a set of programs due to a specific control flow pattern, despite these programs being correct.

These results underscore the effectiveness of state embedding in identifying a diverse array of logic bugs, and generally, state embedding can detect logic bugs that result in a divergence between the approximations and the concrete states.

Bug Impact. Logic bugs within the eBPF verifier hold critical implications. All of the 15 bugs found by SEV are located within the verifier, specifically in `verifier.c`, and most of them are critical. We have demonstrated that two of the found bugs are exploitable, and each can be exploited to achieve local privilege escalation. Beyond these, other uncovered bugs have diverse implications. For example, some bugs in the range analysis can circumvent the verifier's enforcement that the return values of certain programs must be within specified ranges, thereby potentially affecting the caller. The bugs in the state pruning procedure can be used to load programs containing infinite loops, leading to system hangs.

4.3 Assorted Bug Samples Found by SEV

To further demonstrate the characteristics of the uncovered bugs, we highlight several examples in this section.

Figure 5a: The range analysis is an important component of the verifier since it is the foundation for various safety checks, *e.g.,* memory access check and control flow validation. Figure 5a shows a logic bug found by SEV in the range analysis, where the verifier incorrectly tracks registers' states after simulating the fall-through path of the branch condition. The root cause is the verifier's inability to correctly handle the `JSLE` instruction when comparing a range with a non-overlapping constant. More concretely, R9 is initialized at first and updated subsequently, and the range of R8 and the value of R4 are non-overlap at #6, after which the verifier incorrectly marks R8 and R9 as constant values. After the arithmetic operations (#7 and #8), the runtime value of R9 is one at #9, which differs from the verifier's approximation, a constant zero. The bug can break the verifier's restriction. For instance, the verifier enforces that the return value of certain program types can only be zero to not modify the caller states, yet the bug leads to programs with arbitrary return values being loaded. After we reported the bug, the maintainers also enhanced the return value-checking logic in the verifier.

Figure 5b: The verifier allows privileged users to load programs with back-edges and detects and rejects infinite loops during checking. SEV uncovered a logic bug in the verifier that incorrectly rejects programs with well-defined structures. For the program shown in Figure 5b, the verifier rejects it, reporting an incorrect back-edge from #3 to #4. However, such an error is not accurate, since #3 to #4 is not a back-edge. Furthermore, the behavior of the program is correct as the loop in the program is bounded. The root cause is that the control flow-checking procedure of the verifier is incapable of handling the structure pattern of the program. The bug

```
0: (b7) r9 = -2                      ; R9=-2
1: (37) r9 /= 1                      ; R9=scalar()
2: (bf) r8 = r9                      ; R9=scalar(id=1) R8_w=scalar(id=1)
3: (56) if w8 != 0xfffffffe goto pc+4 ; R8=scalar(var_off=(0xfffffffe;
                                          0xffffffff00000000))
4: (65) if r8 s> 0xd goto pc+3       ; R8=scalar(smax=13)
5: (b7) r4 = 2                       ; R4=2
6: (dd) if r8 s<= r4 goto pc+1       ; R4=2 R8_w=0xfffffffe
7: (cc) w8 s>>= w9                   ; R9=0xfffffffe R8=scalar()
8: (77) r9 >>= 32                    ; R9=0
9: (57) r9 &= 1                      ; R9=0
10: (95) exit
```

(a) A bug in the verifier's range analysis.

```
0: (b7) r4 = 0x35
1: (b7) r8 = r4
2: (05) goto+2
3: (1f) r9 -= r4
4: (1f) r9 -= r8
5: (0f) r8 += r4
6: (a6) if r8 < 0x64 goto-4
7: (bf) r0 = r9
8: (95) exit
```

(b) A bug in the CFG checking.

```
0: (bf) r0 = r10               ; R0=fp0
3: (18) r5 = 0x1d00000025      ; R5= 0x1d00000025
2: (bc) w9 = w0                ; R9=scalar(var_off=(0x0; 0xffffffff)
3: (47) r9 |= -12              ; R9=scalar(var_off=(…; 0xb))
4: (0f) r9 += r0               ; R9=fp(off=0, u32_min=-12)
5: (76) if w5 s>= 0xfffffff6 goto pc+16  ; R5=0x1d00000025
6: (72) *(u8 *)(r9 -221) = -19   ; stack_depth=221
7: (95) exit
```

(c) A bug in the stack depth tacking.

```
0: (18) r4 = map_ptr
1: (18) r1 = 0x1d
2: (55) if r4 != 0x0 goto pc+4
3: (1c) w1 -= w1
4: (18) r9 = 0x32
   (00) reserved_code
5: (56) if w9 != 0xfffffff4 goto pc-2
6: (95) exit
```

(d) A bug in the jump target checking.

Figure 5: Assorted eBPF programs that trigger logic bugs. The left part of each sub-figure shows the eBPF instructions and the content after each semicolon presents the verifier's approximations, illustrating the bug cause. `scalar()` and `fp()` show that the tracked value is a scalar and stack pointer, `var_off()` is the `tnum` domain, and `stack_depth` is the tracked depth of used stack.

causes all the correct programs with this structure pattern to be rejected, thus requiring heavy code refactoring to mitigate the verifier's bug. The inserted folding functions triggered the bug, demonstrating the additional advantages of the folded variant of state embedding. The bug has been fixed and the patch was back-ported to the stable kernels.

Figure 5c: The verifier tracks the stack depth of the program and uses this information to subsequently allocate the stack area before execution, thus the correctness of the calculated stack depth is important. However, SEV uncovered that the verifier incorrectly overlooks the variable offset in stack accesses, leading to the collected stack depth being smaller than the size that the program may access at runtime. As illustrated in Figure 5c, at first, R9 is a scalar with a minimum value of -12, and the tracked range information is correct. The instruction #4 adds the stack pointer to R9, thereby making R9 a stack pointer with a variable offset. The verifier incorrectly marks the stack depth of the program as 221 at the stack writing instruction #6 without considering the variable offset of R9, *i.e.,* its possible minimum value. The bug has existed for four years and is exploitable for privilege escalation.

Figure 5d: Most instructions of eBPF adopt basic encoding with an eight-byte length, while a special kind of load instruction uses the wide instruction encoding and appends a second eight-byte immediate, the code of the second part is the reserved code. The target of jump instructions in eBPF programs must be within bounds and be a valid instruction. However, SEV detected a logic bug in the verifier that reports programs containing invalid jump targets with an incorrect reason. As depicted in the figure, the instruction #4 is a special load that uses the wide instruction encoding and its second part contains the reserved code. The program incorrectly jumps from #5 to the second part of #4, *i.e.,* jumping into the reserved code, yet the verifier incorrectly reports that the program contains an invalid load instruction. The root cause of the bug is that the verifier overlooks the special case while checking the jump target.

```
0: (18) r9 = 0x00000018         ; R9= 0x18
1: (85) call get_cgroup_id#123 ; R0=scalar()
2: (5c) w9 &= w0                ; R9=var_off(…;0x18)
3: (b5) if r0 <= 0xfffffffb goto pc+3
                                ; R0=var_off(…;0x3)
4: (5d) if r9 != r0 goto pc+2   ; R0=-4 R9=-4
5: (c7) r9 s>>= 23              ; R9=-1
6: (95) exit
```

Figure 6: A logic bug in the verifier.

Figure 6: SEV also uncovered that the range analysis of the verifier is incapable of correctly handling equality comparison when the ranges of two operands do not overlap. This issue is exemplified in Figure 6. At #2 the mask of R9 is 0x18, and the verifier determines all the bits of R9 are known to be zero except for the fourth, and the fifth bit (unknown). The mask of R3 is 0x3, where only the lower two bits are unknown and all the other bits are one, and thus the range of R9 and R3 are non-overlap. However, the verifier erroneously assigns both R0 and R9 as -4 after the if-condition with equality comparison at #4, where the runtime value of R9 consistently remains 0x18, *i.e.,* a logic bug in the verifier.

4.4 Throughput Impact

SEV synthesizes state-embedded programs by profiling runtime states using the tracer mentioned in Section 3 and subsequently integrating the generated folding functions and embedding the folded concrete state. In this section, we evaluate the performance impact of SEV on three key aspects: the influence of profiling on program execution speed, the effect of state embedding on the verification time, and the overall impact on testing throughput.

The evaluation procedure is as follows. First, SEV is utilized to continuously generate programs. For accepted programs, we execute them both with and without the tracer and compare the execution times in both scenarios to determine the impact of the profiling. Subsequently, we embed the concrete states and measure the difference in verification time between the state-embedded program and the original program. To account for any nondeterministic factors, the above process is repeated multiple times for each program. Finally, to ascertain the impact on testing throughput, we run SEV with and without the profiling and state embedding. All the comparisons are carried out over a 24-hour period, and the average results are compiled and reported.

The evaluation results show that the average impact of profiling on program execution is 5.3%. This relatively minor impact is primarily due to the profiling being performed at the basic block level, where the tracer efficiently appends the states to a shared buffer. In terms of the verification time, the impact of state embedding results in an average increase of 17.2%. The minimal complexity added to the original program by state embedding, which involves arithmetic operations and a single embedding of the sink, contributes to this increase. Notably, the overall impact on testing throughput is only 1.6%. This lower impact, compared to program execution and verification time, is because state embedding is conducted only after programs are accepted, an infrequent event. In addition, the embedding constitutes a small part of the entire testing campaign, which also includes the generation, test case persistence, *etc.* In summary, we conclude that state embedding imposes a reasonable overhead and the impact is well within expectations, given its ability to uncover logic bugs.

4.5 Discussion

Coverage Impact. Coverage in our context involves two aspects: (1) in the generated programs (raw coverage), and (2) in the verifier (induced coverage), where the former may affect the profiling stage and the latter is related to the testing sufficiency. To ensure a stable raw coverage, we adopt closed eBPF programs as mentioned in Section 3.2 that require only a single round of profiling. For the induced coverage, we cover the verifier's functionality with the program generator, and optimizing the induced coverage is orthogonal to this work.

False Positives/False Negatives. As illustrated in Section 3, in principle, state embedding does not introduce any invalid operations to the original program except for the embedded sink. In practice, our approach has not resulted in any false positives, *e.g.,* the found bugs pose certain security implications. Being a testing technique, our approach can suffer from false negatives. The major reason is that state embedding requires programs accepted by the verifier, yet the generator may not be able to explore a diverse, thorough search space. Our main goal in this work is to detect a wide range of logic bugs with the principled idea, *i.e.,* concrete states being contained in the approximation. In addition, the inserted arithmetic instructions may hinder some non-contained states, albeit with a low likelihood. SEV operates within a continuous testing loop, which therefore inherently enhances the detection of such anomalies over time, even if a specific combination of state values momentarily evades detection.

Verifier Changes. The eBPF subsystem is undergoing continuous updates, incorporating new features as it develops. Despite these changes, state embedding remains widely applicable and is not limited to specific static checkers. Our implementation, importantly, does not depend on the internal workings of the verifier. This is because SEV mainly transforms the accepted programs to embed concrete states, after which the program is delivered to the verifier for validation. Changes within the verifier, such as new abstract domains, primarily affect how the state-embedded program is validated, while the transformation remains unaffected.

State Pruning. The state pruning procedure [12] in the verifier evaluates the current state against the known safe states to determine redundancy, thereby pruning explored paths. However, this technique cannot be applied to eliminate extra paths for direct embedding. When comparing states, the verifier performs a detailed analysis of registers marked as precise. The registers of inserted sinks are marked precise, yet their value ranges vary between the branch-taken and fall-through paths, making them unprunable. Folding addresses this by embedding the folded state once at the end of the program.

5 Related Work

eBPF Verification. Existing work [16] applies formal verification to several components of the eBPF verifier, significantly advancing the verifier's correctness. For instance, recent efforts [52] have proved the soundness of the implementation of the tristate number in the verifier, and we did not detect any bugs in the corresponding location, *i.e.,* `tnum.c` in the kernel. However, `tnum` is a small component of the verifier, which mainly contains 200 lines of code. Agni [45] is more ambitious, aiming to verify the range analysis of the verifier, which consists of 2,100 lines of code. The tool automatically converts the C source code to SMT formulas and utilizes SMT solvers to detect discrepancies between the specification and the implementation. The work concludes that the tool proved the soundness of range analysis in Linux v5.19, the latest version when the work was conducted. Beyond reasoning about the correctness of the verifier, Jitterbug [34] applies automated verification on the eBPF JIT compiler.

Nevertheless, devising specifications for the eBPF verifier automatically or manually is challenging and requires deep domain knowledge. Consequently, the synthesized specifications can be incomplete or inconsistent with the implementation. For example, while Agni aims to perform verification on the verifier's range analysis, the generated verification conditions are incomplete [4, 9, 10, 14]. Indeed, we identified logic bugs in this component within the same kernel version. In comparison, state embedding has the following unique advantages: (1) our approach does not require predefined specifications, but utilizes state embedding to perform fine-grained checks for logic bug detection; (2) state embedding is capable of testing various components not just the range analysis, *e.g.,* SEV also uncovered bugs in the memory access validation; and (3) since we leverage the verifier to check whether the concrete states are contained in the approximation, our approach is agnostic to the verifier's internal and can be applied along with its fast changes. Therefore, our approach significantly complements the existing work.

Static Analyzer Testing. Some work [27, 53] proposes to detect bugs in static analyzers by collecting the information of both analyzers and programs and directly comparing them. For example, Wu et al. [49] propose to profile pointer alias at runtime and compare the information directly with the knowledge from the alias analysis algorithm. Similarly, Buzzer [26] extracts the verifier log, conducts an offline comparison between the log information and the collected map value, and uncovered one logic bug. However, this direct comparison approach requires: (1) collecting and parsing both the runtime information and the analyzers' states, and (2) correctly conducting the comparison, which requires substantial domain knowledge to interpret the verifier states and is coupled with concrete implementations. In comparison, SEV only profiles program states at each basic block and utilizes state-embedded programs to enable the verifier to conduct the validation, thus being efficient and agnostic to verifiers. In addition, Bugariu et al. [17] propose to test abstract domains by interactively invoking operations and using the mathematical properties of the domains as test oracles. However, applying this approach in the eBPF verifier is challenging: (1) the verifier is integrated into the kernel and does not support interactively invoking its internal operations, and (2) the eBPF verifier does not provide a specification and mixes up the abstract operators in one domain with the refining operations [45], thus the mathematical properties summarized are not directly applicable in this context.

Differential Testing. α-Diff [29] conducts differential testing between several static analyzers to identify logic bugs in them. However, this approach hinges on the precondition that the precision of static checkers is comparably high and that reference implementations are well-established. Except for the verifier in Linux, Gershuni et al. [22] propose a potential reference verifier. Yet, it lacks support for some important features, *e.g.,* various modes of basic ALU instructions, and experiences precision issues [11]. Therefore, using the tool for differential testing would be ineffective in detecting logic bugs in the kernel verifier. Furthermore, our approach differs significantly from differential testing methods and is an instance of metamorphic testing. α-Diff relies on multiple checkers and does not provide ground truth for each variant it produces. In contrast, state-embedded programs contain the ground truth by construction, eliminating the need for other reference verifiers.

Assertion Generation. Existing work aims to automatically generate assert statements to detect logic bugs [48, 51], establishing input-output relationships using program analysis or deep learning. In contrast, our approach does not aim to assert input-output relations, which is nontrivial for the verifier's logic bug detection, but to validate a core property: the concrete states must align with the verifier's approximations. It embeds concrete states into the program, leveraging the verifier to conduct the containment checks. Therefore, state embedding fundamentally differs from assertion generation, in terms of both intent and methodology.

Kernel Fuzzing. Fuzz testing [21, 30, 38] is an effective approach and has been applied in kernel scenarios. For instance, Syzkaller [47] is capable of testing the eBPF subsystem by invoking the `bpf()` system call with random arguments. It has been integrated into upstream testing and has uncovered many memory errors [46]. Similarly, another work by iovisor [25] employs libfuzzer [36] to generate random byte sequences for testing the verifier. BVF [42] captures memory errors in eBPF programs with sanitation to indirectly detect correctness bugs. Nevertheless, existing fuzzers [33] highly rely on sanitizers to capture bugs [39, 40], and they experience difficulties in the verifier's logic bug detection. Therefore, our approach complements the existing fuzzing work and state embedding can be utilized for direct logic bug detection.

6 Conclusion

In this paper, we have introduced state embedding, a novel and effective technique for logic bug detection in the eBPF verifier. Our approach systematically transforms the program to embed concrete states. The state-embedded program can subsequently be used to test the verifier by validating whether or not concrete states are contained in the approximation of the verifier. By applying state embedding in testing the eBPF verifier, our prototype SEV has successfully uncovered 15 logic bugs—many are critical, and two are exploitable for local privilege escalation—demonstrating the effectiveness of our approach.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd, Ken Birman, for their valuable feedback on the earlier versions of this paper. Furthermore, we are grateful for the comments and suggestions from our colleagues at ETH Zurich.

References

[1] CVE-2021-3490. https://nvd.nist.gov/vuln/detail/CVE-2021-3490.

[2] CVE-2021-4159. https://nvd.nist.gov/vuln/detail/CVE-2021-4159.

[3] CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[4] Divergence between Specification and Implementation in Agni. https://github.com/bpfverif/agni/issues/12. Accessed: Nov 20, 2023.

[5] eBPF Build Config. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests/bpf/config?id=0dd3ee311255.

[6] eBPF Instruction Set. https://www.kernel.org/doc/html/next/bpf/instruction-set.html.

[7] eBPF Selftests. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/tree/tools/testing/selftests/bpf.

[8] eBPF Verifier. https://docs.kernel.org/bpf/verifier.html.

[9] False Negatives and Incomplete Specifications in Agni. https://github.com/bpfverif/agni/issues/15. Accessed: Nov 20, 2023.

[10] Incapable of Generating Formulas for Various Kernel Versions in Agni. https://github.com/bpfverif/agni/issues/10. Accessed: Nov 20, 2023.

[11] PREVAIL Issues. https://github.com/vbpf/ebpf-verifier/issues. Accessed: Nov 20, 2023.

[12] State pruning in the eBPF verifier. https://docs.kernel.org/bpf/verifier.html#pruning.

[13] Unprivileged bpf(). https://lwn.net/Articles/660331/.

[14] Verification Not Reach Completion after Prolonged Time in Agni. https://github.com/bpfverif/agni/issues/13. Accessed: Nov 20, 2023.

[15] Andrea Arcangeli. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[16] Sanjit Bhat and Hovav Shacham. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf, 2022.

[17] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of ASE '18*, page 768–778, 2018.

[18] Paul Chaignon. Cyclomatic Complexity of the eBPF Verifier. https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html, 2023.

[19] Jonathan Corbet. CAP_PERFMON. https://lwn.net/Articles/812719, 2020.

[20] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL*, page 238–252, 1977.

[21] Google developers. OSS-fuzz: Continuous Fuzzing of Open Source Software. https://github.com/google/oss-fuzz, 2017. Accessed: Nov 20, 2023.

[22] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of PLDI*, page 1069–1084, 2019.

[23] Tejun Heo. sched: Implement BPF Extensible Scheduler Class. https://lwn.net/Articles/916290/.

[24] William E. Howden. Theoretical and Empirical Studies of Program Testing. In *Proceedings of ICSE*, page 305–311, 1978.

[25] iovisor. bpf-fuzzer: Fuzzing Framework Based on libfuzzer and Clang Sanitizer. https://github.com/iovisor/bpf-fuzzer.

[26] Juan José López Jaimez and Meador Inge. Buzzer. https://github.com/google/buzzer.

[27] Timotej Kapus and Cristian Cadar. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of ASE*, page 590–600, 2017.

[28] Jim Keniston. Linux Kprobe. https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[29] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of ISSTA 2019*, page 239–250, 2019.

[30] lcamtuf. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2013.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of PLDI*, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of USENIX'93*, page 2, 1993.

[33] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the Security of Linux EBPF Subsystem. In *Proceedings of APSys*, page 87–92, 2023.

[34] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *Proceedings of OSDI*, 2020.

[35] Jan Onderka and Stefan Ratschan. Fast Three-Valued Abstract Bit-Vector Arithmetic. page 242–262, 2022.

[36] LLVM Project. libFuzzer: a Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[37] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux eBPF. https://ebpf.io.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*, pages 167–182, August 2017.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of USENIX ATC*, page 28, 2012.

[40] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, page 62–71, 2009.

[41] Alexei Starovoitov. CAP_BPF. https://lwn.net/Articles/820560, 2020.

[42] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of EuroSys*, page 689–703, 2024.

[43] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.*, 53(1), feb 2020.

[44] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *Proceedings of CGO*, page 254–265. IEEE Press, 2022.

[45] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In Constantin Enea and Akash Lal, editors, *CAV*, pages 226–251, 2023.

[46] Dmitry Vyukov and Andrey Konovalov. Syzbot Dashboard. https://syzkaller.appspot.com/upstream, 2015.

[47] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an Unsupervised Coverage-guided Kernel Fuzzer. https://github.com/google/syzkaller, 2015.

[48] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of ICSE*, page 1398–1409, New York, NY, USA, 2020.

[49] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective Dynamic Detection of Alias Analysis Errors. In *Proceedings of ESEC/FSE*, page 279–289, 2013.

[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of PLDI*, page 283–294, 2011.

[51] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of ICSE*, page 163–174, New York, NY, USA, 2022.

[52] Shung-Hsi Yu. Model Checking (a very small part) of BPF Verifer. https://gist.github.com/shunghsiyu/a63e08e6231553d1abdece4aef29f70e. Accessed: Nov 20, 2023.

[53] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and Understanding Bugs in Software Model Checkers. In *Proceedings of ESEC/FSE*, page 763–773, 2019.

[54] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel Storage Functions with eBPF. In *OSDI*, pages 375–393, July 2022.

Using Dynamically Layered Definite Releases for Verifying the RefFS File System

Mo Zou[1,2], Dong Du[1,2], Mingkai Dong[1,2], Haibo Chen[1,2,3]
[1]*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*
[2]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
[3]*Huawei Technologies Co. Ltd*

Abstract

RefFS is the first concurrent file system that guarantees both liveness and safety, backed by a machine-checkable proof. Unlike earlier concurrent file systems, RefFS provably avoids termination bugs such as livelocks and deadlocks, through the *dynamically layered definite releases* specification. This specification enables handling of general *blocking scenarios* (including ad-hoc synchronization), facilitates *modular* reasoning for *nested blocking*, and eliminates the possibility of circular blocking.

The methodology underlying the aforementioned specification is integrated into a framework called MoLi (Modular Liveness Verification). This framework helps developers verify concurrent file systems. We further validate the correctness of the locking scheme for the Linux Virtual File System (VFS). Remarkably, even without conducting code proofs, we uncovered a critical flaw in a recent version of the locking scheme, which may lead to deadlocks of the entire OS (confirmed by Linux maintainers). RefFS achieves better overall performance than AtomFS, a state-of-the-art, verified concurrent file system without the liveness guarantee.

1 Introduction

This paper presents RefFS, a concurrent file system with a mechanized proof of both safety and liveness properties. Liveness means that each operation of RefFS provably terminates under the assumption of fair scheduling. The proof rules out a wide range of bugs that occur in concurrent file systems [67], such as deadlocks, livelocks, and infinite loops.

Proving the absence of termination bugs is important, because they are too subtle to be correctly handled by developers. For instance [5], a task might not deadlock with another task via a direct ABBA[1] pattern, but instead through a complex circular dependency chain involving multiple tasks. A wide range of other termination bugs [54] occur, posing a threat to the software system. Once triggered, these bugs can lead to serious consequences, such as a system hang [16].

Testing and program analysis techniques (see §2 for more detail) have been used to detect (a subclass of) termination bugs. Although effective in practice, they cannot cover all possible cases. Formal verification is a promising approach. Researchers have made tremendous progress in concurrent file system verification [20, 101] and liveness verification [43, 56, 74]. Yet, modular liveness verification (focusing on one operation or one line of code at a time) of concurrent file systems remains an open problem.

In principle, proving liveness requires a well-foundedness argument [2, 60], i.e., within a finite number of steps, some progress event happens. For a sequential program, a well-founded metric such as the remaining steps of the program measures its progress [46, 66]. With each step, the metric must decrease, but not infinitely. Hence, the program provably terminates after running out the metric.

Unfortunately, proving liveness for a concurrent file system still faces the following challenges. First, the approach should support general *blocking scenarios*. A thread that is blocked in a busy waiting loop (e.g., to acquire a lock) cannot achieve progress by its own steps, but relies on the steps of *other* threads (e.g., the thread owning the lock). We aim for general busy waiting loops. This should be distinguished from work [12, 48] that supports only lock primitives, ignoring ad-hoc synchronization, which is common in file systems (see §2).

Furthermore, it is important to support modular reasoning, even though *nested blocking* causes progress dependencies between threads. Consider the following case. An `unlink` operation owns `parent` and requests `child`. Thread t_1 that tries to acquire `parent` is blocked by `unlink`, which itself is blocked by another thread t_2 that owns `child`. A metric for t_1's progress towards acquiring `parent` would include not only `unlink`'s steps to release `parent`, but also t_2's steps to release `child`. The latter contributes to t_1's progress *indirectly*. Explicitly considering such indirect steps hampers modularity. This issue becomes more pronounced with more threads chained by nested blocking.

Last but not least, the approach should prevent *circular nested blocking*. While an intuitive approach might specify a static order for nested blocking, the complexity of file systems introduces *diverse* and *dynamic* blocking order. On the one hand, file systems exhibit diverse nested blocking scenarios, with different blocking orders and concurrently executed by multiple threads. On the other hand, these nested blocking scenarios exhibit dynamic order, i.e., the exact rank of some specific blocking event cannot be known statically. For instance, the parent-child nested blocking of `unlink` refers to a dynamic set of inode pairs as a file system evolves. Two inodes with parent-child relation may swap their positions, ex-

[1] One acquires locks in the order of AB while another in the order of BA.

hibiting opposite orders in `unlink`. In `rename`, the blocking order of old and new parents is also unknown in advance. To avoid circular dependency, formally capturing the blocking order is essential but extremely challenging.

To meet such challenges, this paper makes the following contributions:

- A methodology for proving liveness, based on an acyclic waits-for graph. A vertex refers to an action waited by a blocked thread. The blocking thread must definitely fulfill the action. A directed edge represents a waits-for dependency between actions. To avoid circular dependencies, the waits-for graph must be acyclic.

- The dynamically layered definite releases (*DLDR*) specification, which supports modular liveness reasoning about concurrent file systems, with the following key ideas. (1) *Definite release* specifies that an acquired lock will always be released. (2) It proves termination of ad-hoc busy waiting lock loop based on a metric-decreasing idea. (3) We assign each definite release a layer, and allow a definite release to wait for only a higher-numbered one. Acquiring a lock waits only for the definite release of the lock (direct steps); layers decouple dependencies between definite releases (indirect steps), achieving modular reasoning. (4) The layering has two components: one follows the file system hierarchy (a parent may wait for a child), which is acyclic by definition; a *temporary dependency* models the non-deterministic blocking order in rename, and ensures acyclicity by construction.

- A protocol-level proof of Linux VFS's locking scheme based on an extension of the *DLDR* specification. The proof follows the Linux directory locking documentation [29]. We found a serious deadlock flaw; it was confirmed and fixed (we prove the fix correct).

- The MoLi framework for verifying termination and functional correctness of concurrent file systems. MoLi supports (1) definite releases with dynamic layers to achieve modular termination reasoning, and (2) non-atomic abstract operations to model non-atomic implementations. The framework is mechanized in Coq to ensure the reliability of the verification. Currently, MoLi does not support crash safety. MoLi's soundness has been formally proved on paper (a mechanized Coq proof is left as future work).

- The RefFS file system, the first modularly-verified concurrent file system with termination guarantees. RefFS supports highly concurrent path traversal using reference counting (refcount) [30]. Users are not bothered by the more fine-grained behaviors because the abstraction of RefFS hides refcounts, locks, and internal data structures, and exhibits atomic directory lookups.

The rest of this paper is organized as follows. §2 motivates this work with a study of termination bugs. §3 explains the MoLi methodology. §4 introduces the *DLDR* specification and its extension to directory locking in Linux. §5 describes the MoLi framework. §6 presents RefFS's design and verification. §7 evaluates RefFS and MoLi. §8 relates MoLi to previous work. §9 concludes.

2 Motivation

2.1 Study of Termination Bugs

Termination bugs cover a wide range, from non-concurrent ones (such as infinite loops) to concurrent ones (such as deadlocks and livelocks). A recent survey [16] on security vulnerabilities in file systems shows that about 7% of CVEs are related to non-termination. A prior study [67] on file system patches reveals that up to 40% of concurrency bugs are due to deadlocks. Deadlock-related semantic bugs are hard to diagnose, e.g., misuse of the `GFP_KERNEL` flag [70], and may hurt the system for years.

From a verification perspective, this raises several important questions: (1) How can one classify these bugs based on the challenges they pose for verification? (2) What are the primary classes of termination bugs? (3) What makes these bugs dominant and challenging to avoid? Answering these questions can help focus our verification efforts. Therefore, we performed a comprehensive study of termination bugs in Linux file systems (from 2020 to 2023). We collected 213 bugs in total by reading commit messages of patches [54]. We make the following observations.

Bug classification. Termination bugs can be classified based on whether they are concurrency bugs or not. 18% of termination bugs are non-concurrent. They include infinite loops or recursion, due mainly to logic mistakes (e.g., missing checks [82]) or generic errors (e.g., inappropriate truncation [42] or overflow [94]). 82% of termination bugs are concurrent. Within concurrent bugs, 95% of them are deadlocks and 5% are livelocks. Deadlock occurs when a thread becomes blocked, waiting for a specific action that never happens, and none of the involved threads can make progress. In livelock, a thread is constantly delayed, resulting in an inability to make progress.

Let us now look into deadlocks to understand the underlying factors contributing to their prevalence.

Ad-hoc synchronization. 46% of deadlocks (all percentages hereafter are relative to deadlocks) involve ad-hoc synchronization, where a thread is waiting for a specific event, such as transaction completion [11], flushing of dirty inodes [9], or other custom synchronization points [76, 90]. Deadlock analysis tools commonly focus on well-known and structured synchronization patterns, e.g., lock acquisition and release, but ad-hoc synchronization often lacks such patterns, which makes it challenging to analyze and detect.

Nested blocking. 21% of deadlocks are of type AA, where a thread is blocked by itself [25, 72]. The remaining 79% involve nested blocking. Nested blocking occurs when a task that is blocking another task, gets blocked itself. For instance, nested acquisition of locks may cause nested blocking. 42% of deadlocks involve *both* nested blocking and ad-hoc synchronization [1, 24]. 23% of deadlocks involve at least three concurrent tasks [4, 7]. To prevent such bugs, it is necessary to examine, not only the local blocking order, but also the absence of global, circular dependencies within the system. However, existing approaches often fail to present a global order of dependencies, hampering the ability to detect such deadlocks effectively.

Dynamic order. 8% of deadlocks exhibit a dynamic blocking order, where the exact order between two blocking events cannot be predetermined. For instance, object removal (`unlink`/`rmdir`) acquires inode locks in a parent-child order, with the definitions of "parent" and "child" based on the current state of the file tree. As the file tree evolves, the set of inode pairs that satisfy this parent-child relationship dynamically changes. Similarly, many other data structures, such as the forest structure in BTRFS and various list implementations, also exhibit dynamic blocking order. These scenarios further contribute to the complexity of the issue. Even for experts in the domain, mistakes still occur [10, 86, 87], highlighting the difficulty of effectively managing these complexities.

To summarize, this paper focuses primarily on addressing the deadlock-related challenges in file systems, i.e., ad-hoc synchronization (§4.1), nested blocking (§4.2), and dynamic order (§4.3). We briefly discuss the support for livelocks (§4.5) and non-concurrent bugs (§5.3).

2.2 Limitations of Previous Work

There are a number of program-analysis-based techniques that aim at detecting deadlock [51, 91], livelock [13] or infinite loop [15, 17] respectively. Although effective in practice, their common problem is false positives. Programmers still have to manually confirm or reproduce the bug.

Various fuzzing-based testing tools [38, 50] can also reveal termination bugs. However, they and previous program-analysis tools cannot avoid false negatives.

For instance, the Linux kernel has a runtime validator to check locking correctness [31]. Users inform the validator of the hierarchy (a fixed order) between lock objects. This has the following drawbacks. First, the validator does not recognize ad-hoc synchronization. Second, it does not provide a general principle on how to handle dynamic locking order, whose hierarchy cannot be predetermined. Third, the annotations by developers may be wrong or insufficient, giving rise to both false positives [95] and false negatives.

Some efforts support deadlock-freedom (DF) verification [12, 48, 62, 93]. They track dependencies between blocking primitives to prevent circular blocking. This approach treats deadlock-freedom as a safety property, and does not prevent livelocks or non-termination of critical sections. They often assume known lock primitives, and do not apply to ad-hoc synchronization.

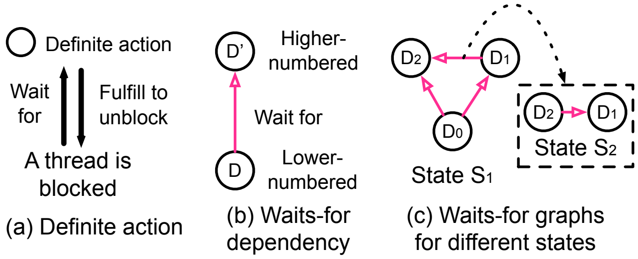


Figure 1: **The MoLi methodology for proving termination.** Waits-for graph consists of definite actions (vertices) and waits-for dependencies (edges). In (c), the dependencies between $\mathcal{D}_1$ and $\mathcal{D}_2$ are different at different states.

Some of these DF efforts [14, 61] also have limited support for dynamic lock orders. They consider only situations where a lock order change has *local* effects, i.e., it suffices to locally check some ordering constraints of the current operation to ensure acyclicity, without considering concurrent operations. For instance, for a rotation of a balanced tree, it suffices to check only the relations between the moved nodes, to ensure a global tree-based partial order. However, in a file system, it is necessary to check the absence of circular dependencies globally after the order change, which requires nontrivial concurrency reasoning (see §4.3 for more detail).

There is theoretical work [32, 64] to help reason about general blocking scenarios. LiLi [64] proposes a program logic to support the verification of starvation freedom and deadlock freedom. But LiLi does not support layered reasoning (see §4.2), and thus does not allow modular reasoning over nested blocking (i.e., verifying each line of code independently). TaDA Live [32] introduces layers to capture blocking orders, but it does not support layer changes caused by concurrent operations. Neither TaDA Live nor LiLi has a mechanical framework or an executable implementation.

3 The MoLi Methodology

Our approach to proving liveness is to exhibit an acyclic waits-for graph. A vertex is an action that a blocked thread is waiting for. A directed edge represents a waits-for dependency between actions. The waits-for graph shrinks as follows (we will discuss its growth in §4.5). Vertices with out-degree zero (not waiting for anything) must be unblocked to be removed from the graph. New vertices become leaves and are unblocked. Consequently, all waited actions eventually happen, creating progress for the blocked threads. Below, we discuss how this approach handles the challenges of file systems.

To support general blocking scenarios, MoLi borrows ideas from previous work [64]: a thread t is blocked when another thread does not fulfill the unblocking action that t is waiting for (e.g., releasing the lock or finishing the transaction); thus, the specification should describe actions that one thread will definitely fulfill (Fig. 1a). All such definite actions should be specified by proof authors, and constitute the vertices of the waits-for graph (Fig. 1c).

In nested blocking, a definite action, e.g., releasing lock $\texttt{L}_1$ by thread t, gets blocked and waits for another definite action, e.g., releasing lock $\texttt{L}_2$ by another thread. We represent this waits-for dependency as a directed edge. To capture acyclicity, MoLi requires proof authors to layer vertices so that a vertex only waits for a *higher-numbered* vertex (Fig. 1b). For instance, proof authors can define the layers as the reverse topological order of the waits-for graph.

To capture the dynamic order of nested blocking, MoLi allows to define layers *dynamically* according to the system state. For instance (Fig. 1c), definite action $\mathcal{D}_1$ waits for $\mathcal{D}_2$ in one state, while this dependency may be the opposite in a different state. The layers reflect the waits-for dependencies of the current state.

To ensure acyclicity despite layer changes, MoLi enforces a *dependency condition* on layers: for any ongoing dependency between definite actions, their layer relation represents this dependency and stays unchanged despite state changes. For instance, as long as definite action $\mathcal{D}_1$ waits for $\mathcal{D}_2$, the layer of $\mathcal{D}_1$ must remain less than $\mathcal{D}_2$ despite state changes. Because all ongoing dependencies are immune to state changes, the system is never in danger of circular dependency.

We now explain in more detail how to apply this methodology to a concurrent file system.

4 Dynamically Layered Definite Releases

Directory locking refers to the locking scheme used for directory operations. Ensuring its correctness is important yet challenging. We tackle the challenges by introducing the dynamically layered definite releases specification (§4.1-4.3). Then, we apply the specification to the Linux Virtual File System (VFS) (§4.4) and discuss how to support delay (§4.5).

4.1 Definite Release

The rule for concurrent access in a Linux FS is to protect each inode with its own associated, fine-grained lock. A thread t acquiring a lock may be blocked by another thread holding the lock. To prove t's termination, we need to show that t will eventually become unblocked. Consider the code snippet in Fig. 2a[2]. The code traverses from the cur directory and looks up the name path[i] to find the next inode. The lookup is protected by holding the lock on cur, and the lock is released afterward. Assume all threads only execute this piece of code.

```
// Pre: no lock owned
while(path[i]!=NULL)
  lock(cur);
  next=lookup(cur,path[i]);
  if (next==NULL) {
    unlock(cur); return;}
  unlock(cur);
  cur=next; i++;
}
```

(a) traversal loop.

```
def lock(cur):
  int i;
  i=getAndInc(cur.next);
  while(i!=cur.owner){}

def unlock(cur):
  cur.owner=cur.owner+1;
```

(b) ticket lock.

Figure 2: **Single locking in path traversal.** Termination of lock(cur) relies on other threads releasing the lock by increasing cur.owner.

A fair lock, such as a ticket lock (Fig. 2b), is used to ensure termination; every thread lines up for the lock by getting a ticket. A thread acquires the lock if its ticket equals owner, and releases the lock by increasing the owner. The question is why the lock(cur) statement would terminate.

Blocking is caused by the absence of environmental behavior, e.g., not releasing the lock. To prove termination, the specification should describe the certainty of some state transition. For lock-based blocking, we propose a domain-specific specification called *definite release*.

Definition 1 (Definite Release)

> **Definite release** says, for some thread t and lock, if t owns the lock, t will eventually release the lock.

Definite release is inspired by the *definite action* notion proposed by LiLi [64], which can characterize an action that will definitely happen. However, one key difference is that definite action in LiLi does not support layered reasoning, and thus cannot modularly handle nested locking (see §4.2).

Specifically, definite release ($\mathcal{D}$) specifies a state transition: "t owns the lock" ⇝ "t releases the lock", where, informally, the notation ⇝ states that the state transition *eventually* happens. Definite release supports busy-waiting lock loops (not just lock primitives). For instance, definite release of a ticket lock can be formalized as $(\texttt{owner} = \text{t}.\texttt{i}) \leadsto (\texttt{owner} = \text{t}.\texttt{i} + 1)$, where t.i means the local variable i of thread t.

Rely-guarantee style reasoning. Definite release establishes a protocol that helps reason about the termination of locks as follows. First, once acquired, the release of a lock must be **guaranteed** by each thread. Second, to prove the termination of a lock, a thread may **rely** on some other thread releasing the lock. However, the above protocol does not avoid circular reasoning, which is usually unsound in proving termination. For instance, in a deadlock, each thread relies on the other to release, but this is circular and will never happen. LiLi's approach fixes this by requiring that each thread fulfills a definite release *without* waiting itself for any definite release.

Definite release achieves thread-modular verification of Fig. 2, focusing on one thread at a time, rather than explicitly considering steps of concurrent threads. **Rely**: if thread t is

[2] This simplified version is incorrect because it omits reference counting; we will fix this in §6

```
// Pre: no lock owned
lock(parent);
child=lookup(parent,name);
lock(child);
// Perform checks and
// do unlink/rmdir
...
unlock(parent);
```

Figure 3: **Code snippet for nested locking in `unlink/rmdir`.** The release of `parent` may be blocked by the acquisition of `child`. Code for error handling omitted.

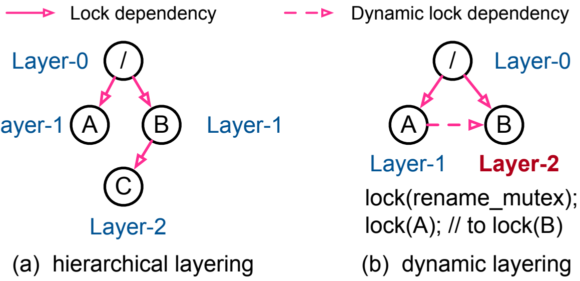


Figure 4: **Waits-for graphs and layerings.** Hierarchical layering accounts for parent-child lock dependencies. Dynamic layering captures the dynamic lock dependencies in `rename`.

blocked, spinning inside the **while** loop of `lock`, t relies on definite release, i.e., the increase of `owner` by lock holder; t can acquire the lock, because it needs to wait for a finite number of definite releases only. **Guarantee**: once t acquires the lock of `cur`, assuming that `lookup` terminates, t guarantees to fulfill the release without waiting for any definite release.

4.2 Hierarchical Layering

Some file system operations require nested locking (holding one lock and requesting another). For instance, consider the algorithm for removing an inode (in `unlink` or `rmdir`) in Fig. 3. The algorithm acquires `parent` to look up `child`. The lock on `parent` does not protect its children. To check whether `child` can be removed, we acquire `child`. When the operation finishes, it releases `parent`.

Unfortunately, in such nested locking, the release of the first lock may be blocked waiting for the release of the second lock. Hence, the guarantee of the first lock's definite release no longer holds.

If we stick to LiLi's approach in §4.1, we cannot prove the first lock by using its definite release, but have to specify more fine-grained actions that can definitely happen without waiting for any actions. For instance, acquiring `parent` may have to wait for (1) the release of `child` if the thread that owns `parent` is blocked requesting `child`, and (2) the release of `parent` if the thread that owns `parent` is not blocked. As a result, the proof for `lock(parent)` explicitly considers how `lock(child)` would be unblocked first, which is not modular. A modular approach would verify each line of code separately. The evaluation in §7.2 shows that LiLi's approach may cost 7 times the proof effort (measured in the lines of Coq) than our modular approach presented below.

We define *lock dependency* as below.

Definition 2 (Lock Dependency)

For nested locking in order of `A` and `B`, the definite release of lock `A` may depend on the definite release of lock `B`. This defines the **lock dependency** relation from `A` to `B`.

Intuition on termination. Lock dependency coincides with waits-for dependency (§3). Fig. 4a shows the possible lock dependencies in a file system as a *waits-for graph*. There is a lock dependency from a parent to its child, as shown by the directed edge. The dependencies extend transitively to the parent's descendants. For example, `root` has a lock dependency on `B`, which has a lock dependency on `C`. If the waits-for graph ever contains a cycle, the system is deadlocked.

The good news is that this waits-for graph so far follows the file system tree, so the dependency chain ends at the leaf inodes. The definite release of a leaf inode does not wait for anything. The definite releases of other inodes in the chain occur one by one in the leaf-to-root direction. Consequently, the definite release of all inodes must happen, ensuring that the parent-child nested locking terminates.

Hierarchical layering for definite releases. It is not harmful for definite releases to have dependencies *as long as* the dependencies are not circular. To formalize this intuition, we choose to assign a layer to each definite release, and allow a definite release to wait only for a *higher-numbered* one. In a file system, we use *hierarchical layering* — the layer of an inode's definite release (an inode's layer in short) is the length of the path from `root`. For instance, in Fig. 4a, `root` is assigned 0, and after each hop, the layer increases by one.

More formally, hierarchical layering can be represented as below. A *layer function* $\mathcal{HL}$ takes the inode number `inum` and file system state `FS` to return `inum`'s layer under `FS`. If `inum` is reachable from the root, distance computes the length of path (with type *Nat*) from the root to `inum` under `FS`. The layer is undefined otherwise.

$$\mathcal{HL}(\texttt{inum},\texttt{FS}) \overset{\text{def}}{=} \text{distance}(\text{root}(\texttt{FS}),\texttt{inum},\texttt{FS}) \quad \text{if defined}$$

For a well-formed `FS`, each inode is reachable from the root following a unique path, i.e., each inode has a uniquely determined layer. Hence, inodes are totally ordered, and dependencies are not circular for any well-formed `FS` (assume `FS` does not change for now). Users specify the well-formedness of `FS` as invariants (e.g., each child has only one parent), and prove that invariants always hold.

Modular reasoning with layering. Each thread should still **guarantee** to release an acquired lock. But its fulfillment can wait for a *higher-numbered* definite release. Hierarchical layering ensures the wait is not circular. A thread proves the termination of a `lock` statement independently by only **relying** on definite release of the lock.

```
def lock_rename(d1,d2):
1 if(d1==d2){
2   lock(d1);return;}
3 lock(rename_mutex);
4 if(ancestor(d2,d1)){
5   lock(d2);
6   lock(d1);
7   return;
8 }
9 lock(d1); TDep:=(d1,d2)
10 lock(d2); TDep:=None
11 return;
```

Figure 5: **Nested locking in `lock_rename`.** The locking order is dynamic, e.g., `lock_rename(d1,d2)` and `lock_rename(d2,d1)` may acquire `d1` and `d2` in an opposite order. Code in gray boxes captures the dynamic order using ghost state. These locks are released when `rename` exits.

Layering decouples the dependency between definite releases and enables modular reasoning for Fig. 3. When t is blocked inside `lock(parent)`, t relies on definite release of `parent`, and proves termination by considering only `parent`'s internal waiting queue, without looking into other code. t proves `lock(child)` similarly. After t acquires `parent`, its definite release requires to prove `parent`'s layer is less than `child`, which is true by definition of layers.

4.3 Dynamic Layering

The `rename` operation moves an inode (more precisely, the subtree with the inode as root) from its old parent to a new parent. It needs to acquire the locks of both directories to ensure atomicity. The function `lock_rename` in Fig. 5 is a simplified version of the implementation of Linux VFS (ignore the code in gray boxes for now). If the two directories are the same, it only needs to acquire one lock. To avoid concurrent issues, VFS requires a cross-directory `rename` to acquire the global per-filesystem lock `rename_mutex`. Holding `rename_mutex` prevents another cross-directory `rename` from changing the ancestry relationship. Furthermore, to not conflict with the parent-child order, if one directory is the ancestor to another, it acquires the ancestor first. If not, the locking order does not matter, so the code chooses a default order, i.e., old parent first (`d1` before `d2`).

Dynamic lock dependency. In contrast to the parent-child lock dependency, the lock dependency in `lock_rename` cannot be predetermined. Specifically, prior to acquiring `rename_mutex`, the lock dependency between `d1` and `d2` is *not stable* since other threads might dynamically alter it. Once `rename_mutex` is acquired, the lock dependency becomes fixed. If `d2` is an ancestor to `d1`, the lock dependency is from `d2` to `d1` as shown in lines 5-6. Otherwise, it is from `d1` to `d2` as shown in lines 9-10. Furthermore, after lines 6 and 10 where the code has acquired the respective locks for `d1` and `d2`, we can remove the dependency between them, because one of them no longer waits for the other.

Intuition on termination. The waits-for graph remains acyclic during `lock_rename`, which ensures termination. Before `lock_rename` executes, the graph is tree-shaped. Then there is a dynamic lock dependency between `d1` and `d2`. According to the locking rules in `lock_rename`, this dynamic lock dependency is not from a child to its ancestor, and thus does not form a cycle with existing parent-child dependencies. Finally, removing the dynamic lock dependency will not introduce a cycle. The acyclicity of the waits-for graph ensures that definite releases happen in order, despite dynamically-changing lock dependencies. The code terminates, because all definite releases are guaranteed to be satisfied.

Dynamic layering. We propose *dynamic layering* to capture the lock dependency in `lock_rename`. The layer for each inode is defined as the length of the *longest* path from the root in the waits-for graph. For instance, after the introduced dynamic lock dependency (drawn as a dashed arrow) in Fig. 4b, the longest path from `root` to `B` is `root`-`A`-`B`, so the layer for `B` is now 2.

To encode dynamic layers, we leverage *ghost state* [75], a commonly used verification technique. Ghost state is added by users in the abstract model to assist the proof, which does not influence (or exist in) the concrete program. We introduce a *temporary dependency* `TDep`, a globally unique value that tracks the lock dependency introduced by `lock_rename`. `TDep` is an option type of a pair of inums, i.e., either `None` or $(\texttt{inum}_1, \texttt{inum}_2)$ representing there is a lock dependency from $\texttt{inum}_1$ to $\texttt{inum}_2$.

Dynamic layering $\mathcal{DL}$ is defined below. $\mathcal{DL}$ takes `inum` to return its layer under *full state* `S`, where `S` includes file system state `FS` and ghost state `TDep`. The root inode has layer 0. If there is no dynamic dependency to `inum`, i.e., `inum` does not equal the second item in `TDep` (written $\texttt{TDep.inum}_2$ for simplicity), `inum`'s layer is one plus its parent's layer. If `inum` equals $\texttt{TDep.inum}_2$, its layer is one plus the larger layer between the hierarchical layers (as defined in §4.2) of its parent and $\texttt{TDep.inum}_1$. Otherwise, the layer is undefined.

$$
\begin{aligned}
&\texttt{S} = (\texttt{FS}, \texttt{TDep})\\
&\mathcal{DL}(\texttt{inum}, \texttt{S}) \overset{\text{def}}{=}\\
&\begin{cases}
0 & \text{if } \texttt{inum} = \text{root}(\texttt{FS})\\
\mathcal{DL}(\texttt{par}, \texttt{S}) + 1 & \text{if } \exists \texttt{par}, \text{parent}(\texttt{par}, \texttt{inum}, \texttt{FS})\\
 & \quad \wedge \texttt{inum} \neq \texttt{TDep.inum}_2\\
\max\{\mathcal{HL}(\texttt{TDep.inum}_1, \texttt{FS}), & \text{if } \exists \texttt{par}, \text{parent}(\texttt{par}, \texttt{inum}, \texttt{FS})\\
\quad \mathcal{HL}(\texttt{par}, \texttt{FS})\} + 1 & \quad \wedge \texttt{inum} = \texttt{TDep.inum}_2\\
\text{undefined} & \text{otherwise}
\end{cases}
\end{aligned}
$$

This definition focuses on inode locks. For the per-filesystem lock `rename_mutex`, we also specify a definite release, whose layer is a special minimum value so it can depend on all other definite releases.

The code in gray boxes (Fig. 5) presents feasible updates to `TDep`. Only the thread that holds `rename_mutex` can set `TDep`, and `TDep` is `None` when no thread holds `rename_mutex`. `TDep` is set to `(d1,d2)` after `lock(d1)` in line 9 to represent the upcoming lock dependency from `d1` to `d2`. Note that we do not need to set `TDep` after `lock(d2)` in line 5 because the lock dependency from `d2` to `d1` is already present in the waits-for graph due to transitive parent-child dependencies. We set `TDep` to `None` after line 10 to remove the lock dependency.

Reasoning with dynamic layering. Each `lock` statement still enjoys a modular proof by relying on its definite release. The guarantee of definite release can wait for a higher-numbered definite release. Now, taking state changes into consideration, we must further prove that the dependency relation (i.e., layer relation) stays unchanged during the wait. We call this the *dependency condition*.

The dependency condition is vital to acyclicity. Note that a circular dependency can happen only if a state change introduces a direct or indirect dependency opposite to an existing dependency. However, the dependency condition forbids this, by requiring the dependency relation to be stable.

Let's prove the dependency condition for Fig. 3. When a thread has acquired `parent`, and gets blocked by `lock(child)` at line 4, `parent`'s definite release waits for that of `child`. The dependency condition requires to show that `parent` is lower-numbered than `child`, even in the presence of concurrent operations modifying file-system state. This is the case because (1) the current thread holding `parent` prevents a concurrent `unlink`/`rmdir`/`rename` from removing `child` and (2) according to the definition of $\mathcal{DL}$, `child`'s layer is greater than or equals `parent`'s layer $+1$.

4.4 Directory Locking in Linux VFS

To understand whether dynamic layering scales to more directory locking orders in VFS, we extend dynamic layering to cover all the nested locking scenarios mentioned in the Linux documentation [29]. The extra lock dependencies that have not been discussed yet are the following: (1) the *dir-to-non-dir* dependency, from a directory to a non-directory, and (2) the *inode-pointer* (i.e., inode address) dependency, from a non-directory to a non-directory with larger inode pointer. For instance, (1) link creation locks the parent and then the non-directory source, or (2) rename locks the source and the target when they are non-directories.

To capture these dependencies, a layer could either be (`Dir`, *nat*) for a directory with its dynamic layer ($\mathcal{DL}$ defined in §4.3), or (`NonDir`, *addr*) for a non-directory with its inode address. Comparison rules are:

- (`Dir`, *nat*) $<$ (`NonDir`, *addr*) for the *dir-to-non-dir* dependency;
- (`NonDir`, $addr_1$) $<$ (`NonDir`, $addr_2$) iff $addr_1 < addr_2$ for the *inode-pointer* dependency;
- (`Dir`, n_1) $<$ (`Dir`, n_2) iff $n_1 < n_2$.

These rules give a total order of layers, because a directory is always acquired before a non-directory and the two groups are ordered by dynamic layers and inode pointers, respectively.

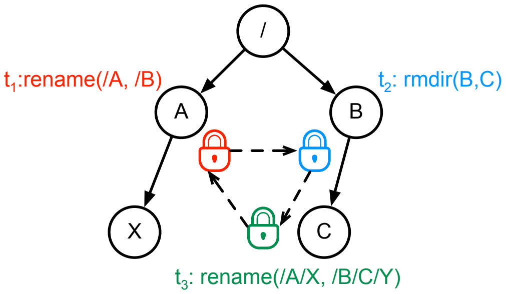


Figure 6: **A deadlock bug in Linux VFS.** Locking order according to increasing inode pointer order is (`C`, `A`, `B`), which conflicts with parent-child order (`B`, `C`).

When layers do not change, deadlocks will not happen if the code acquires locks in this total order (this can be easily verified). When considering layer changes, one proves the dependency condition to prevent circular dependency. We have shown the reasoning for the most challenging case, i.e., `rename` in §4.3. Proof of all other cases is provided in [99].

Doing the proofs helped us uncover a flaw in a recent version of the locking scheme (commit 28ecee [53]). Apart from locking the two parents, `rename` may also need to lock the source inode (under the old parent) to be renamed, and the target inode (under the new parent) if it already exists. The locking rules before commit 28ecee used to be:

- locking the source if it is a non-directory;
- locking the target if it exists;
- (if one need to lock both) locking them following the order mentioned above, i.e., directory before non-directory, and non-directories in inode pointer order.

However, commit 28ecee additionally locks the source even when it is a directory, for the purpose of updating its pointer to the new parent. For a non-cross-directory rename, this introduces a new, dynamic order between source and target subdirectories that is not protected by `rename_mutex`. Commit 28ecee takes it for granted that locking source and target subdirectories (and also two parents) in inode pointer order would be enough to establish a total order between directories.

However, the problem is that inode pointer order is not transitive with parent-child order, as shown in Fig. 6. Specifically, assume the inode pointer order is `C` $<$ `A` $<$ `B` (all are directories) and assume three operations have finished path-name lookups. t_3 owns `rename_mutex` and `C`, and requests `A`. t_1 owns `root` and `A`, and requests `B`. t_1 owns `B`, and requests `C`. Now, there is a deadlock. The maintainer confirmed this [98].

The fix [87] is to acquire the source subdirectory only in the cross-directory rename case, because a non-cross-directory rename does not change the parent of the source. The locking of source and target subdirectories is now protected under `rename_mutex`, and we can capture this order by constructing a dynamic layering (see [99] for details).

We found this flaw when we failed to define the dynamic layering specification for the scheme. Indeed, having a formal specification that effectively captures lock dependencies is crucial. Such a specification not only aids in conducting formal proofs, but also enhances our fundamental understanding of the system. Interestingly, even without engaging in code proofs, the specification itself can help uncover practical bugs and vulnerabilities.

4.5 Discussion about Support for Delay

A thread will not terminate if it is infinitely delayed in an infinite loop. For example, when a thread requests an unfair lock, e.g., test-and-set lock, other threads repeatedly preempt the lock first. Delays are benign as long as there is whole-system progress, e.g., the thread that preempts the lock can make progress. To allow benign delays, while preventing infinite delay without whole-system progress, previous work [64] proposed the concept of token transfer. The basic premise is that each thread is assigned a finite number of tokens. When a thread causes delays for other threads, it should either show progress itself or consume its tokens.

Let us go back to the discussion in §3. The waits-for graph grows due to normal execution or delay; either case has a decreasing metric that shows progress. When a thread normally executes, although it may get blocked by a **while** loop, its code size decreases. We assume a fixed but arbitrary number of threads to ensure the waits-for graph does not grow infinitely. In the delay case, a thread is made to execute more steps and may get blocked, but the delaying thread shows progress or consumes its tokens.

The mechanism for delay is necessary for proving the absence of livelock, where a thread is constantly delayed in infinite loops. This pattern does not appear in RefFS. For the sake of simplicity in presentation, we will omit these details.

5 The MoLi Framework

5.1 Overview

The MoLi framework verifies the functional correctness and termination of file systems based upon the following ideas.

- MoLi expresses correctness with *termination-preserving refinement* [66], which means that all observable events (e.g., output and termination events) from the implementation (i.e., concrete data structures and operations) must also be produced from the abstraction (i.e., logical layouts and abstract operations over them). Hence, functional correctness and termination of implementation is ensured, provided the abstraction is correct in these aspects. Termination-preserving refinement also helps build proofs in a layered way, i.e., low-level code can be replaced by its abstraction in a higher-layer proof. For instance, the proof of applications can use the abstraction of file systems.
- MoLi supports *compositional* concurrency reasoning with *rely and guarantee conditions* [35, 49]. A rely condition specifies the interference that a thread will accept from the other threads. A guarantee condition specifies the transitions that a thread will make. If the rely condition of a thread is implied by the guarantee of all other threads, and each thread is individually correct, then the concurrent system is correct.

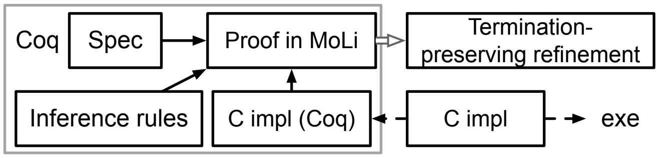


Figure 7: **The MoLi workflow.** Users provide a specification. MoLi helps users develop code proofs with inference rules.

MoLi supports modular liveness reasoning with layered definite actions. Fig. 7 shows the workflow of MoLi. MoLi is a framework built on Coq. MoLi supports the verification of C language (the Coq model of C language follows an existing framework [101]). Users specify the specification (§5.2), and then follow the inference rules provided by MoLi to manually perform the Hoare-style verification (§5.3). The framework is sound, which ensures that the proof implies the termination-preserving refinement.

5.2 Specification in MoLi

A specification includes the abstraction, rely-guarantee conditions, invariants, definite actions, and the layer function, as defined next.

Abstraction. The abstraction includes the abstract representation of the concrete state and abstract operations (*Aop*) on it. An abstraction can hide implementation details, which is easier to check and less error-prone than the concrete implementation. MoLi provides a specification language, which allows users to write non-atomic abstract operations. The language has standard commands such as **while** and **if**, and is suitable for expressing abstract operations: (1) the language's state includes the abstract state and a local *abstract stack*, which maps variables to *abstract values*, e.g., lists; (2) the language supports user-supplied primitives that model atomic transitions of abstract state; (3) it also supports atomic block $\langle C \rangle$ where C executes atomically (see §6.1 for an example).

State. In MoLi, state includes not only the concrete state accessed by the implementation, but also several auxiliary parts, i.e., the specfication language's state, tokens, and ghost state. MoLi uses *tokens* as local state to ensure termination (as explained in §5.3). Users may also introduce *ghost state*, which exists only in the abstract model, to assist verification.

Rely/guarantee conditions (R/G) and invariants (I). R and G

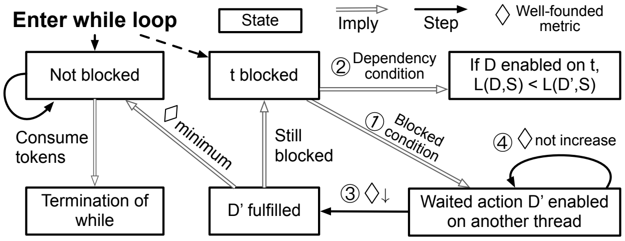


Figure 8: **Logic for termination of a while loop.** Users specify the number of *tokens* and a *well-founded metric* $\Diamond$. In the non-blocking case, tokens strictly decrease for each iteration. In the blocking case, $\Diamond$ strictly decreases whenever a waited definite action ($\mathcal{D}'$) executes. When $\Diamond$ decreases to its minimum, the thread is no longer blocked. The dependency condition enforces an enabled definite action ($\mathcal{D}$) waits only for a higher-numbered one ($\mathcal{D}'$).

define the allowed state transitions and are checked at each step: one checks that (1) the current thread's state transitions satisfy G, (2) the pre-/post-conditions of a command stay true, even when a concurrent thread changes the state, as long as this change is allowed by R, and (3) I specifies an invariant on state, which must remain true under all transitions. R/G and I define the concurrency protocol. Previous efforts [35, 101] provide more detail.

Definite action and layer function. A *definite action* describes a state transition, written $P \rightsquigarrow Q$, which means that, once assertion P is true, (1) Q will eventually hold, and (2) P is preserved by both current and environmental thread until Q holds. The second condition ensures that P may not become false until the definite action is fulfilled. A definite action is called **enabled** when P holds. A *layer function* $\mathcal{L}$ takes a definite action and a state $\mathtt{S}$ to return a layer if defined.

5.3 Verification in MoLi

The judgement $\mathcal{L},\mathcal{D},R,G,I \vdash \{P \wedge Aop\}C\{Q \wedge \mathtt{skip}\}$ means (1) starting from the precondition P, the operation C must terminate to reach the postcondition Q, and meanwhile (2) an abstract operation terminates (from *Aop* to `skip`), simulating the concrete operation C. Here, both P and Q imply the invariant I and specify the consistency relation between the concrete and abstract state. MoLi provides inference rules to help users prove the judgement holds. A top-level rule, called the OBJECT rule, proves the well-formedness of the specification and the judgement for each method of the object. The OBJECT rule establishes termination of the implementation and abstraction, and a termination-preserving refinement between them. To verify each method, users follow inference rules of C language statements to step through the program.

Most rules for C language, e.g., **sequence** and **if** rules, are standard and similar to previous work [35, 101]. We explain the WHILE rule (see Fig. 8 and the simplified rule below).

Termination of a* while *loop. The WHILE rule requires establishing the judgement of the loop body (see the first line of the rule). Assuming the loop body can terminate, we check whether the loop is blocked by other threads. Based on that, the logic uses different strategies to ensure termination, i.e., consumption of tokens in the non-blocking case and decrease of metric by executing definite actions in the blocking case.

$$\frac{\begin{array}{l}\mathcal{L},\mathcal{D},R,G,I \vdash \{P \wedge B\}C\{P\} \\ \text{if not blocked, consume tokens} \\ \text{if blocked, prove conditions ①-④} \\ \text{(see ①-④ in the text and Fig. 8)}\end{array}}{\mathcal{L},\mathcal{D},R,G,I \vdash \{P\}\textbf{while } (B)\{C\}\{P \wedge \neg B\}} \ (\text{WHILE})$$

If the **while** loop is not blocked, we must show the **while** loop can iterate for only a finite number of rounds. Following previous efforts [46, 64], we require that each iteration consumes resources called *tokens*. Users specify the number of tokens before the **while** loop. The loop terminates after exhausting its tokens. For instance, the traversal **while** loop in Fig. 2a can iterate only a finite number of rounds, bounded by the length of `path`, which specifies the number of tokens.

In contrast, if a thread t is blocked, its termination relies on definite actions of other threads. Users define a well-found *metric* (i.e., that cannot infinitely decrease) for the **while** loop. Whenever a definite action happens, the metric must decrease. When the metric decreases to its minimum, the thread is no longer blocked. Specifically, users prove the following.

① **Blocked condition**: if t is blocked, t must be waiting for some definite action $\mathcal{D}'$ (to be specified) to happen, which is enabled on another thread.

② **Dependency condition**: if t has an enabled definite action $\mathcal{D}$, its layer must stay *less* than that of $\mathcal{D}'$ until $\mathcal{D}'$ is fulfilled.

③ **Metric decrease**: whenever $\mathcal{D}'$ is fulfilled, the metric decreases.

④ **Metric non-increase**: the metric never increases.

Let us prove the termination of ticket lock in Fig. 2b under the above conditions. Suppose a thread t is blocked in the **while** loop of `lock(cur)`. A precondition is that t does not own `cur`. Then we can prove ① t waits for the definite release of `cur`, which is enabled on another thread. If t has enabled definite actions, e.g., t has acquired other locks, the layer function must have captured these lock dependencies. With the layer function, users prove ② the layers of owned locks are stably less than `cur` until t's ticket equals `cur.owner`. One can specify the well-founded metric as `t.i-cur.owner`, which measures the distance from t's ticket to the current owner. ③ The metric decreases whenever an environment thread increases `cur.owner`, and ④ never increases. When the metric decreases to zero, the loop terminates.

Lock abstraction. We can prove the termination of lock implementations with the WHILE rule. But MoLi follows recent work [65], and provides a specialized inference rule for fair locks (e.g., ticket lock) to ease the burden. The abstract state of a lock is an integer `L`, whose value is either `0` when available or `t` when owned by thread `t`.

The reasoning of a lock operation is similar to the WHILE rule, only with a different metric non-increase condition: the metric never increases under transitions where the lock keeps being unavailable. We allow the metric to increase when the lock is available, because a fair lock guarantees that, if the lock becomes available for a finite number of times, a thread will eventually become the first to the lock.

Definiteness of definite actions. The WHILE rule assumes that a definite action will be fulfilled once enabled. To meet this assumption, the OBJECT rule checks two things.

- **Well-formedness**: all steps preserve an enabled definite action of some thread except that the thread itself may fulfill the definite action.
- **Postcondition restriction**: the postcondition of an operation implies no enabled definite actions.

MoLi requires an enabled definite action to be fulfilled because the following conditions hold.

- When an operation terminates, it must be fulfilled according to the **postcondition restriction**.
- Whenever the thread is blocked in a **while** loop, the proof of **while** loop ensures the termination by only relying on higher-layer definite actions according to the **dependency condition**; intuitively, this will not introduce unsound circular dependencies, so those higher-layer definite actions can indeed be fulfilled.
- The thread is proved to terminate, and thus fulfills the definite action itself due to **well-formedness**.

Soundness. After users prove each operation by following the inference rules, the framework constructs an *overall termination metric* for each thread, and shows that the overall metric strictly decreases. The overall metric combines several metrics that users have provided in their proofs.

- The totally ordered layering of definite actions ensures that the waits-for graph shrinks until the thread's waited definite action is not blocked.
- The well-founded metric for a **while** loop measures the progress from the execution of the waited definite action, until the **while** loop is not blocked.
- The **while** loop tokens count down the iterations.

We have formally proved the following main theorem (see [100] for the full pen-and-paper proof).

```
// Error handling omitted
// Def of Inode omitted
struct inodelock{
  Inode *inode;
  int refcount;
  lock lk;
}

def getilock(ilock):
  ⟨ilock->refcount++⟩

def putilock(ilock):
  ⟨ilock->refcount--;
  if(ilock->refcount==0){
    free(ilock);
  }⟩
```

```
def traversal(cur,path):
  local i=0, ret;
  getilock(cur);
  while(path[i]){
    ret=lookup(cur,path[i]);
    cur=ret;i++;}
  return cur;

def lookup(par,name):
  local child;
  lock(par);
  child=find(par,name);
  getilock(child);
  unlock(par);
  putilock(par);
  return child;
```

Figure 9: **Reference counting and traversal in RefFS.** By holding a refcount in hand, `lookup` requests `par` without worrying `par` has been freed.

Theorem 1 (Main Theorem)

Given the implementation and abstraction, if there exist *rely*/*guarantee* conditions, an *invariant*, *definite actions* and a *layer function*, such that for each operation C of the implementation and corresponding abstract operation Aop, the judgment $(\mathcal{L},\mathcal{D},R,G,I \vdash \{P \wedge Aop\}C\{Q \wedge \texttt{skip}\})$ holds w.r.t. the pre-/post-conditions by applying inference rules, then the implementation ensures termination and is a termination-preserving refinement of the abstraction.

6 Design and Verification of RefFS

RefFS is a concurrent in-memory file system running on FUSE. We verify its interfaces that manipulate the file system structure (e.g., `mkdir`/`mknod`, `rmdir`/`unlink` and `rename`), and that perform input and output to files (e.g., `open`, `read`, `write` and `close`). This covers most common operations.

6.1 Implementation and Abstraction

RefFS reuses code from a previously verified concurrent file system, AtomFS [101], e.g., the internal functions that operate on directories and files. However, RefFS uses reference counting (refcounting) for traversal, which is more fine-grained and provides better performance than lock coupling used by AtomFS. Consequently, the rename implementation, file-descriptor-based interfaces and abstraction of RefFS are different from AtomFS, as explained below. We also prove liveness guarantee of RefFS.

Refcounting. In Fig. 9, struct `inodelock` wraps over an inode with a lock for protecting the struct, and a reference count for resource reclamation. The `refcount` field counts the references to the struct. `getilock` increases `refcount` by one. `putilock` decreases `refcount` by one and frees the struct when `refcount` becomes zero. In other words, `refcount` can prevent use-after-free as long as a thread still holds a `refcount` that other threads may not decrease (i.e., `refcount>0`). We use the atomic block notation $\langle C \rangle$ to represent that command C executes atomically. This is achieved with locks (not shown in the code).

To correctly implement refcounting, RefFS takes care of the following aspects.

① **Initialization**: when allocated, an inode's `refcount` is initialized to 1, marking that there is one reference in its parent's directory entry; this `refcount` is decreased when the inode is removed from its parent (the root's initial reference cannot be decreased).

② **Reference increase**: a thread can increase the `refcount` of an inode when it already holds its `refcount` (a thread can directly increase the `refcount` of root).

③ **Reference decrease**: a thread can decrease a `refcount` that belongs to the thread, e.g., the thread has increased the `refcount` previously.

④ **Reference counting**: each thread decreases the `refcount` that it no longer needs, and the value of `refcount` equals the number of all references combined.

③ is the key to stablizing `refcount>0` and preventing use-after-free, because if a thread has increased a `refcount`, other threads may not arbitrarily decrease it. The above refcounting protocols are formalized as rely/guarantee conditions and invariants, and proved for RefFS.

Let us see how the `traversal` function in Fig. 9 obeys and leverages the above protocols. The precondition specifies that either `cur` is `root`, or the current thread holds a reference to `cur`, so that `traversal` can increase the `refcount` of `cur` (②). The code invokes `lookup` for each item of the path. `lookup` can request `par`'s lock without worrying that `par` is freed, because the code holds `par`'s refcount (③). After having found the `child` in `par`, the thread holds a reference to `child`, due to owning the directory entry of `child` in `par`. Therefore, it can increase the `refcount` of `child` (②). It then releases `par`'s lock and refcount (③ and ④).

Refcounting provides the following performance benefits.

- During path traversal, there are intervals where an operation has searched the parent directory and is about to lookup the child directory. The operation does not need to nestedly lock parent and child to protect the child from being freed during the interval. Instead, `traversal` prevents use-after-free by holding a `refcount` of child. This creates more parallelism because concurrent operations can bypass each other during path traversal.
- Some operations use a file descriptor (FD) to directly access an inode, and leverage refcounting to prevent use-after-free. For instance, `open` increases the `refcount` of the target inode, and returns the inum as FD, thereafter, `read` and `write` operations can directly access the inode.

```
def rename(src,sn,dst,dn):
... //Traverse common
  path of src and dst
rel=pathrel(src,dst);
if(rel!=0){
  lock(rename_mutex);}
... //Traverse to get
  src and dst directories
if (rel==0){
  lock(sdir);
} else if(rel==1){
  lock(sdir);
  lock(ddir);
} else {
  lock(ddir);
  TDep:=(ddir,sdir)
  lock(sdir); }
...
//If dn exists in ddir
TDep:=(sdir,dchild)
lock(dchild);
TDep:=None
...
```

Figure 10: **Highlighting lock acquisitions of RefFS `rename`.** `pathrel(src,dst)` returns 0 if `src` equals `dst`, returns 1 if `src` is a proper prefix of `dst`, and returns -1 in other cases. Depending on the result, the code decides whether to acquire `rename_mutex` after traversing the common path of `src` and `dst`. Code in gray boxes updates ghost state.

Refcounting brings challenges for liveness reasoning because we need to consider more intricate lock dependencies of FS operations. Specifically, when using lock coupling (concurrent operations cannot bypass each other), an operation that starts the traversal first would not be blocked. This is not the case with refcounting: an operation may be blocked by another operation that bypasses it, or by a file-descriptor-based operation. Nevertheless, MoLi's modular verification approach effectively manages the complexities.

Rename implementation. In Fig. 10 (ignore the code in gray boxes for now), `src`/`dst` is the path to the old/new parent, and `sn`/`dn` is the name of source/target. `rename` first traverses the common path of `src` and `dst` (using `traversal` in Fig. 9). After getting the reference of their least common ancestor, the algorithm decides whether this is a cross-directory rename by comparing `src` and `dst`. If they are not equal (or cross-directory), the code acquires `rename_mutex`. The code then traverses the *remaining* path to get the references of the old and new parents. Holding `rename_mutex` ensures that the relative position between the two directories will not be changed by concurrent renames. Therefore, one may use the path arguments to know whether they are ancestors to each other, e.g., if `src` is a proper prefix of `dst`, this means that `sdir` is an ancestor to `ddir`. The code acquires the lock of the ancestor first. Otherwise, it will acquire them in a default order. If the target (i.e., `dchild`) already exists, the code acquires its lock.

Non-atomic abstraction. The abstraction of RefFS consists of abstract file system state and abstract operations. The abstract file system, called `AFS`, is a mapping of inode number (inum) to an *abstract inode*. An *abstract inode* is either a file (a list of bytes) or a directory (a mapping of name to inum).

For an operation that needs path traversal, the correspond-

```
def MKDIR(path,n):
1 local cur=root,tmp,i=0;
2 while(path[i]){
3  ⟨tmp=lookup(cur,path[i]);
4  if(tmp==NULL){
5    return -1;}
6  cur=tmp;i++⟩}
7 ret do_mkdir(cur,n);
```

Figure 11: **Abstract operation `MKDIR` of RefFS.** `MKDIR` consists of a series of atomic directory lookups (line 3-6) and an atomic critical section (line 7).

ing abstract operation hides as much implementation detail as possible (e.g., hiding refcount, locks and internal data structure), and guarantees atomic directory lookups and an atomic fulfillment of the operation after locating the target inodes. Fig. 11 shows the abstract operation `MKDIR`. `lookup` and `do_mkdir` primitives model atomic transitions of the abstract file system. We simplify the abstract operation by grouping the loop body (lines 3-6) into an atomic block.

6.2 Verification of RefFS

We use ghost state, the temporary dependency `TDep`, to assist the proof, as introduced in §4.3. In Fig. 10, code in gray boxes updates it. `TDep` is set to `(ddir, sdir)` before line 14 to represent the upcoming lock dependency between them. `TDep` may be updated to `(sdir,dchild)` before line 18 to establish the lock dependency from `sdir` to `dchild`. `TDep` is reset to `None` after line 18.

The termination proofs include the checks in the OBJECT rule and the termination proof of locks and while loops. Since while loops are not blocked in RefFS, their proofs are trivial.

Definiteness check. The well-formedness of definite releases holds because once a thread holds a lock, all steps keep the fact true until the thread releases the lock itself. The post-condition of each operation specifies that the thread does not own any lock, so it must be the case that definite releases are fulfilled before the thread reaches the end.

Proof for locks. When thread t is blocked in `lock(L)`, t waits for the definite release of `L` (specified as $\mathcal{D}'$). The metric is 0 when `L` is available and 1 otherwise. The following holds. (1) Blocked condition: some thread t' holds `L`, so $\mathcal{D}'$ is enabled on t'. (2) Dependency condition: the dynamic layering ensures that for any lock that t owns, its layer is stably lower than `L`. (3) Metric decrease: when `L` is released, the metric decreases. (4) Metric non-increase for the lock: when `L` is not available, the metric stays 1 and never increases.

7 Evaluation

This section empirically answers several questions:

- Can RefFS provide good performance for real-world applications, and does reference-counting perform better compared with lock-coupling (§7.1)?
- Compared with previous work, how modular are the termination proofs using MoLi (§7.2)?
- How much is the verification effort (§7.3)?
- Can MoLi help eliminate bugs in practice (§7.4)?

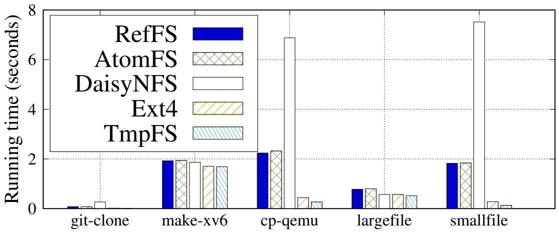


Figure 12: **Applications.** The figure shows the running time of different applications, i.e., git, make and cp. Largefile operates on a big file with 10MB. Smallfile operates on 10K files with 1KB size.

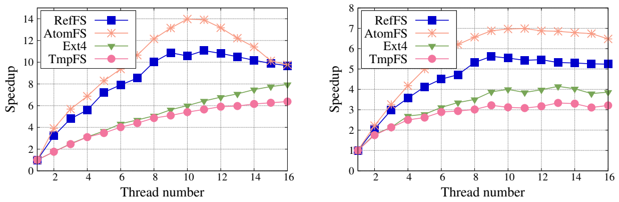
7.1 Performance Evaluation

Experimental setup. We run all of the experiments on a server machine (AWS EC2 i3.metal instance) with 72 cores (2.3GHz), 512GB DRAM, and a local 15,200GB SSD (8 disks) running Linux 5.15.8. We limit our experiments to one 36-core socket to avoid variability. We compare the performance of RefFS to the widely-used disk file system ext4 [84], the verified concurrent file system AtomFS [101], the verified concurrent NFS server DaisyNFS [20], and an in-memory file system tmpfs. All the file systems use in-memory storage.

Application performance. RefFS is complete enough to run many kinds of realistic software, including Vim [85] and GCC [37]. To evaluate application performance, we select two microbenchmarks and three application workloads: LFS microbenchmark [71, 77], cloning the git repository of xv6-public, compiling the sources of the xv6 file system with a makefile and copying the source code of qemu. These workloads are also used by previous verified file systems [22, 101]. The application workloads use only a single core.

In Fig. 12, RefFS achieves similar results to AtomFS, and better performance than DaisyNFS in most cases, due to the network I/O overhead of DaisyNFS. The worse performance of RefFS compared with tmpfs and ext4 is due mainly to the lack of fine-grained optimizations, e.g., highly optimized path traversals and optimized structures for data and metadata. Running RefFS with FUSE also introduces overhead. These issues can be overcome by more engineering.

File system scalability. We adopt two commonly used workloads in Filebench [36], Fileserver and Webproxy, to measure the scalability of RefFS. We evaluate with 16 cores and increase the number of threads used in the workloads. We do not evaluate DaisyNFS because its in-memory disk can only



(a) Scalability on Fileserver. (b) Scalability on Webproxy.

Figure 13: **Scalability of RefFS.** The overall scalability of RefFS is similar to AtomFS.

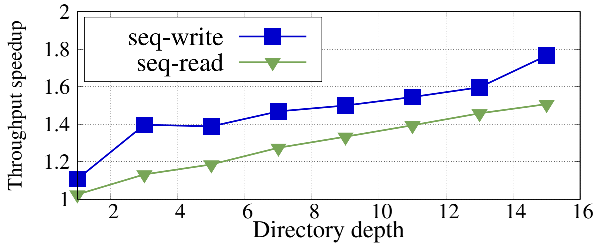


Figure 14: **Speedup of RefFS over AtomFS.** RefFS achieves higher speedup over AtomFS as the directory depth increases in LargeFile benchmarks.

support about 400MB of space, while the scalability tests consume more than 3GB in many cases.

The speedup results are in Fig. 13. RefFS can scale up to 9 cores. AtomFS shows a similar scalability, but the actual throughput (not shown in figure) of RefFS is better than AtomFS at all loads (1.08–1.43x higher in Fileserver and 1.03–1.32x higher in Webproxy). RefFS's performance is worse than ext4 and tmpfs, as expected.

Other benefits of reference counting. AtomFS uses lock coupling to traverse the path even for read and write operations. In RefFS, reference counting allows read and write to directly access the inode because `open` has increased the inode's reference. To show the benefit of this, we evaluate RefFS and AtomFS using LargeFile benchmarks under different depths of directories. In Fig. 14, with the depth increase, the speedup of RefFS over AtomFS becomes higher in both seq-write and seq-read tests in LargeFile.

Table 1: Lines of Coq proof for verifying RefFS.

Component	**LOC**	**Component**	**LOC**
Abstraction and aops	.1K	Invariant	.7K
Rely/guarantee	.4K	Code	.4K
Layered definite releases	.1K	Proof	32K
Total			**33.7K**

7.2 Modularity of Termination Proofs

Dynamically layered definite releases allow to verify each lock separately, and reuse the termination proofs for all each lock statement. To evaluate the benefits, we also use the non-layered definite actions (LiLi's approach in §4.1) to verify the termination of a lock statement in RefFS. The crucial difference is that the non-layered approach has to reason globally about the dependency chain.

Suppose there is a lock dependency chain of `root` → `B` → `C` as shown in Fig. 4a, where each thread in the chain owns a lock and requests the next lock except the last thread. Now suppose t wants to acquire `root`'s lock. To prove t's progress, LiLi's non-layered approach needs to prove how the chain gets shorter until t can acquire `root`. The following complexities exist.

In LiLi's approach, one must specify actions that can definitely happen on their own. In this case, t first waits for the lock release of `C`. Defining the action needs following the dependency chain, which requires a proof that the chain is free of cycles. Proving this fact in a general situation requires establishing global invariants, adding to the proof burden. The resulting definite action is very fine-grained and less intuitive.

Furthermore, to show the progress created by the fine-grained definite action, one should define a decreasing metric. However, the definition of the metric is unavoidably complex. Defining it as the length of the dependency chain does not work. Because after lock `C` is released, the thread that owns `B` acquires `C`, and may go on requesting other locks, thus getting involved in a even longer dependency chain. As a result, defining this metric requires considering a thread's local progress (e.g., its remaining steps), and the length of the dependency chain, with the former prioritized before the latter (we omit further detail). This poses a significant proof burden in the metric-decrease and metric-non-increase proofs.

By contrast, the specification and termination proof for a lock statement with our layered approach (see §6.2) focus only on the lock. The extra proof burden is the dependency condition, which requires to show any owned locks are lower-numbered than the lock to acquire. This proof is usually trivial with dynamic layering. Code proofs in Coq have confirmed the analysis—the non-layered approach needs 3K LOC while the layered approach requires less than 0.4K LOC.

7.3 Verification Evaluation

Verification effort. MoLi reuses the code from CRL-H [101] framework, including the support for C language and concurrency reasoning. MoLi's extension mainly devotes to the logic for termination and the model of non-atomic abstract operations, which is about 3K LOC. Table 1 shows the lines of proof for verifying RefFS. RefFS also reuses AtomFS's internal functions inside the critical section and their proofs, except that now we also verify termination.

Trusted computing base and tests. Our work has some trusted parts. The abstraction of RefFS is trusted. VFS, FUSE, C compiler, C implementation of a lock and memory allocator of glibc are trusted. Termination proof assumes a fair scheduler and a sequentially consistent hardware model. We also test RefFS with xfstests, a comprehensive file system testing suite, which reports no bugs.

Limitations. Our prototypes of MoLi and RefFS still have limitations. Currently, RefFS is an in-memory file system and does not consider crashes. In general, the reasoning for termination is orthogonal to crashes because the recovery procedure will restore the state to re-execute the code. Hence, the termination proof still applies to the non-crash setting. When a crash happens in the middle of a program, what MoLi does not consider is the termination of the recovery procedure, whose precondition is the crashed state. To support crash safety, one may combine the techniques in DaisyNFS [18–20].

MoLi does not support reasoning about termination in the presence of interrupts or exceptions. Similar to crash safety, supporting them requires considering intermediate states.

RefFS has simplified read access to use exclusive locks. Nevertheless, we can use read-write locks in RefFS and reason with their implementations in MoLi.

7.4 Bug Discussion

We discuss whether MoLi can help find practical bugs. Non-concurrent termination bugs such as logic and low level programming errors [59, 83] will fail the proof, because one cannot define a well-founded metric that decreases for each iteration. In AA-deadlocks [25,58,72], when a thread requests for a lock again, the layer of the waited action (definite release of the lock by other threads) is not larger than that of enabled action (definite release of the lock by the current thread), which will violate the dependency condition. In deadlocks caused by nested blocking [6,23,39,97], proof authors either fail to define the layers, or define the wrong layers, later finding that the layers cannot pass the dependency condition proof.

For deadlocks with dynamic orders [3, 8, 55, 96], MoLi allows defining state-dependent dynamic layers to precisely represent such orders. Therefore, such bugs can be discovered during proofs. For deadlocks that involve ad-hoc synchronization [24,52,69], MoLi's general notion of definite actions can specify and verify them, similar to the bug types above.

These bug patterns also exist in other domains, e.g., memory management [27] and network [45] subsystems in an OS, database and web applications [68]. Therefore, we believe MoLi is also applicable to these domains.

8 Related Work

Starting from the seminal work of seL4 [57], these years have witnessed tremendous progress on the verification of systems, including operating systems [40,73,80], distributed systems [43,78,92], file systems [21,22,47,79] and many others [26,28,63,81,88,89]. Yet, only few of them guarantee systems' liveness, and none of the proposed frameworks could be used for concurrent file systems.

VSync [74] relies on a special *await loops* shown in lock implementations, and proposes *await model checking* to automatically verify the termination of lock primitives, even under weak memory models. It does not support general while loops. VSync relies on a specific client library for correctness; the client library may still not cover all corner cases, especially for large-scale systems such as file systems.

CCAL [41] has been used to verify the termination of an MCS lock [56] by organizing the implementation into layers. However, it does not provide a program logic to guide the proofs, so it is unclear how CCAL can be used to prove the termination of concurrent file systems.

Ironfleet [43] verifies distributed systems with a blend of TLA and Hoare style automated verification. However, this methodology does not have the power for concurrency and termination verification in file systems. Ironfleet's reduction approach for concurrency verification relies on implementation being atomic, but file systems, e.g., RefFS, are not atomic. On termination, Ironfleet does not consider blocking and dynamic lock dependencies as shown in file systems.

Reference counting, a widely used technique in Linux, has also caused many severe bugs [44]. Various methods (e.g., invariant-based [33,34] and anti-pattern based [44]) have been proposed to detect these bugs. Although effective in practice, they still suffer from false positives and false negatives. Our work verifies the correctness of refcounting by showing the implementation using it can refine an abstraction where its details are hidden.

9 Conclusion

This paper has presented MoLi for verifying concurrent file systems. It supports the dynamically layered definite releases specification, with which we verify RefFS, the first modularly verified concurrent file system with termination guarantee. We also for the first time formally prove the correctness of directory locking scheme in Linux VFS. The formal specification has helped us uncover real Linux bugs in practice.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments. We are especially grateful to our shepherd Marc Shapiro, whose suggestions significantly improved this paper. We thank Xinyu Feng and Hongjin Liang for discussions on the metatheory of MoLi. This work is supported by the National Natural Science Foundation of China (No. 61925206 and 62132014). Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

References

[1] Paulo Alcantara. cifs: fix potential deadlock in cache_refresh_path(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9fb0db40513e27537fde63287aea920b60557a69, 2023.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distrib. Comput.*, 2(3):117–126, sep 1987.

[3] Josef Bacik. btrfs: drop path before adding new uuid tree entry. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9771a5cf937129307d9f58922d60484d58ababe7, 2020.

[4] Josef Bacik. btrfs: fix lockdep splat in add_missing_dev. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=fccc0007b8dc952c6bc0805cdf842eb8ea06a639, 2020.

[5] Josef Bacik. btrfs: fix potential deadlock in the search ioctl. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a48b73eca4ceb9b8a4b97f290a065335dbcd8a04, 2020.

[6] Josef Bacik. btrfs: move the chunk_mutex in btrfs_read_chunk_tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=01d01caf19ff7c537527d352d169c4368375c0a1, 2020.

[7] Josef Bacik. btrfs: open device without device_list_mutex. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=18c850fdc5a801bad4977b0f1723761d42267e45, 2020.

[8] Josef Bacik. btrfs: unlock to current level in btrfs_next_old_leaf. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0e46318df8a120ba5f1e15210c32cfab33b09f40, 2020.

[9] Josef Bacik. btrfs: exclude mmaps while doing remap. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8c99516a8cdd15fe6b64a12297a5c7f52dcee9a5, 2021.

[10] Josef Bacik. btrfs: fix lockdep splat with reloc root extent buffers. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=b40130b23ca4a08c5785d5a3559805916bddba3c, 2022.

[11] Josef Bacik. btrfs: unlock locked extent area if we have contention. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9e769bd7e5db5e3bd76e7c67004c261f7fcaa8f1, 2022.

[12] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *ESOP*, pages 611–639, 2019.

[13] Johann Blieberger, Bernd Burgstaller, and Robert Mittermayr. Static detection of livelocks in Ada multitasking programs. In *Proceedings of the 12th International Conference on Reliable Software Technologies*, Ada-Europe'07, page 69–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *SIGPLAN Not.*, 37(11):211–230, nov 2002.

[15] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 161–169, USA, 2009. IEEE Computer Society.

[16] Miao Cai, Hao Huang, and Jian Huang. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, page 609–633, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*,

SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021.

[20] Tej Chajed, Joseph Tassarotti, Mark Theng, M Frans Kaashoek, and Nickolai Zeldovich. Verifying the daisynfs concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, 2022.

[21] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Zhihao Cheng. ubifs: Fix deadlock in concurrent bulk-read and writepage. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f5de5b83303e61b1f3fb09bd77ce3ac2d7a475f2, 2020.

[24] Zhihao Cheng. ubifs: Fix deadlock in concurrent rename whiteout and inode writeback. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=afd427048047e8efdedab30e8888044e2be5aa9c, 2021.

[25] Zhihao Cheng. ubifs: Fix aa deadlock when setting xattr for encrypted file. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a0c51565730729f0df2ee886e34b4da6d359a10b, 2022.

[26] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Hugh Dickins. mm: lock newly mapped vma with corrected ordering. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1c7873e3364570ec89343ff4877e0f27a7b21a61, 2023.

[28] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. Automated verification of idempotence for stateful serverless applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 887–910, Boston, MA, July 2023. USENIX Association.

[29] Linux documentation. Kernel subsystem documentation » filesystems in the linux kernel » directory locking. https://www.kernel.org/doc/html/latest/filesystems/directory-locking.html, 2023. Referenced December 2023.

[30] Linux documentation. Kernel subsystem documentation » filesystems in the linux kernel » pathname lookup. https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html, 2023. Referenced December 2023.

[31] Linux documentation. Locking in the kernel » runtime locking correctness validator. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html, 2023. Referenced April 2023.

[32] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), nov 2021.

[33] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2009.

[34] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, page 352–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 315–327, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Filebench. Filebench, 2019.

[37] GNU. Gcc, the gnu compiler collection. https://www.gnu.org/software/gcc/, 2019. Referenced April 2019.

[38] Google. syzkaller - kernel fuzzer, 2023.

[39] Andreas Gruenbacher. gfs2: Fix deadlock between gfs2_{create_inode,inode_lookup} and delete_work_func. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=dd0ecf544125639e54056d851e4887dbb94b6d2f, 2020.

[40] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, 2016. USENIX Association.

[41] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 646–661, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Chunhai Guo. erofs: avoid infinite loop in z_erofs_do_read_page() when reading beyond eof. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8191213a5835b0317c5e4d0d337ae1ae00c75253, 2023.

[43] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Liang He, Purui Su, Chao Zhang, Yan Cai, and Jinxin Ma. One simple api can cause hundreds of bugs an analysis of refcounting bugs in all modern linux kernels. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 52–65, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Allison Henderson. net:rds: Fix possible deadlock in rds_message_put. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f1acf1ac84d2ae97b7889b87223c1064df850069, 2024.

[46] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, page 124–133, USA, 2013. IEEE Computer Society.

[47] Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using disksec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 323–338, Carlsbad, CA, 2018. USENIX Association.

[48] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[49] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983.

[50] D. Jones. Trinity: A linux system call fuzz tester, 2019.

[51] Horatiu Jula, Daniel M Tralamazza, Cristian Zamfir, George Candea, et al. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, volume 8, pages 295–308, 2008.

[52] Jan Kara. ext4: fix deadlock with fs freezing and ea inodes. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=46e294efc355c48d1dd4d58501aa56dac461792a, 2020.

[53] Jan Kara. fs: Lock moved directories. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=28eceeda130f5058074dd007d9c59d2e8bc5af2e, 2023.

[54] Linux kernel stable tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/log/, 2024.

[55] Hyeong-Jun Kim. f2fs: compress: fix potential deadlock of compress file. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=7377e853967ba45bf409e3b5536624d2cbc99f21, 2021.

[56] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of mcs lock—layer by layer. In *Asian Symposium on Programming Languages and Systems*, pages 273–297. Springer, 2017.

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Konstantin Komarov. fs/ntfs3: Changing locking in ntfs_rename. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0ad9dfcb8d3fd6ef91983ccb93fafbf9e3115796, 2022.

[59] Greg Kurz. fuse: Fix infinite loop in sget_fc(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e4a9ccdd1c03b3dc58214874399d24331ea0a3ab, 2021.

[60] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, 1977.

[61] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, 2009.

[62] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, pages 407–426, 2010.

[63] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, 2022.

[64] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 385–399, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[66] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[67] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 31–44, USA, 2013. USENIX Association.

[68] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[69] Filipe Manana. btrfs: fix deadlock when cloning inline extent and low on free metadata space. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3d45f221ce627d13e2e6ef3274f06750c84a6542, 2020.

[70] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377. ACM, 2015.

[71] mit pdos. mit-pdos/fscq: Fscq is a certified file system written and proven in coq. https://github.com/mit-pdos/fscq, 2019. Referenced April 2019.

[72] Trond Myklebust. Nfs: Don't deadlock when cookie hashes collide. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=648a4548d622c4ae965058db1a6b5b95c062789a, 2022.

[73] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[74] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 530–545, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, may 1976.

[76] Bob Peterson. gfs2: fix a deadlock on withdraw-during-mount. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=865cc3e9cc0b1d4b81c10d53174bced76decf888, 2021.

[77] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 1–15, New York, NY, USA, 1991. Association for Computing Machinery.

[78] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[79] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, 2016. USENIX Association.

[80] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 287–305, Carlsbad, CA, 2018. USENIX Association.

[81] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 866–881, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Theodore Ts'o. ext4: add error checking to ext4_ext_replay_set_iblocks(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1fd95c05d8f742abfe906620780aee4dbe1a2db0, 2021.

[83] Theodore Ts'o. ext4: add error checking to ext4_ext_replay_set_iblocks(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1fd95c05d8f742abfe906620780aee4dbe1a2db0, 2021.

[84] Theodore Ts'o and Stephen Tweedie. Future directions for the ext2/3 filesystem. In *Proceedings of the USENIX annual technical conference (FREENIX track)*, 2002.

[85] vim. welcome home: vim online. https://www.vim.org, 2019. Referenced April 2019.

[86] Al Viro. rename(): avoid a deadlock in the case of parents having no common ancestor. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a8b0026847b8c43445c921ad2c85521c92eb175f, 2023.

[87] Al Viro. rename(): fix the locking of subdirectories. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=22e111ed6c83dcde3037fc81176012721bc34c0b, 2023.

[88] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 249–262, Carlsbad, CA, July 2022. USENIX Association.

[89] Jiawei Wang, Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, Jitang Lei, Viktor

Vafeiadis, Hermann Härtig, and Haibo Chen. BWoS: Formally verified block-based work stealing for parallel processing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 833–850, Boston, MA, July 2023. USENIX Association.

[90] Wengang Wang. ocfs2: fix deadlock between setattr and dio_end_io_write. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=90bd070aae6c4fb5d302f9c4b9c88be60c8197ec, 2021.

[91] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, volume 8, pages 281–294, 2008.

[92] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[93] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, page 602–629, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Darrick J. Wong. xfs: fix s_maxbytes computation on 32-bit kernels. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=932befe39ddea29cf47f4f1dc080d3dba668f0ca, 2020.

[95] Darrick J. Wong. xfs: more lockdep whackamole with kmem_alloc*. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=6dcde60efd946e38fac8d276a6ca47492103e856, 2020.

[96] Darrick J. Wong. xfs: fix an abba deadlock in xfs_rename. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=6da1b4b1ab36d80a3994fd4811c8381de10af604, 2021.

[97] Chao Yu. f2fs: compress: fix potential deadlock. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3afae09ffea5e08f523823be99a784675995d6bb, 2021.

[98] Mo Zou. Re: [PATCH] Documentation: fs: fix directory locking proofs. https://lore.kernel.org/linux-fsdevel/CAHfrynPiUWiB0Vg3-pTi_yC6cER0wYMmCo_V8HZyWAD5Q_m+jQ@mail.gmail.com/, 2023.

[99] Mo Zou. Directory locking proof for Linux VFS. https://ipads.se.sjtu.edu.cn/pub/projects/reffs, 2024.

[100] Mo Zou. The soundness proof of MoLi. https://ipads.se.sjtu.edu.cn/pub/projects/reffs#soundness_proof, 2024.

[101] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

Anvil: Verifying Liveness of Cluster Management Controllers

Xudong Sun[†], Wenjie Ma[†], Jiawei Tyler Gu[†], Zicheng Ma[†], Tej Chajed[‡],
Jon Howell[◇], Andrea Lattuada[◇], Oded Padon[◇], Lalith Suresh[⋆], Adriana Szekeres[◇], Tianyin Xu[†]

[†]University of Illinois Urbana-Champaign [‡]University of Wisconsin-Madison
[◇]VMware Research [⋆]Feldera

Abstract

Modern clouds depend crucially on an extensible ecosystem of thousands of controllers, each managing critical systems (e.g., a ZooKeeper cluster). A controller continuously *reconciles* the current state of the system to a desired state according to a declarative description. However, controllers have bugs that make them never achieve the desired state, due to concurrency, asynchrony, and failures; there are cases where after an inopportune failure, a controller can make no further progress. Formal verification is promising for avoiding bugs in distributed systems, but most work so far focused on safety, whereas reconciliation is fundamentally not a safety property.

This paper develops the first tool to apply formal verification to the problem of controller correctness, with a general specification we call *eventually stable reconciliation*, written as a concise temporal logic liveness property. We present Anvil, a framework for developing controller implementations in Rust and verifying that the controllers correctly implement eventually stable reconciliation. We use Anvil to verify three Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit, which can readily be deployed in Kubernetes platforms and are comparable in terms of features and performance to widely used unverified controllers.

1 Introduction

Modern clouds are powered by cluster managers such as Kubernetes [12], Borg [89], ECS [73] and Twine [87]. These systems manage large-scale cluster resources and all applications running atop them. Architecturally, these systems comprise a collection of *controllers* that implement all the cluster-management logic based on the *state reconciliation* principle [4, 30]. Controllers are loosely coupled microservices, each monitoring the cluster state and continuously reconciling the *current* cluster state to match a *desired* state. In Kubernetes for example, controllers manage everything from system resources (e.g., pods, data volumes, networking, and stateful services) to application lifecycles (e.g., provisioning, upgrades, and scaling). There is a thriving ecosystem of thousands of domain-specific controllers that extend Kubernetes [46, 47, 60, 81, 84]. All these controllers perform critical operations, making their correctness paramount.

Implementing correct controllers is immensely challenging, due to the enormity of cluster state space and the complexity of failure events (e.g., node crashes, network interruptions, and asynchrony issues). Recent automated controller testing tools [44, 85] found many bugs with severe consequences such as system outages, data loss, and resource leaks in popular Kubernetes controllers. Buggy controllers have caused many production incidents [45, 57, 71, 74].

This paper addresses the controller correctness challenge with two major contributions: (1) *eventually stable reconciliation*, a general specification for controller correctness which we develop as a liveness property, and (2) Anvil, a framework for implementing practical controllers and formally verifying that a controller implements eventually stable reconciliation. We have developed and verified practical Kubernetes controllers for managing critical systems using Anvil.

Challenges and contributions. Addressing controller correctness with formal verification poses several challenges. The first challenge is to define a correctness specification that is generally applicable to diverse controllers, powerful enough to preclude a broad range of bugs, and concise enough for manual inspection, together with appropriate assumptions that make it possible to implement the specification. The second challenge lies in proving that the controller implements this specification: controllers are complex, feature-rich real-world systems that do not have pen-and-paper proofs that we can reference. This problem is exacerbated by the fact that controllers run in a complex and dynamic environment, where the controller must handle unexpected faults, asynchrony, and conflicts when interacting with other controllers.

We present Eventually Stable Reconciliation (ESR) as a general specification of controller correctness (§3). ESR is a liveness property, which states that a controller should *eventually* reconcile the cluster to a desired state, and then *always* keep the cluster in the desired state. ESR captures the essential functionality that controllers should provide in a precise language, and it precludes a broad range of bugs caused by factors like inopportune failures and conflicts with other controllers. ESR is also realistic and captures the necessary premise to reach the desired state. We formalize ESR as a concise Temporal Logic of Actions (TLA) [58] formula.

A common challenge in proving liveness is that the proof depends on subtle *fairness assumptions*, including assumptions about possible faults. Overly strong assumptions (e.g., the controller can crash at most once) lead to weak correctness guarantees, and overly weak assumptions (e.g., the controller can keep crashing forever) make liveness verification untenable. Anvil employs an assumption that covers a broad range of fault scenarios—an arbitrary number of faults can happen, but eventually faults stop happening. This assumption is similar in spirit to partial synchrony [40] but for faults.

To prove that a controller satisfies ESR, one must consider the controller's interactions with the environment in which it runs. Anvil models this environment, including the shared cluster state, asynchronous network, other controllers, and a realistic fault model (§4.3). The environment model also encodes assumptions on fair scheduling and faults. Anvil abstracts general liveness reasoning patterns in the environment into reusable lemmas to reduce proof effort (§4.4).

With the reusable models and lemmas provided by Anvil, developers can prove that the controller makes progress from any cluster state towards potential desired states in the presence of asynchrony, faults, and conflicts with other controllers. We present a proof strategy to disentangle the challenges of proving ESR (§5), which divides the proof into two lemmas: (1) starting from any possible state resulting from potential interleaving of previous execution and faults, the controller progresses towards the desired state in a stable environment, and (2) the environment eventually becomes stable. Both lemmas can be proven using the temporal proof rules that Anvil provides (under the fairness assumptions). We have applied this proof strategy to verify three controllers using Anvil.

Implementation. We implemented Anvil for verifying Kubernetes controllers on top of Verus [61], an SMT-based deductive verification tool for Rust. With Verus, developers can implement controllers in Rust and formally verify their implementations. Verus does not support temporal logic reasoning, so Anvil provides a TLA embedding on top of first-order logic (§4.2) to enable TLA-style temporal reasoning.

We used Anvil to implement three practical Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit (§6). These controllers can readily be deployed in real-world Kubernetes platforms; they provide feature parity and competitive performance w.r.t. existing mature, widely used (but unverified) controllers. The verification effort is manageable, with the proof-to-code ratio ranging from 4.5 to 7.4 across the controllers. The verification process exposed deep bugs in both our early implementations and unverified reference controllers. Although Anvil is primarily designed for liveness verification, it also supports safety verification; we prove a safety property specific to the RabbitMQ controller: the controller *never* performs unsafe scaling operations.

Summary. This paper makes the following contributions:

- Eventually stable reconciliation (ESR), a general specification for controller correctness as a liveness property;
- Anvil, a framework for developing practical controllers and formally verifying that the controller implementations satisfy correctness properties such as ESR;
- three representative and practical Kubernetes controllers verified using Anvil; and
- an evaluation of the end-to-end correctness and performance of the three verified controllers.

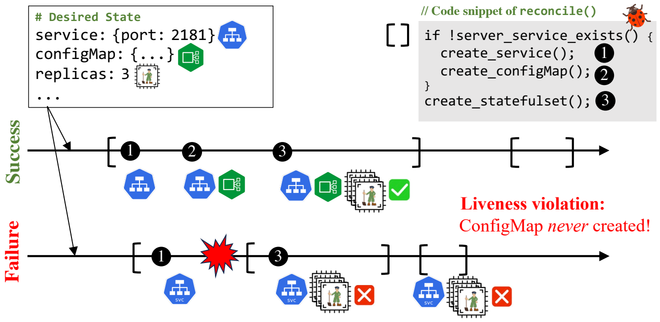
Anvil and the verified controllers are available at `https://github.com/vmware-research/verifiable-controllers`.

2 Implementing Controllers

Controllers follow the *state reconciliation* principle: each controller runs a control loop that continuously reconciles the cluster's current state to the desired state [4, 8]. At each loop iteration, a *reconciliation* procedure checks whether the current cluster state matches the desired state; if not, it performs corrective operations to move the cluster towards the desired state (e.g., launching new replicas in an ensemble of servers when existing replicas fail). The operations query or update the cluster state, represented by shared data objects. These state objects are exposed by REST-based API servers and are stored in a logically centralized data store like etcd [1]. The desired cluster state is described declaratively and can be dynamically updated during the lifecycle of a running controller. The reconciliation procedure is typically implemented in a `reconcile()` function, which is invoked whenever the desired state description (or its relevant cluster states) is changed.

Figure 1 exemplifies the reconciliation process of a Kubernetes controller for managing ZooKeeper. To create a ZooKeeper cluster, the controller takes three steps to create: ❶ a networked service (a Kubernetes service object [17]), ❷ a ZooKeeper configuration (a config map object [14]), and ❸ a stateful application (a stateful set object [18]) with three replicas. Each step is performed by creating a new state object of the corresponding resources via the Kubernetes API, which then triggers Kubernetes built-in controllers, e.g., the StatefulSet controller will create three sets of pods and volumes to run containerized ZooKeeper nodes. In the end, the cluster state matches the desired state. Later, if the desired state changes (e.g., its `replicas` is increased), the ZooKeeper controller will start a new iteration of reconciliation that updates the stateful set object to scale up the ZooKeeper cluster.

Correctness challenges. A bug in a controller's reconciliation can result in the controller *never* being able to match the desired state, even when `reconcile()` is called repeatedly. Controllers are expected to be level-triggered [53]: `reconcile()` can be called from any current cluster state to match any given desired state, with no guarantee that the controller has seen the entire history of cluster state changes [86]. In addition, controllers must tolerate unexpected failures and asynchrony while running `reconcile()`, which leads to a state-space explosion that makes testing controllers difficult. Figure 1 shows one of many bug patterns



```
# Desired State
service: {port: 2181}
configMap: {...}
replicas: 3
...
```

// Code snippet of reconcile()

```
if !server_service_exists() {
  create_service();   ❶
  create_configMap(); ❷
}
create_statefulset(); ❸
```

Figure 1: **An example of state reconciliation of an unverified ZooKeeper controller.** A liveness bug is triggered by a crash during reconciliation. The bug pattern is common in real-world Kubernetes controllers (known as intermediate-state bugs [85]).

```
pub trait Controller {
  type D; // desired cluster state description
  type S; // local state in the state machine

  /// Returns the initial local state (in the state
  /// machine) of every reconcile()
  fn initial_state() -> S;

  /// Returns S: next local state in state machine
  /// Req: external request (e.g. to Kubernetes)
  /// # Arguments
  /// * d: the desired cluster-state description
  /// * r: response to the request from last step
  /// * s: current local state in state machine
  fn step(d: &D, r: Resp, s: S) -> (S, Req);

  /// Returns true if all steps are done
  fn done(s: &S) -> bool;

  /// Returns true for error states
  fn error(s: &S) -> bool;
} // other advanced APIs are omitted
```

Figure 2: **Anvil's basic Controller API.** To implement a controller, developers implement the Controller trait.

```
pub fn reconcile<C>(d: C::D) -> Result<Action, Error>
  where C: Controller {
  let mut s = C::initial_state();
  let mut resp = None;
  loop { // exercise the state machine
    if C::error(&s) {
      return Err(ErrorNeedsRequeue);
    } else if C::done(&s) {
      return Ok(requeue(timeout));
    }
    let (next_s, req) = C::step(&d, resp, s);
    resp = send_external_request::<C>(req);
    s = next_s;
  }
} // details like validity checks are omitted
```

Figure 3: **Anvil code that assembles `reconcile()` using the Controller API in Figure 2.**

```
fn step(d: &ZKD, r: Resp, s: ZKS) -> (ZKS, Req) {
  match s {
    CheckService => { // if the service exists
      let service_get_req = KubeGet { ... }
      return (ReconcileService, service_get_req);
    }
    ReconcileService => {
      /// create/update the service based on response r
      if r.is_ok() {
        let service_update_req = ...;
        return (CheckConfigMap, service_update_req);
      } else if r.is_not_found() {
        let service_create_req = ...;
        return (CheckConfigMap, service_create_req);
      } else {
        return (Error, Noop); // restart reconcile()
      }
    }
    CheckConfigMap => { ... }
    ReconcileConfigMap => { ... }
    CheckStatefulSet => { ... }
    ReconcileStatefulSet => { ... }
    ...
  } // more step branches are omitted
}
```

Figure 4: **A simplified implementation of `step()` using Anvil for creating a ZooKeeper cluster.** Proof-related code is omitted.

of controllers [35, 44, 54, 64, 85]. If the controller crashes between steps ❶ and ❷ during an execution of `reconcile()`, Kubernetes will reboot the controller. The freshly invoked `reconcile()` call now faces the intermediate state created by the previous failed execution (❶). However, in this case, the controller would *never* perform ❷ due to a buggy predicate, which only checks whether the networked service exists, but not whether the config map also exists. As a result, the cluster state would never match the desired state—a liveness violation. Such liveness violations are notoriously hard to detect by testing or model checking [55].

Implementing controllers with Anvil. In Anvil, developers implement a controller using a state machine; this style is common practice in unverified controllers as well [2, 6], and in Anvil it enables TLA-style verification. Figure 2 shows a snippet of the Anvil Controller API specified using a Rust trait: it involves defining the initial state and the transitions of a state machine. Anvil's `reconcile()` uses the state machine as shown in Figure 3: it starts from the initial state and invokes `step()` iteratively until all steps are done or if any step encounters an error. Each iteration of `step()` returns the next state in the state machine, together with an external request. The external request is typically a REST call to Kubernetes APIs, but can also be extended to non-Kubernetes APIs (§6.1). The response to the external request is passed as an argument to the next iteration of `step()`. Note that the API enforces no more than one external request per `step()`, making the state-machine transition atomic with respect to cluster-state changes. Anvil's `reconcile()` interfaces a trusted Kubernetes client library (kube-rs [11]) which invokes `reconcile()` upon changes, handles its output, and requeues the next invocation.

Figure 4 shows the `step()` implementation of a ZooKeeper controller (Figure 1). The `step()` function takes the desired state description of the ZooKeeper cluster (`d`), the response (`r`) to the request from last step (if any), and the current local state (`s`), and deterministically returns the next local state and the external request. The state machine starts from the `CheckService` state, where it returns a request to read the service object [17] from the Kubernetes API (`service_get_req`) and the next state to transition to `ReconcileService`. The `reconcile` method (Figure 3) fetches the service object using the Kubernetes API, and moves on to the next iteration of `step()`, bringing the state machine to `ReconcileService` branch. The controller proceeds to create or update the service based on the response of `service_get_req`. In this way, the controller progressively reconciles each cluster-state object and eventually matches the desired state declared by `ZKD`.

Next, we present a general controller correctness specification, termed eventually stable reconciliation, in §3, and then explain how to verify it using Anvil in §4 and §5.

3 Eventually Stable Reconciliation (ESR)

Controllers reconcile the cluster state to match the desired state. While the details vary between controllers, and some controllers may have additional correctness guarantees, we formalize a general property called *eventually stable reconciliation (ESR)* that captures this ubiquitous pattern.

ESR captures two key properties of any controller's state reconciliation behavior: (1) *progress*: given a desired state description, the controller must eventually make the cluster state match that desired state (unless the desired state changes), and (2) *stability*: if the controller successfully brought the cluster to the desired state, it must keep the cluster in that state (unless the desired state changes).

To make our specification general, we do not wish to commit to any particular bound on the time or number of operations the controller takes to bring the cluster to the desired state. So, we want to talk about guaranteed *eventualities*. Such unbounded eventualities are naturally described using *temporal logics* [80]. We use TLA (temporal logic of actions) [58], a linear-time temporal logic well-suited to our needs. TLA is designed for reasoning about a system described as a state machine. The behavior of the state machine is captured by its set of traces, infinite sequences of system states where the first state is a valid initial state and each subsequent state is obtained via a valid transition from the previous state.

We formalize ESR as a TLA formula that should hold for *all* traces of the system's execution, where the system includes both the controller and its environment, under all possibilities for asynchrony, concurrency, and faults (e.g., controller crashes). We use d to denote a state description, $\mathtt{desire}(d)$ to denote whether d is the current description of the desired state, $\mathtt{match}(d)$ to denote whether the current cluster state matches the description d. Our definition of ESR is given by the following formula:

$$\forall d. \square \big(\square \mathtt{desire}(d) \Rightarrow \Diamond \square \mathtt{match}(d) \big). \qquad (1)$$

Informally, ESR asserts that if at some point the desired state stops changing, then the cluster will eventually reach a state that matches it, and stay that way forever. The temporal operators $\Diamond$ (eventually) and $\square$ (always) are used in temporal logics to reason about the future of an execution trace. If a predicate P talks about the current state, then $\square P$ says that P holds in the current state and all future states, while $\Diamond P$ says that P holds in the current or some future state. Temporal logics such as TLA also allow nesting of temporal operators; for example, $\Diamond \square P$ means that eventually we get to a point such that from that point onwards, P always holds.

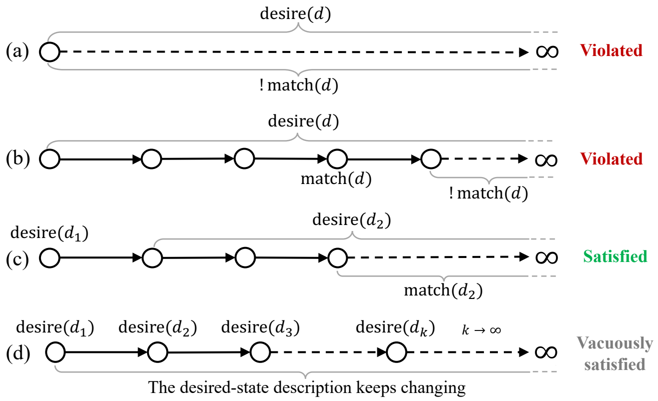


Figure 5: **Executions that violate or satisfy ESR.**

The formalization of ESR is a key contribution of this paper in that it captures the key correctness properties shared by virtually all controllers: progress and stability. We elaborate on this with a detailed dissection of eq. (1).

The innermost conclusion of the formula is $\Diamond \square \mathtt{match}(d)$, which states that eventually ($\Diamond$) the controller matches the desired state (progress), and from then on, it always ($\square$) keeps the cluster state at the desired state (stability). In front of this expression, $\square \mathtt{desire}(d)$ is a realistic and necessary premise for the controller to match the desired state—if the desired state description keeps changing forever, the controller will keep chasing a moving target forever, and nothing can be guaranteed as we do not wish to assume a bound on how long state reconciliation takes. The outer $\square$ in eq. (1) says that $\square \mathtt{desire}(d) \Rightarrow \Diamond \square \mathtt{match}(d)$ always holds, meaning that the controller continuously reconciles the cluster state *regardless of its past execution*. Finally, the $\forall d$ states that the controller reconciles all desired state descriptions.

Figure 5 illustrates the ESR definition in some examples, some that satisfy the definition and others that do not: (a) violates progress because the cluster state never matches d, (b) violates stability because the cluster state first matches d but then deviates from d, (c) satisfies ESR because the cluster state eventually matches and always matches d_2, and (d) vacuously satisfies ESR because the desired state never stops changing, so $\square \mathtt{desire}(d)$ does not hold for any fixed d.

The verification goal for each controller is to prove that the controller satisfies ESR—*all* possible executions of the controller satisfy ESR. We use `model` to describe all possible executions of the controller that runs in an environment with asynchrony, concurrency and faults. We use $\leadsto$ (leads-to) notation to simplify the presentation of the ESR property, where $P \leadsto Q$ means $\square(P \Rightarrow \Diamond Q)$. Then the statement that the controller satisfies ESR is formalized as:

$$\mathtt{model} \models \forall d. \square \mathtt{desire}(d) \leadsto \square \mathtt{match}(d). \qquad (2)$$

The power of ESR. Strictly speaking, ESR (eq. (1)) only guarantees one successful state reconciliation—the one that happens after the desired state stops changing forever. However, in practice the controller has no way of knowing if the desired state will change in the future or not. Therefore, we can expect that a controller that satisfies ESR will bring the cluster to match the desired state (and keep it like that) for any desired state that remains unchanged for *long enough*. ESR achieves this without getting into the gory details of defining exactly how long is long enough. Note further that because of the outermost □ in eq. (1), a controller that satisfies ESR will deliver *multiple* successful state reconciliations, assuming that the desired state goes through a series of slow changes.

Our analysis shows that ESR can ensure the absence of a broad range of controller bugs [44, 64, 85]. For example, recent testing tools [44,85] detected 70 bugs across 16 popular controllers that the controller never matches the desired state due to improper handling of corner-case state descriptions, inopportune failures and concurrency issues, which consist of 69% of all the detected bugs. All such bugs are precluded by ESR. Prior work [64] also reported failure patterns where the cluster state, after matching a desired state, then deviates due to conflicting interactions with other controllers. Such bugs, as stability violations, are also precluded by ESR.

4 Anvil

Anvil is a framework for developing controllers and mechanically proving that they implement correctness specifications like ESR. Anvil is built on top of Verus [61], an SMT-based deductive verification tool for Rust backed by Z3 [39], in similar spirit to Dafny [62]; it offers a Hoare-logic [52] framework for reasoning modularly about imperative code in Rust.

Figure 6 shows the workflow of using Anvil to verify a controller. The developer first provides 🅐 a *controller model* (an abstract state machine) and then proves two theorems: 🅑 the Controller trait implementation (Figure 4) conforms to the controller model and 🅒 the controller model, together with a model of the environment (e.g., the network, other controllers, faults), satisfies specifications like ESR (eq. (2)).

Writing the controller model and verifying the implementation conforms to the model are straightforward. The controller model is an abstract state machine with the same structure as the implementation state machine. To prove conformance, developers prove that each step in the implementation corresponds to exactly one step in the model using standard Floyd-Hoare style reasoning (§4.1). Note: the controller model is written in Verus' specification language to enable verification.

Verifying the model entails ESR is more challenging: developers need to apply temporal logic reasoning on the interaction between the controller and its environment (including faults) at the model level to prove ESR. To reduce developers' burden on specification and proof, Anvil provides (1) a TLA embedding (§4.2) that defines temporal logic operators on top of first-order logic to enable specification and proof in temporal logic (Verus does not support temporal logic), (2) a model of the controller environment (§4.3), including components that a controller interacts with, faults that a controller must tolerate, and reasonable assumptions on fair scheduling and faults that controller liveness depends on, and (3) reusable lemmas (§4.4) that encode temporal proof rules and liveness and safety properties of the interactions between a controller and the environment; these lemmas can be directly assembled into developers' ESR proofs.

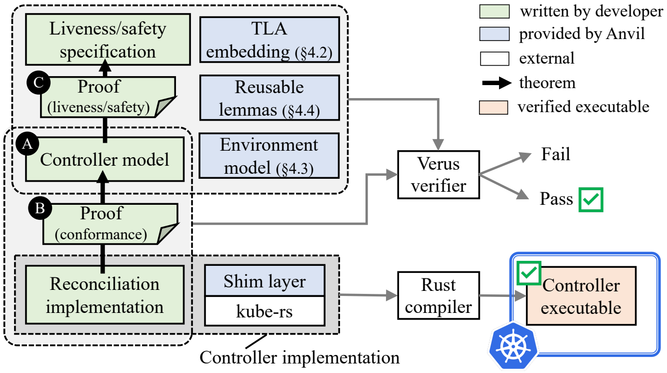


Figure 6: **An overview of Anvil's workflow.**

In this section, we explain how Anvil supports the verification of controllers and then present a general, effective strategy for developing proofs to verify ESR in §5.

Assumptions. Anvil relies on the following assumptions: (1) The TLA embedding correctly defines TLA concepts [58]. (2) The controller environment model correctly describes the interactions between the controller and its environment. (3) The specification of the unverified APIs for querying and updating the cluster state correctly describes the behavior of these APIs. (4) The verifier (Verus and Z3), the Rust compiler, and the underlying operating system are correct.

4.1 Controller Model

To verify controller correctness, developers first write a controller model and prove the controller implementation conforms to this model, similar to prior work [49, 51]. The controller model is a mathematical, state-machine representation of the imperative controller implementation, which abstracts the data types in the implementation and enables TLA-style verification. Given the proof of implementation-model conformance, the model is not assumed to be correct in Anvil's overall verification guarantee.

Anvil provides an API for developers to write the controller model, shown in Figure 7. This API defines a state machine and is similar to the Controller API in Figure 2, except that all the methods and variables are written in *ghost code* [61, 62]. Ghost code is auxiliary code that describes properties of programs and is used for verification only—the code is erased before compilation and thus poses no runtime overhead. Concretely, in the controller model, all the methods are Verus' `spec` functions which are purely functional, and all the variables are ghost types that represent an abstract view of the variables in the implementation, e.g., a heap-allocated Rust `Vec` is represented as a mathematical sequence (Verus' `Seq`).

```
pub trait ControllerModel {
  type DV; // view of the desired state description
  type SV; // view of local state in the state machine
  spec fn m_initial_state() -> SV;
  spec fn m_step(d: DV, r: RespV, s: SV) -> (SV, ReqV);
  spec fn m_done(s: SV) -> bool;
  spec fn m_error(s: SV) -> bool;
} // other advanced APIs are omitted
```

Figure 7: **Anvil's ControllerModel API.** Developers use the API to write the controller model (an abstract state machine). It mirrors the implementation trait (Figure 2) but is written in ghost code.

Given a controller implementation, writing the controller model is straightforward. Given a `step()` implementation in the Controller API (Figure 2), developers write a corresponding `m_step()` using the ControllerModel API (Figure 7). If the implementation's `step()` returns a Kubernetes-API request, the model's `m_step()` correspondingly returns a ghost-type request (`ReqV`) that queries the Kubernetes API model (§4.3.1). The other trait methods are largely identical to their counterparts in the implementation except for the data types.

For each implementation data type defined by developers, such as the types for the desired state description and the state machine's local state (e.g., `D` and `S` in Figure 2), developers need to define a corresponding ghost type (e.g., `DV` and `SV`), typically by replacing implementation data types with corresponding ghost types. For example, if `D` has a field of Rust `Vec` type, `DV` will have a field of Verus `Seq` type. Developers also need to define a `view()` function that converts an implementation object to the corresponding ghost-type object.

Implementation-model conformance. Developers need to prove that the implementation state machine has the same initial state, transitions and termination conditions as the model state machine through `view()`. Figure 8 shows the theorem to prove conformance for the ZooKeeper controller's `step()` in Figure 4. This theorem states that the model's `m_step()` produces the same output (in ghost types), given the same input (in ghost types) of the implementation's `step()`.

```
fn step(d: &ZKD, r: Resp, s: ZKS) -> (res: (ZKS, Req))
  ensures res@ == ZKControllerModel::m_step(d@, r@, s@)
{ ... } // implementation body is omitted
```

Figure 8: **The conformance theorem written as a postcondition of `step`.** The `step` function is executable (part of the controller implementation). The symbol `@` is a shorthand for `.view()` in Verus, which converts an implementation type into a ghost type.

The key challenge in enabling and automating the conformance proof is to reason about data types defined in external, unverified libraries. For example, the controller implementation needs to use data types that define Kubernetes state objects from the kube-rs [11] library, but Verus cannot directly reason about definitions from unverified libraries. So, Anvil defines wrappers that translate every Kubernetes state-object type to its corresponding ghost type. These wrappers are straightforward to implement and are trusted; Anvil includes unit tests that cover *all* the trusted wrapper methods.

The controller implementation uses the wrapper types instead of raw types from kube-rs, and the model uses the corresponding ghost types. For verification, Verus automatically tracks the wrapper's view (`view()`) through the postconditions of the wrapper methods used in the controller implementation. Verus compares the object's view to the ghost object used in the controller model to check the conformance proof; e.g., to prove the theorem in Figure 8, Verus compares the returned request's view and its counterpart in the model.

With this design, the conformance proof is done by standard Floyd-Hoare style reasoning [52] and is largely automated by Verus. Most of the manual proof effort is the requirement to ask Verus to prove two objects are equal if they have the same properties, e.g., to prove a `Vec`'s view (in the implementation) and the corresponding `Seq` (in the model) are equal.

4.2 TLA Embedding

To enable liveness reasoning on top of Verus, Anvil develops a TLA embedding that models important concepts in TLA. Anvil follows IronFleet [51] and models three major concepts as follows: (1) *an execution* is an infinite sequence of system states encoded as a mapping from natural numbers to states, (2) *a temporal predicate* is a boolean predicate on executions, and (3) *a temporal operator* (e.g., $\Diamond$, $\Box$ and $\leadsto$) is a function that transforms one temporal predicate into another. Every temporal operator is defined using only first-order quantifiers on executions. Suppose `P` is a temporal predicate and `ex` is an execution, `eventually(P)` (resp. `always(P)`) is a temporal predicate that holds true of `ex` if `P` is true on some (resp. all) suffixes of `ex`, that is, at some (resp. all) future time.

With the TLA embedding, developers can specify the theorem that the controller satisfies ESR (eq. (2)) as in Figure 9. The definition of `desire` is typically reused among controllers but can also be extended if more premises are required for liveness. The definition of `match` varies across controllers; e.g., the `match(d)` for the ZooKeeper controller in Figure 4 checks if the service, config map and stateful set exist in the data store and match the desired state description `d` (Figure 10).

```
// model ⊨ ∀d. □desire(d) ⇝ □match(d)
model.entails(
  forall(|d: DV|
    always(desire(d)).leads_to(always(match(d)))
  ))
```

Figure 9: **The ESR theorem specified using the TLA embedding.**

In the style of specifying systems [59], Anvil diligently abstracts away executions: developers model components at the levels of state and action (transition between states), then complete liveness proofs with temporal operators. Essentially, Anvil encourages developers to express concepts as *state*

```
spec fn match(d: ZKDV) -> TemporalPredicate {
  lift(|s: ClusterState| { // lift a state predicate
    let store = s.state_object_data_store;
    store.contains(service_name(d))
    && store.contains(config_map_name(d))
    && store.contains(stateful_set_name(d))
    && store[stateful_set_name(d)].replicas == d.size
    && ... // more statements are omitted
  })
}
```

Figure 10: **The definition of the ZooKeeper controller's `match`.** The temporal predicate, when applied to an execution, checks the first state to see if the state objects exist in `store` and match `d`. `ZKDV` is the view of the ZooKeeper desired state description (`ZKD`).

predicates over individual states or *action predicates* over individual transitions. Developers can convert a state predicate to a temporal predicate using a `lift` function [59]: an execution satisfies the lifted predicate if its first state satisfies the state predicate; lifting an action predicate likewise applies the predicate to the first two states of an execution. For example, the temporal predicate `match(d)` is defined by lifting a state predicate as shown in Figure 10. In this way, developers focus on reasoning about individual states and actions when proving invariants and lift them to temporal predicates when applying temporal proof rules (§4.4.1). This differs from IronFleet which interacts directly with instantiated executions throughout the liveness proof. We present Anvil's temporal reasoning style in detail in §5.2.

4.3 Modeling Controller Environment

To reason about interactions between a controller and its environment, Anvil models the controller environment. The goal is to describe the external behavior of different components in the environment and capture the factors that affect a controller's correctness, including asynchrony, concurrency and faults. To this end, Anvil models the environment as a compound state machine, consisting of individual state machines that depict the behavior of different components, such as the network and the API server, as well as faults. The environment model also comes with reasonable assumptions on fair scheduling and faults that liveness depends on.

4.3.1 Modeling Environment Components

Anvil models the environment as a compound state machine with each inner individual state machine modeling one component that a controller interacts with, including:

- an asynchronous network that delivers messages among components with no ordering guarantees;
- the cluster-state data store and the API server; the cluster state is stored in the logically centralized data store (e.g., etcd [1]) and exposed by the API server which handles the controller's query or update requests;
- other controllers in the environment that might interact with the to-be-verified controller; and
- clients that request desired cluster states; clients can update the desired cluster state at any time.

Anvil embeds the controller model in the compound state machine to reason through the interaction between the controller and its environment. The compound state machine, in each step, chooses one individual state machine and invokes one step of that state machine. All the steps are atomic regarding how the cluster state advances (e.g., the API server only handles one request to update the cluster state in each step).

The compound state machine model naturally captures asynchrony and concurrency challenges for controllers. For example, time-of-check to time-of-use (TOCTOU) issues can happen when the cluster state has changed since the last time the controller queried it, but the controller issues an update based on its stale view of the cluster state.

A model of Kubernetes environment. Anvil models the Kubernetes cluster-state data store as a map that stores state objects. Anvil models Kubernetes API servers' mechanisms for validating and coordinating controller requests, including its multi-version concurrency control mechanism wherein each object is versioned. Requests from the controllers must be validated with a version check to take effect.

Anvil models Kubernetes built-in controllers that interact with other controllers, including (1) the garbage collector [16] which deletes a state object if all of its listed owners have been deleted, (2) the StatefulSet controller [18] which manages stateful applications, and (3) the DaemonSet controller [15] which manages daemons (e.g., for monitoring) on every node.

4.3.2 Modeling Faults

Anvil models common faults that happen in modern clusters as actions in the compound state machine; the compound state machine in each step chooses to either let one component take one step or let one fault happen. Anvil models two types of faults: (1) *controller crash*: the controller can crash and reboot an arbitrary number of times. Each crash makes the controller stay offline for an arbitrary number of steps before it is rebooted. After a crash, the controller loses its internal (in-memory) state and has to start over from the beginning of its reconciliation procedure. (2) *request failures*: any request sent by the controller can fail at any point due to network timeouts or the API server being busy.

4.3.3 Specifying Liveness Assumptions

Liveness verification needs careful assumptions. In a concurrent, asynchronous system, *fairness assumptions* are needed to prove that something eventually happens as it relies on the system and its environment getting a chance to take certain actions—a property that is expected to hold in practice but must be nonetheless explicitly incorporated in our formal assumptions. This problem is especially pronounced for controller liveness: a controller's reconciliation (1) relies on other components' actions to complete, and (2) can be interrupted by faults or conflicting actions from other controllers. Anvil makes assumptions that the environment eventually allows the controller to make progress.

Weak fairness assumptions on actions. Applying the *weak fairness* [58] assumption is effective to make the liveness property hold, without assuming any specific fair scheduling. A weak fairness assumption states that if an action A remains "enabled" (i.e., the action can possibly occur), the action eventually occurs: $\Box\texttt{enabled}(A) \leadsto A$. The predicate $\texttt{enabled}(A)$ is true, if for S (the first state of the execution), there *exists* a next state S' such that $A(S,S')$ is true; that is, it is possible for A to occur and transition to S'.

We include fairness assumptions in the model by assuming weak fairness on the actions of the controller and other components in the environment.

Assumptions on faults. Controller liveness also needs assumptions on faults. If the compound state machine chooses to reboot the controller in every step, the controller will never get a chance to finish reconciliation. However, overly strong assumptions like "the controller crashes only once" lead to weak correctness guarantees. To strike a balance, we assume that faults can happen an arbitrary number of times but eventually stop happening, in the spirit of partial synchrony [40].

To incorporate this assumption, we add a "disable-fault" action for each type of fault to the compound state machine. We then add the weak fairness assumption to disable-fault actions. That is, the disable-fault action eventually happens, after which the corresponding type of fault no longer happens.

Assumptions on other controllers. Controllers share the cluster state and thus can conflict with each other. A controller's liveness relies on conflicts being eventually resolved, which mandates assumptions on other controllers. In Kubernetes as an example, the built-in StatefulSet controller can compete with the target controller forever. Suppose the controller uses a stateful set to manage a stateful application and updates the stateful set to match the desired state description. At the same time, the StatefulSet controller continuously updates the stateful set to publish the current status of each running node. When the two controllers are updating the same object concurrently, only one can succeed [13]. Thus, the environment model can adversarially keep letting the target controller lose the race and never reach the desired state.

Anvil assumes that the StatefulSet controller eventually stops updating the stateful set *until* the target controller updates the stateful set again. Similar to the fault assumption, we add to our model an action (with weak fairness) that disables the built-in StatefulSet controller's updates on a stateful set; the target controller's successful update to this stateful set will enable the StatefulSet controller again. Anvil makes the same assumption on how the built-in DaemonSet controller updates daemon sets.

4.4 Reusable Lemmas

Proving ESR requires applying temporal proof rules to reason about the controller's interaction with the environment. This is challenging in two ways: (1) temporal reasoning does not have good automation because SMT solvers like Z3 lack decision procedures for temporal operators, and (2) the interaction between the controller and the environment is complex and is subject to asynchrony and faults. To reduce developers' proof effort, Anvil provides a library of reusable lemmas that encode (1) commonly used temporal proof rules and (2) generic reasoning patterns in the controller environment.

```
proof fn leads_to_transitive(
  model, P, Q, R: TemporalPredicate
)
  requires
    model.entails(P.leads_to(Q)),
    model.entails(Q.leads_to(R))
  ensures model.entails(P.leads_to(R))
{ ... } // proof body is omitted
```

Figure 11: **The leads-to transitivity lemma.**

4.4.1 Temporal Reasoning Lemmas

Anvil provides temporal reasoning lemmas that encode commonly used proof rules to improve temporal reasoning automation. These lemmas are useful for proving liveness for any controller. One example is the *leads-to transitivity* lemma (Figure 11). It shows that if $P \leadsto Q$ and $Q \leadsto R$, then $P \leadsto R$, all under the same assumption *model*. The proof of this lemma involves using the temporal logic definitions, reasoning about an arbitrary time in an execution where P holds, and showing there exists a corresponding time where R eventually holds (using an intermediate time when Q holds, as guaranteed by the preconditions). In return, the developer can easily invoke the lemma without reference to execution or specific indices (these are hidden in the temporal logic lemmas). The leads-to transitivity lemma is frequently used for chaining leads-to formulas to deduce ESR: in our controllers used as case studies, the lemma is used over 50 times. So far, Anvil includes statements and proofs of 70+ such temporal reasoning lemmas, representing a broad range of temporal reasoning patterns.

4.4.2 Environment Reasoning Lemmas

Environment reasoning lemmas prove liveness and safety properties of the interaction between a controller and the environment. We have developed 60 such lemmas. These lemmas are generic to all controllers, and developers can assemble the lemmas into their proofs. We present a representative lemma derived from Anvil's Kubernetes environment model.

Example lemma on the garbage collector (GC). Developers need to reason about their controller's interaction with the built-in GC (§4.3.1). The GC's job is to delete orphan objects whose owner [21] no longer exists: e.g., a stateful set owns a set of pods, thus deleting the stateful set orphans these pods. The GC can conflict with the controller: (1) after the controller updates the owner of an orphan object, the GC deletes the object due to its stale view [86], and (2) the controller attempts to update an object that was deleted by the GC.

To prove ESR, developers need to prove that eventually the GC stops racing with the controller on the object. To help developers prove that eventually the GC stops trying to delete an object x (as x has an existing owner), Anvil provides a lemma with the precondition that any request from the controller that tries to (re)create or update x sets x's owner to an existing object, and the postcondition that eventually if x exists, it has an existing owner (Figure 12). This lemma saves developers the trouble of reasoning about a long chain of the GC execution, including that the GC eventually sends a request to delete x (if it is an orphan), the network eventually delivers the request, and the API server eventually handles the deletion. This lemma takes 200+ lines of proof code and is used in verifying all of the controllers in §6.

```
proof fn eventually_always_has_an_existing_owner(
  model: TemporalPredicate, x: ObjectKey
)
  requires model.entails(
    always(each_req_sets_an_existing_owner(x))),
    ... // some preconditions on fairness are omitted
  ensures model.entails(
    eventually(always(has_an_existing_owner(x))))
{ ... } // proof body is omitted
```

Figure 12: **The garbage collector lemma.** If each request that tries to create or update x sets x's owner to an existing object, then eventually it is always true that if x exists, it has an existing owner.

5 Proving the ESR Theorem

Proving the ESR theorem requires developers to reason about how the controller makes progress starting from any cluster state towards any desired state. We leverage the opportunity that all controllers follow the state-reconciliation principle and develop a proof strategy for ESR. The proof strategy is realized by temporal reasoning using Anvil's TLA embedding and lemmas. We present the proof strategy for ESR and temporal reasoning with Anvil in detail.

5.1 Proof Strategy for ESR

The key idea of our proof strategy is to divide the proof into two main lemmas by separation of concerns: (1) proving that the environment eventually gets stable, and (2) proving that the controller, starting from *any* state (`any_state()`) resulted from arbitrary previous executions and faults, eventually achieves the desired state in this stable environment. Here an environment is stable if (1) the controller does not conflict with the other controllers, (2) faults do not happen, and (3) the desired state description remains unchanged. The ESR theorem is finally proved by combining the two lemmas using temporal proof rules (e.g., leads-to transitivity). Figure 13 shows the high-level proof structure.

5.1.1 Environment is Eventually Stable

Proving that the environment is eventually stable is straightforward and is largely automated by Anvil's lemmas. For example, developers can directly invoke Anvil's lemma which proves that faults eventually stop happening based on Anvil's assumption of faults (§4.3.3). However, proving that the controller eventually stops conflicting with the other controllers still requires certain controller-specific reasoning. Take the garbage collector (GC) as an example, developers can use Anvil's lemma on the GC (Figure 12) to prove that the GC eventually stops racing with the controller on any object, after they prove that the controller correctly sets the owner of the target objects (required by the GC lemma).

```
proof fn ESR_proof()
  ensures model.entails(forall(|d: DV|
    always(desire(d)).leads_to(always(match(d)))
  )) /* the ESR theorem */ {
  // (1) prove ∀d.model ⊨ □desire(d) ⇝ stable_model(d)
  env_is_eventually_stable();
  // (2) prove ∀d.stable_model(d) ⊨ any_state() ⇝ □match(d)
  liveness_in_stable_env();
  // (3) prove model ⊨ ∀d.□desire(d) ⇝ □match(d)
  ...
  leads_to_transitive(...);
}

proof fn env_is_eventually_stable() // lemma 1
  ensures forall |d| model.entails(
    always(desire(d)).leads_to(stable_model(d))) {...}

proof fn liveness_in_stable_env() // lemma 2
  ensures forall |d| stable_model(d).entails(
    any_state().leads_to(always(match(d)))) {...}
```

Figure 13: **High-level structure of the ESR proof.** `model` describes the original environment in §4.3. `stable_model`(d) describes the stable environment: faults and conflicts stop, and the desired state d is stable.

A notable corner case emerges due to asynchrony: even if the desired state description remains unchanged, the controller could still be affected by an older version of the desired state. Consider an execution where the controller crashes right after sending a request to match d_1, then the desired state description is updated to d_2 and remains unchanged from then, but the old request for d_1 is still pending in the network. After the restarted controller sends a new request to match d_2, the two requests will conflict with each other—the two requests try to make the cluster state match two different versions of the desired state. To address this problem, we prove that after the desired state description eventually stabilizes, any controller request for any previous version of the desired state will eventually leave the network.

5.1.2 Liveness in a Stable Environment

Within the stable environment, developers focus on proving that the controller reaches the desired state through each reconciliation step, without considering faults or conflicts.

The main challenge is to prove liveness starting from any possible state. The state here includes both the shared cluster state and the controller's internal state: the cluster state can result from any possible interleaving between the controller's previous execution and arbitrary faults, and the controller internally can be running any reconciliation step.

It is tedious to reason about different executions starting from every internal state. For the ZooKeeper controller in Figure 4, it would require reasoning about controller executions starting from `CheckService`, `ReconcileService` and all other branches in `step()`, respectively. To reduce proof burden, we organize the proof in three stages (Figure 14). First, we

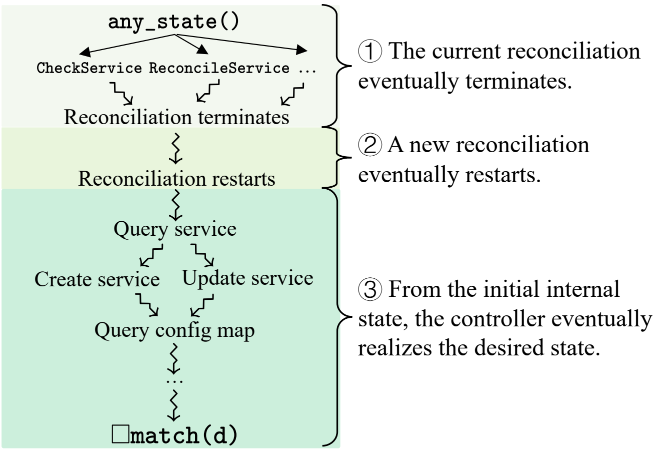

any_state()
CheckService ReconcileService ...
Reconciliation terminates
Reconciliation restarts
Query service
Create service
Update service
Query config map
...
□match(d)
① The current reconciliation eventually terminates.
② A new reconciliation eventually restarts.
③ From the initial internal state, the controller eventually realizes the desired state.

Figure 14: **Proving liveness in a stable environment.**

prove a termination property: the controller's current reconciliation (the current invocation of `reconcile()` in Figure 3) eventually terminates regardless of its current internal state. This is done by reasoning about internal states backward, e.g., `CheckService` leads to termination if all its successor states lead to termination. Second, we prove that a new reconciliation eventually starts after the previous one terminates. This holds as Anvil requeues the next invocation of `reconcile()` when the current terminates (Figure 3). Lastly, we only need to reason about the controller execution starting from its initial internal state in the new reconciliation (e.g., `CheckService` in Figure 4) to prove that the controller eventually creates and updates all the state objects to match the desired state.

To reason about the controller execution starting from its initial internal state, we need to reason about how the controller manages each state object. We observe that controllers often employ similar workflow for managing different objects, which can be leveraged to develop general lemmas to further reduce proof burden. For example, the ZooKeeper controller in Figure 4 manages its service, config map and stateful set with a similar pattern: (1) querying the object and (2) creating or updating the object depending on the query result. We develop a lemma parameterized by state objects which proves that, from the step that the controller queries the object, eventually the object always exists and matches the desired state. The lemma internally reasons about how the controller creates or updates the object to match the desired state.

5.2 Temporal Reasoning with Anvil

The proof strategy for ESR is realized by temporal reasoning. With Anvil, developers perform temporal reasoning by focusing on reasoning about state and action predicates using Anvil's TLA embedding and lemmas. We use the example in Figure 15 to demonstrate temporal reasoning with Anvil.

Developers perform temporal reasoning to prove that *all* possible executions allowed by a model satisfy a property *Prop* (`model` $\models$ *Prop*). A model is defined as the initial state

```
1  // model ≜ init ∧ □next ∧ fairness(...)
2  let model = lift(init).and(always(lift(next))
3    .and(fairness(...)));
4
5  // (1) prove model ⊨ P ⤳ Q
6  // if P holds, P or Q will hold in the next state
7  assert forall |s, s'| P(s) && next(s, s')
8  implies P(s') || Q(s') by { ... }
9  // if P holds, running A makes Q hold in the next state
10 assert forall |s, s'| P(s) && next(s, s') && A(s, s')
11 implies Q(s') by { ... }
12 // if P holds, A is enabled (A can possibly occur)
13 assert forall |s| P(s) implies enabled(A)(s) by { ... }
14 wf1(model, next, A, P, Q);
15
16 // (2) prove model ⊨ Q ⤳ R
17 ...
18 wf1(model, next, A, Q, R);
19
20 // (3) prove model ⊨ P ⤳ R
21 leads_to_transitive(model, lift(P), lift(Q), lift(R));
22
23 // (4) prove model ⊨ P ⤳ □R
24 assert forall |s, s'| R(s) && next(s, s')
25 implies R(s') by { ... }
26 leads_to_stable(model, lift(next), lift(P), lift(R));
27
28 // (5) prove model ⊨ □Inv
29 assert forall |s| init(s) implies Inv(s) by { ... }
30 assert forall |s, s'| Inv(s) && next(s, s')
31 implies Inv(s') by { ... }
32 invariant_by_induction(model, init, next, Inv);
```

Figure 15: **Temporal reasoning with Anvil.** Developers focus on reasoning about states and actions and applying TLA proof rules.

(`init`), all possible next-state actions (`next`), and fairness assumptions (line 2-3). Fairness assumptions are only used for proving liveness properties such as ESR.

Proving ESR often involves proving that if condition P holds then eventually Q holds (i.e., $P \leadsto Q$). For example, if the controller sends a request, then eventually the request is received and handled by the API server. Proving $P \leadsto Q$ is typically done by applying the WF1 rule [58]. WF1 states that "Action A makes P lead to Q" with four requirements (1) running any action in a state satisfying P makes either P or Q hold in the next state, (2) running A in a state satisfying P makes Q hold in the next state, (3) P implies that A is enabled (i.e., A can possibly occur) and (4) A has the weak fairness assumption. To apply Anvil's `wf1` lemma (line 14), developers focus on proving (1)-(3) by reasoning about P, Q, A and all other actions allowed by the model (line 7-13), and (4) is automatically proved by the definition of the model.

Proving ESR requires reasoning about a sequence of actions. For example, the controller sends a request, the API server handles the request, and the controller receives the response and continues to send the next request. To prove that the controller makes progress through multiple actions, developers apply the `leads_to_transitive` lemma (line 21) to combine multiple leads-to properties into one ($P \leadsto R$).

To reason about stability (if $P \leadsto R$, then $P \leadsto \Box R$), developers need to demonstrate that R is preserved by all possible actions (if R holds, then it will hold in the next state) and apply the `leads_to_stable` lemma (line 24-26).

Proving ESR (or other properties) often requires invariant reasoning by induction (line 29-31). For example, to prove that a state object x always exists, developers need to prove an invariant that the controller *never* deletes x. Such invariants are often required when applying `wf1` and `leads_to_stable`.

6 Case Studies

We use Anvil to build three verified Kubernetes controllers for managing different applications and services (ZooKeeper, RabbitMQ, and FluentBit). For each controller, we use a mature, widely used controller as a reference (either the official Kubernetes controller of the applications or from companies that offer related products). We verify ESR for all three controllers, and a safety property of the RabbitMQ controller.

Feature parity. We aim to implement verified controllers that are feature rich with production quality. For the ZooKeeper and RabbitMQ controllers, we implement key features offered by the reference controllers [20, 23] including scaling, version upgrading, resource allocation, pod placement, and configurations, as well as network and storage management. For the FluentBit controller, we implement *all* the features offered by the reference controller [19]. We also implement important features missing in the reference controllers. For the ZooKeeper controller, we implement a feature that the controller automatically restarts each ZooKeeper server to load the new configuration once the configuration changes. For the FluentBit controller, we implement a feature that the controller allows users to customize how a load balancer discovers FluentBit daemons. All the verified controllers can readily be deployed in real-world Kubernetes platforms and manage their respective applications.

Experience. Anvil's Controller API (Figure 2) is expressive to implement all the features of the controllers. For verification, we spent around two person-months on verifying ESR for the ZooKeeper controller, during which we developed the proof strategy (§5). We took much less time (around two person-weeks) to verify the other two controllers using the same proof strategy and similar invariants. We find Anvil's ability to formally verify a controller's implementation invaluable. We discovered deep bugs via verification. Some of them also exist in the reference controllers but were not detected by testing [44, 85].

6.1 ZooKeeper Controller

We implement and verify a full-fledged ZooKeeper controller, using the controller [23] from Pravega [22] as the reference. Figure 4 is a simplified version of our ZooKeeper controller. We discuss two challenges of verifying the controller.

Supporting non-Kubernetes APIs. We extended Anvil to support non-Kubernetes APIs to implement features like scaling. To scale a ZooKeeper cluster, the controller needs to change ZooKeeper membership by invoking ZooKeeper APIs. We implement procedures to invoke ZooKeeper APIs as callbacks invoked by `reconcile()` (Figure 3); Anvil decides whether to invoke Kubernetes APIs or ZooKeeper APIs based on the request object returned by the controller `step()`.

Invoking ZooKeeper APIs needs new specifications beyond what Anvil supplies. Hence, we write a trusted model (an abstract state machine) of the ZooKeeper APIs used by our controller and register it with the extensible compound state machine. To prove liveness, we assume weak fairness on the ZooKeeper API model: if the controller sends a request to a deployed ZooKeeper cluster, it eventually receives a response.

Reasoning about dependencies between state objects. To prove ESR, we need to reason about *dependencies* between state objects—the desired state of one object depends on the current state of another object. For example, to support reconfiguration, our controller attaches the version number of the config map to the stateful set as an annotation [7]. To ensure the ZooKeeper servers managed by the stateful set use the updated configuration, the desired state of the stateful set should contain the current version number of the config map as an annotation. To verify the correctness of reconfiguration, in ESR, `match` asserts that each state object matches the desired state description (as in Figure 10), and the annotation in the stateful set matches the current version of the config map. We prove that the config map's version eventually becomes stable and thus the annotation eventually matches the version.

Bugs precluded. We found and fixed two liveness bugs when verifying our ZooKeeper controller. The first bug occurs when the controller crashes between the steps of scaling ZooKeeper and cannot continue reconciliation after restart, similar to Figure 1. This led us to find a similar bug in the reference controller we reported in [26]. Recent work [85] applied extensive fault-injection testing on this controller but failed to find this bug, because the bug only manifests in specific timing under specific workloads (not covered by tests).

The second bug was caused by the controller trying to update immutable fields in a stateful set. Kubernetes always rejects the update, so the controller never finishes its reconciliation. Our environment model captures how Kubernetes validates each request (§4.3), which helped us find this bug.

6.2 RabbitMQ Controller

We implement and verify a full-fledged controller for RabbitMQ, a widely used message broker [24]. We use the official RabbitMQ controller as the reference [20].

Verifying safety. Besides ESR, we verify a safety property for our controller. The official RabbitMQ controller disallows scaling down a RabbitMQ cluster by reducing the stateful set's `replicas` due to data loss concerns [25]. The recommended practice is to export the data, redeploy RabbitMQ with fewer replicas, and import the data back. So, our controller prevents reducing `replicas` count. We prove a safety property stating that the replica count *never* decreases using Anvil. The safety proof is done by standard inductive proof. For example, we first prove invariants like "no request in the network reduces replicas," and conclude the replicas in the data store never decreases using the invariants.

Bugs precluded. We found a safety bug and a liveness bug via verification. The safety bug was caused by a concurrency issue involving the RabbitMQ controller and the Kubernetes garbage collector (GC). Initially, we restricted that replicas never decreases in desired state descriptions using Kubernetes' validation rule [5]. However, safety can still be violated, because the GC may not immediately remove orphan stateful sets. If the stateful set updated by the controller was created by an old (already deleted) desired state description that set a larger replicas (r_1) than the current one (r_2), the controller would in fact decrease the stateful set's replicas ($r_1 \rightarrow r_2$). We fixed the bug by enforcing the controller to wait for the GC to delete orphan stateful sets.

The liveness bug was caused by a naming rule we inherited from the reference controller. The bug causes the controller to assign the same name for service objects from *different* RabbitMQ clusters. In this case, the desired state descriptions of two RabbitMQ clusters drive the controller to change each other's service object back and forth, thus neither can reach desired states stably. We caught this bug because the oscillation behavior prevented us from proving the cluster state eventually *always* matches the desired state description in the presence of another conflicting description. We fixed the bug by changing the naming schema. The same bug also exists in the reference controller.

6.3 FluentBit Controller

We implement and verify a controller for FluentBit, a popular logging and metrics service [9]. FluentBit is deployed as a group of daemons collecting and processing data on different nodes in a cluster. We use the official FluentBit controller as the reference [19] and implement *all* its features.

Incremental verification. To evaluate the efforts of maintaining an evolving controller, we first implemented and verified a basic version of the controller that deploys FluentBit daemons, and then added new features incrementally, including version upgrading, daemon placement, reconfiguration. We repaired the proof every time when a new feature was added. We find the efforts of evolving a verified controller manageable (§7.1).

7 Evaluation

We evaluate Anvil along the dimensions of verification effort (§7.1), controller correctness (§7.2) and performance (§7.3). Our evaluation shows that it is pragmatic to implement, verify and evolve practical Kubernetes controllers with Anvil.

7.1 Verification Effort

Table 1 shows the details of each verified controller we built using Anvil. Verifying each controller takes under 3 minutes in real time on a 6-core 16 GB laptop with 11 parallel threads.

	Trusted	**Exec**	**Proof**	**Time to Verify**
	(lines of source code)			(seconds)
ZooKeeper controller §6.1				
Liveness (ESR)	94	–	7245	511
Conformance	5	–	172	9
Controller model	–	–	935	–
Controller implementation	–	1134	–	–
Trusted wrapper	514	–	–	–
Trusted ZooKeeper API	318	–	–	–
Trusted entry point	19	–	–	–
Total	950	1134	8352	520 (154)
RabbitMQ controller §6.2				
Liveness (ESR)	144	–	5211	278
Safety	22	–	358	45
Conformance	5	–	290	18
Controller model	–	–	1369	–
Controller implementation	–	1598	–	–
Trusted wrapper	358	–	–	–
Trusted entry point	19	–	–	–
Total	548	1598	7228	341 (151)
FluentBit controller §6.3				
Liveness (ESR)	115	–	7079	337
Conformance	10	–	201	10
Controller model	–	–	1115	–
Controller implementation	–	1208	–	–
Trusted wrapper	679	–	–	–
Trusted entry point	24	–	–	–
Total	828	1208	8395	347 (96)
Total (all)	2326	3940	23975	1208 (401)

Table 1: **Code sizes and verification time of the controllers verified using Anvil.** Trusted includes the (verified) theorems, trusted assumptions and unverified implementation. Time in brackets is obtained by running the verifier in parallel (11 threads on 6 cores).

87% of proof functions verify in under ten CPU seconds, and the slowest of them takes 120 CPU seconds.

Implementing and verifying each controller takes around 2.5 person-months. The proof-to-code ratio ranges from 4.5 to 7.4 across three controllers. We attribute the relatively low ratio to Anvil's reusable lemmas (§4.4) and our proof strategy (§5). For example, the ESR proof of the RabbitMQ controller uses the same set of leads-to reasoning lemmas to prove nine different state objects eventually match the desired state.

The ESR proof mainly consists of proving invariants and applying temporal proof rules. Proving invariants takes about 40% of the proof, which can potentially benefit from research on inductive invariant inference [42,48,68,69,78,79,91,93]. All our temporal logic reasoning is done by applying Anvil's temporal logic lemmas without unfolding the definition of executions and temporal logic operators.

The verified controllers have a large portion of unverified (trusted) components: 67% of the trusted code is for defining wrapper types of Kubernetes custom objects (used for describing desired states) to integrate kube-rs, and their views to enable verification (§4.1). The ZooKeeper controller also relies on the trusted ZooKeeper API: 180 lines for specifying the ZooKeeper API and 138 lines for implementing the callbacks for Anvil to invoke the ZooKeeper API during runtime.

Controller	Functional testing		Crash testing	
	# Tests	# Bugs	# Tests	# Bugs
ZooKeeper	239	1	212	0
RabbitMQ	197	0	158	0
FluentBit	557	0	484	0

Table 2: **Testing results of the three verified controllers.** The tests cover all the features of the controller under test.

Evolving controllers with Anvil. We measure the efforts to evolve the FluentBit controller with Anvil by incrementally adding features and updating its proof. We first implemented and verified a basic FluentBit controller for deploying FluentBit daemons, then added 28 new features including version upgrading, daemon placement, and various configurations. On average, implementing a feature took less than a day and 47 lines of changes, including 19 lines in the proof. Among them, implementing `metrics_port` required the most changes (403 lines in total and 211 in the proof); it added a new service that routes traffic to the metrics port of each daemon, and we proved the service eventually matches the desired state.

Effort to build Anvil. As a reference, the Anvil framework consists of 5353 lines of reusable lemmas and 7817 lines of trusted code, including the TLA embedding (85 lines), the environment model (1846 lines) and the integration with Kubernetes (5886 lines); 89% of the integration is for defining wrapper types and views of Kubernetes built-in objects (§4.1). All the lemmas are verified in under one minute.

7.2 Controller Correctness

We run extensive end-to-end functional tests on the verified controllers using Acto [44]. Acto generates different desired state descriptions to exercise controller reconciliation under various scenarios. We also run extensive crash tests to check if the verified controllers can recover from random crashes during their reconciliation. The crashes are injected using an implementation of Sieve [85] for Rust controllers.

Table 2 shows the testing results. The crash tests did not find any bug—the verified controllers correctly recovered from all the injected crashes and successfully reconciled the cluster to the desired state. The functional tests found a bug in the ZooKeeper controller (no bug found in other controllers).

The bug is caused by an incomplete specification of a trusted ZooKeeper API that did not cover ZooKeeper misconfigurations. If a misconfiguration results in partial failures (ZooKeeper is still running but cannot serve write requests [67]), the controller fails to update the membership and thus blocks the subsequent reconciliation steps. We fixed this bug by adding configuration validation in the implementation, enhancing the specification, and updating the proofs.

7.3 Controller Performance

The verified controllers have comparable performance to the reference controllers. We use Acto [44] to generate many different desired state descriptions, triggering a sequence of reconciliations. For each desired state, we measure (1) execution times for the target controllers' `reconcile()` methods (Figure 1), and (2) the time it takes for the system to be fully reconciled (e.g., after the controller issues a rolling update). The experiments are run on CloudLab Clemson c6420 machines with dual Intel Xeon Gold 6142 processors, 384GB DRAM, and a 6Gb/s HDD running Ubuntu 20.04 LTS.

Controller	Verified (Anvil)		Reference (unverified)	
	Mean (ms)	Max (ms)	Mean (ms)	Max (ms)
ZooKeeper	439	696	212	413
RabbitMQ	439	725	690	1531
FluentBit	195	303	221	464

Table 3: **Comparison of `reconcile()` execution time (in milliseconds) between the verified controllers and their references.**

Table 3 shows that the verified and reference controllers have comparable execution times. The verified ZooKeeper controller's execution time is about twice that of the reference which implements optimizations to conditionally skip state updates. None of the controllers are latency critical. On average, `reconcile()`'s execution time takes less than 1% of the overall system reconciliation time, most of which is out of the control of the controller (e.g., container restart time).

We also evaluate if the verified controllers introduce more load on the data store which is often the bottleneck for Kubernetes scalability [28, 87]. We measure the disk I/O of etcd and the verified controllers do not cause noticeably more loads—the verified FluentBit controller causes only 0.44% load increase compared to the reference; the other two verified controllers do not cause load increase.

8 Related Work

Anvil is the first effort for building formally verified cluster management controllers. We discuss related work in controller correctness, systems verification and liveness verification.

Controller correctness. Liu et al. [65] use model checking to find if controllers in a specific deployment have conflicting interactions that violate user-supplied policies at the model level (not executables). In contrast, Anvil verifies controller *implementations* against ESR, a *general* controller-correctness specification. Automated testing techniques [44, 85] have found bugs in controller implementations. Anvil precludes such bugs by verifying that the controller implementation satisfies ESR for all executions. It has also revealed bugs that were missed by these automated testing techniques (§6).

Systems verification. Despite the rich literature, most systems verification efforts so far focus on safety rather than liveness [31–34, 36, 37, 49, 50, 56, 63, 66, 75, 82, 83, 88, 90, 94]. A notable exception is IronFleet [51], which also verifies liveness of system implementations.

Anvil differs from IronFleet in the objective and proof technique. Regarding objective, IronFleet verifies a Paxos-based replicated state machine and a sharded key-value store, with system-specific specification (e.g., "if the network is fair then the reliable-transmission component eventually delivers each message"). Differently, Anvil formalizes ESR as a general specification that captures the essence of state reconciliation and verifies multiple controllers against ESR. Anvil shares IronFleet's methodology of using TLA embedding on first-order logic. Different from many IronFleet's liveness proof statements that interact directly with instantiated executions by indexing (Figure 16), Anvil abstracts away executions to let developers model components, at the level of state and action, and complete liveness proofs exclusively with temporal operators (Figure 15).

```
lemma Lemma_PacketSentEventuallyReceivedAndNotDiscarded
  (b:Behavior<LSHT_State>, send_step:int, ...)
  returns (received_step:int, ...)
  requires 0 <= send_step;
  requires SendSingleValid(b[send_step], ...);
  requires ... // other preconditions are omitted
  ensures send_step <= received_step;
  ensures b[received_step].hosts[dst_idx].host.
    receivedPacket == Some(Packet(msg, ...));
{ ... } // proof body is omitted
```

Figure 16: **A representative liveness lemma example from IronFleet (written in Dafny) [10].** The lemma counts steps in one instantiated execution (`Behavior`) to prove that if the packet is sent at `b[send_step]`, it will be received at `b[received_step]`. This lemma, if written in Anvil, will have a postcondition in the form of `model.entails(sent.leads_to(received))` without taking or returning any execution instances or indices.

Liveness verification. Ivy [72, 78] incorporates a technique for proving liveness of distributed protocols using first-order logic [76,77]. Compared to Anvil, Ivy obtains a higher degree of proof automation at the expense of a more restricted modeling logic; we are exploring the potential to leverage some of Ivy's techniques in Anvil. LVR [92] proves liveness of distributed protocols by automatically synthesizing ranking functions with limited manual guidance. LVR is complementary to Anvil and might be able to synthesize ESR proofs for controller implementations. The Alloy analyzer has recently been extended to support linear temporal logic [3, 29, 70], which enables modeling liveness properties of protocols and system abstractions; but only finite instances can be checked and the analyzed abstractions are not formally linked to executable code. More broadly, the rich literature on liveness verification includes program termination [38] and liveness of concurrent programs [27, 41, 43]. These techniques target other systems and their liveness specifications, whereas Anvil's contribution specifically targets controller correctness and connects liveness proofs to an executable implementation.

9 Discussion and Future Work

The correctness of controllers verified by Anvil is not absolute. Anvil relies on trusted components, including the model of the environment, the shim layer, trusted external APIs, and the verifier, compiler, and OS. We indeed found a bug caused by an incomplete trusted assumption (§7.2). We believe that the bug does not diminish the value of Anvil. Anvil formally verifies reconciliation – the core of a controller – and reduces the code one needs to look for bugs in to the trusted assumptions.

Note that ESR does not preclude all possible controller bugs. For example, ESR may not rule out all potential safety violations. Unlike ESR as a *general* correctness specification, safety properties are often controller-specific; e.g., the safety property we verified in §6.2 that the replicas number never decreases is specific to the RabbitMQ controller.

We choose to focus on verifying ESR because ESR is a general, reusable property that precludes a broad range of bugs, and it is straightforward for developers to specify ESR. Some bugs precluded by ESR may be precluded by some safety properties as well, but these safety properties may be more difficult for developers to specify. For example, the bug in Figure 1 could be precluded by a safety property saying "irrecoverable intermediate states never happen." However, specifying such safety properties requires knowledge of the nature of the bugs (e.g., what kind of intermediate states the controller cannot recover from?) [55]. In contrast, specifying ESR only requires knowledge of desired states.

We expect verified controllers to be deployed on real-world Kubernetes platforms, running alongside unverified controllers. If the unverified controllers are custom controllers not modeled in Anvil (§4.3.1), Anvil cannot reason about their interactions with verified controllers, and hence cannot rule out bugs caused by conflicting interactions.

In future work, we aim to gradually replace existing (unverified) controllers with verified controllers using Anvil, including both custom and built-in ones. We plan to extend Anvil to admit multiple verified controllers and verify the interactions among them in a modular way. We also plan to ensure the quality of the trusted model of the environment, the shim layer, and external APIs using lightweight formal methods.

10 Concluding Remarks

This paper presents Anvil, a framework for developing and verifying cluster-management controllers, and ESR, a general specification for controller correctness. We have implemented and verified three Kubernetes controllers using Anvil. Our work shows that it is not only feasible but also pragmatic to implement, verify, and maintain practical Kubernetes controllers. We hope that Anvil and ESR lead to a practical path towards provably correct cloud infrastructures.

Acknowledgement

We thank the anonymous reviewers for their insightful comments on the paper. We thank Jay Lorch, Owolabi Legunsen, Ramnatthan Alagappan, and Yongle Zhang for their valuable feedback and discussions that helped shape this work. This work was funded in part by NSF CNS-2145295 and a VMware Research Gift.

References

[1] etcd. https://etcd.io/.

[2] Flink controller state machine. https://github.com/lyft/flinkk8soperator/blob/master/docs/state_machine.md, 2020.

[3] Alloy 6. https://alloytools.org/alloy6.html, 2021.

[4] Controllers and Reconciliation. https://cluster-api.sigs.k8s.io/developer/providers/implementers-guide/controllers_and_reconciliation.html, 2021.

[5] CustomResourceDefinition Validation Rules. https://kubernetes.io/blog/2022/09/23/crd-validation-rules-beta/, 2022.

[6] Spark controller state machine. https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/pkg/controller/sparkapplication/controller.go#L485-L520, 2022.

[7] Annotations. https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/, 2023.

[8] Controllers. https://kubernetes.io/docs/concepts/architecture/controller/, 2023.

[9] FluentBit. https://fluentbit.io/, 2023.

[10] Ironfleet liveness lemma. https://github.com/microsoft/Ironclad/blob/2fe4dcdc323b92e93f759cc3e373521366b7f691/ironfleet/src/Dafny/Distributed/Protocol/LiveSHT/LivenessProof/LivenessProof.i.dfy#L31, 2023.

[11] kube-rs/kube: Rust Kubernetes client and controller runtime. https://github.com/kube-rs/kube, 2023.

[12] Kubernetes. https://kubernetes.io/, 2023.

[13] Kubernetes API Concepts: Updates to existing resources. https://kubernetes.io/docs/reference/using-api/api-concepts/#patch-and-apply, 2023.

[14] Kubernetes ConfigMaps. https://kubernetes.io/docs/concepts/configuration/configmap/, 2023.

[15] Kubernetes DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, 2023.

[16] Kubernetes Garbage Collection. https://kubernetes.io/docs/concepts/architecture/garbage-collection/, 2023.

[17] Kubernetes Service. https://kubernetes.io/docs/concepts/services-networking/service/, 2023.

[18] Kubernetes StatefulSet. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, 2023.

[19] Official FluentBit controller. https://github.com/fluent/fluent-operator, 2023.

[20] Official RabbitMQ controller. https://github.com/rabbitmq/cluster-operator, 2023.

[21] Owners and Dependents. https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/, 2023.

[22] Pravega. https://cncf.pravega.io/, 2023.

[23] Pravega ZooKeeper controller. https://github.com/pravega/zookeeper-operator, 2023.

[24] RabbitMQ. https://www.rabbitmq.com/, 2023.

[25] RabbitMQ: Scale down. https://github.com/rabbitmq/cluster-operator/issues/223, 2023.

[26] Stateful set never gets updated because zk node is missing. https://github.com/pravega/zookeeper-operator/issues/569, 2023.

[27] Baumann, P., Majumdar, R., Thinniyam, R. S., and Zetzsche, G. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'21)* (Jan. 2021).

[28] Berner, C. Scaling kubernetes to 2,500 nodes. https://openai.com/research/scaling-kubernetes-to-2500-nodes, Jan. 2023.

[29] Brunel, J., Chemouil, D., Cunha, A., and Macedo, N. The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (Sept. 2018).

[30] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[31] Chajed, T., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[32] Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M. F., and Zeldovich, N. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[33] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F., and Zeldovich, N. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[34] Chang, Y.-S., Jung, R., Sharma, U., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[35] Chekrygin, I. Keep the Space Shuttle Flying: Writing Robust Operators. https://youtu.be/uf97lOApOv8?t=1457, May 2019.

[36] Chen, H., Chajed, T., Konradi, A., Wang, S., Ileri, A., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[37] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[38] Cook, B., Podelski, A., and Rybalchenko, A. Proving Program Termination. *Communications of the ACM 54*, 5 (May 2011), 88–98.

[39] De Moura, L., and Bjørner, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)* (Mar. 2008).

[40] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the Presence of Partial Synchrony. *Journal of the ACM 35*, 2 (Apr. 1988), 288–323.

[41] Farzan, A., Kincaid, Z., and Podelski, A. Proving Liveness of Parameterized Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'16)* (July 2016).

[42] Goel, A., and Sakallah, K. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM'21)* (May 2021).

[43] Gotsman, A., Cook, B., Parkinson, M., and Vafeiadis, V. Proving That Non-Blocking Algorithms Don't Block. In *Proceedings of the 36th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'09)* (Jan. 2009).

[44] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[45] Guilloux, S. Writing a Kubernetes Operator: the Hard Parts. https://youtu.be/wMqzAOp15wo?t=411, Nov. 2019.

[46] Haase, S. How an Operator Becomes the Hero of the Edge. In *OperatorCon* (May 2019).

[47] Hall, C. AWS, Google, Microsoft, Red Hat's New Registry to Act as Clearing House for Kubernetes Operators. https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, Mar. 2019.

[48] Hance, T., Heule, M., Martins, R., and Parno, B. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (Apr. 2021).

[49] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., and Parno, B. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[50] Hance, T., Zhou, Y., Lattuada, A., Achermann, R., Conway, A., Stutsman, R., Zellweger, G., Hawblitzel, C., Howell, J., and Parno, B. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[51] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[52] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[53] Hockin, T. Kubernetes: Edge vs. Level Triggered Logic. https://speakerdeck.com/thockin/edge-vs-level-triggered-logic, June 2017.

[54] Howard, J. Building Better Controllers. https://www.youtube.com/watch?v=GKPBQDJ2Hjk&t=160s, Nov. 2023.

[55] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)* (Apr. 2007).

[56] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)* (Oct. 2009).

[57] Lagresle, M. Moving to Kubernetes: the Bad and the Ugly. https://youtu.be/MoIdU0J0f0E?t=263, June 2019.

[58] Lamport, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[59] Lamport, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[60] Lander, R. Kubernetes Operators: Should You Use Them? https://tanzu.vmware.com/developer/blog/kubernetes-operators-should-you-use-them/, July 2021.

[61] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying Rust Programs Using Linear Ghost Types. In *Proceedings of 2023 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)* (Apr. 2023).

[62] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)* (Oct. 2010).

[63] Leroy, X. Formal Verification of a Realistic Compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[64] Liu, B., Kheradmand, A., Caesar, M., and Godfrey, P. B. Towards Verified Self-Driving Infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)* (Nov. 2020).

[65] Liu, B., Lim, G., Beckett, R., and Godfrey, P. B. Kivi: Verification for Cluster Management. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC'24)* (July 2024).

[66] Lorch, J. R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J. R., and Zhao, X. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)* (June 2020).

[67] Lou, C., Huang, P., and Smith, S. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[68] Ma, H., Ahmad, H., Goel, A., Goldweber, E., Jeannin, J.-B., Kapritsos, M., and Kasikci, B. Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)* (July 2022).

[69] Ma, H., Goel, A., Jeannin, J.-B., Kapritsos, M., Kasikci, B., and Sakallah, K. A. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[70] Macedo, N., Brunel, J., Chemouil, D., and Cunha, A. Pardinus: A temporal relational model finder. *J. Autom. Reason. 66*, 4 (2022), 861–904.

[71] Madhu, C. Preventing Controller Sprawl From Taking Down Your Cluster. https://youtu.be/fu5GXo7jmV0?t=732, Oct. 2022.

[72] McMillan, K. L., and Padon, O. Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20)* (July 2020).

[73] Melissaris, T., Nabar, K., Radut, R., Rehmtulla, S., Shi, A., Chandrashekar, S., and Papapanagiotou, I. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)* (Nov. 2022).

[74] Motwani, S., and Maheshwari, A. Deep Dive Into Writing a Kubernetes Operator: Let's Avoid Data Loss and Down Times. https://www.youtube.com/watch?v=2NjMHLACvc0&t=737s, Nov. 2023.

[75] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[76] Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., and Shoham, S. Reducing Liveness to Safety in First-Order Logic. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[77] Padon, O., Hoenicke, J., McMillan, K. L., Podelski, A., Sagiv, M., and Shoham, S. Temporal Prophecy for Proving Temporal Properties of Infinite-State Systems. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD'18)* (Oct. 2018).

[78] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (June 2016).

[79] Padon, O., Wilcox, J. R., Koenig, J. R., McMillan, K. L., and Aiken, A. Induction Duality: Primal-Dual Search for Invariants. In *Proceedings of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'22)* (Jan. 2022).

[80] Pnueli, A. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Oct. 1977).

[81] Ratis, P. Lessons Learned using the Operator Pattern to build a Kubernetes Platform. In *USENIX SREcon* (Oct. 2021).

[82] Sharma, U., Jung, R., Tassarotti, J., Kaashoek, F., and Zeldovich, N. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[83] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[84] Sosa, C., and Bhatia, P. Application management made easier with Kubernetes Operators on GCP Marketplace. https://cloud.google.com/blog/products/containers-kubernetes/application-management-made-easier-with-kubernete-operators-on-gcp-marketplace, May 2019.

[85] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[86] Sun, X., Suresh, L., Ganesan, A., Alagappan, R., Gasch, M., Tang, L., and Xu, T. Reasoning about modern datacenter infrastructures using partial histories. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)* (May 2021).

[87] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the*

14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20) (Nov. 2020).

[88] TAUBE, M., LOSA, G., MCMILLAN, K. L., PADON, O., SAGIV, M., SHOHAM, S., WILCOX, J. R., AND WOOS, D. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)* (June 2018).

[89] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[90] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015).

[91] YAO, J., TAO, R., GU, R., AND NIEH, J. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[92] YAO, J., TAO, R., GU, R., AND NIEH, J. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'24)* (Jan. 2024).

[93] YAO, J., TAO, R., GU, R., NIEH, J., JANA, S., AND RYAN, G. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[94] ZOU, M., DING, H., DU, D., FU, M., GU, R., AND CHEN, H. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

DSig: Breaking the Barrier of Signatures in Data Centers

Marcos K. Aguilera[1], Clément Burgelin[2], Rachid Guerraoui[2], Antoine Murat[2], Athanasios Xygkis[3], and Igor Zablotchi[4]

[1]*VMware Research Group*
[2]*École Polytechnique Fédérale de Lausanne (EPFL)*
[3]*Oracle Labs*
[4]*Mysten Labs*

Abstract

Data centers increasingly host mutually distrustful users on shared infrastructure. A powerful tool to safeguard such users are digital signatures. Digital signatures have revolutionized Internet-scale applications, but current signatures are too slow for the growing genre of microsecond-scale systems in modern data centers. We propose DSig, the first digital signature system to achieve single-digit microsecond latency to sign, transmit, and verify signatures in data center systems. DSig is based on the observation that, in many data center applications, the signer of a message knows most of the time who will verify its signature. We introduce a new hybrid signature scheme that combines cheap single-use hash-based signatures verified in the foreground with traditional signatures pre-verified in the background. Compared to prior state-of-the-art signatures, DSig reduces signing time from 18.9 to 0.7 µs and verification time from 35.6 to 5.1 µs, while keeping signature transmission time below 2.5 µs. Moreover, DSig achieves 2.5× higher signing throughput and 6.9× higher verification throughput than the state of the art. We use DSig to (a) bring auditability to two key-value stores (HERD and Redis) and a financial trading system (based on Liquibook) for 86% lower added latency than the state of the art, and (b) replace signatures in BFT broadcast and BFT replication, reducing their latency by 73% and 69%, respectively.

1 Introduction

Digital signatures are used in many distributed protocols that have revolutionized the Internet through many use cases, such as enabling digital certificates [83], bootstrapping authentication protocols [30, 89], securing and auditing transactions [22, 71], tolerating Byzantine failures [4, 6, 18], and verifying software authenticity [28, 57]. Signatures are irrefutable proofs that someone produced a message, and these proofs can be verified by third parties. This property distinguishes signatures from message authentication codes (MACs) and authenticated symmetric encryption (e.g., SSL/TLS) [54]. Today's signatures are however too expensive for the growing genre of microsecond-scale systems in data centers. Even the fastest signature scheme, EdDSA [17, 50], accounts for 79–95.6% of the latency of applications such as auditable key-value stores, BFT broadcast, and BFT replication (Figure 1).

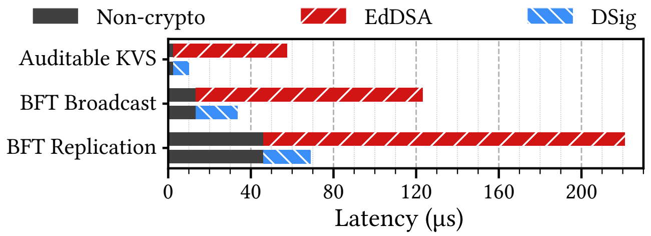


Figure 1: Median latency breakdown of an auditable key-value store (based on HERD [51], §6), a BFT broadcast primitive (CTB [3], §6), and a BFT replication system (uBFT [3], §6) when processing small requests using either EdDSA [50] (state of the art) or DSig. DSig reduces the cryptographic overhead by 86%, 82%, and 87%, respectively, and reduces the overall latency by 83%, 73%, and 69%, respectively.

We propose DSig, the first digital signature system to achieve single-digit microsecond performance for data centers. A key insight underlying the design of DSig is that, in many data center applications, signatures are issued and verified by parties known a priori in the common case, so signers and verifiers can exchange useful information beforehand and do part of the computation before knowing the messages to be signed, thereby reducing the latency of subsequent signature generation and verification. We use this observation to introduce a new hybrid online-offline signature system [37]. Hybrid schemes combine a traditional signature scheme that is slow but can sign many messages, with a hash-based signature scheme (HBSS) that is fast but can sign only one or a few messages. The traditional signature is used to validate a batch of HBSS key pairs, each of which signs one or a few messages. Hybrid signatures have been studied extensively in theory, but practical work has focused only on improving throughput for low-compute devices with limited bandwidth,

Table 1: Comparison of EdDSA and DSig in terms of latency to sign, transmit (tx) and verify; per-core sign and verify throughput; signature size; and background network traffic per signature with a single verifier.

	Latency (µs)			**Tput (Kops)**		**Sig size**	**Bg Net**
	Sign	Tx	Verify	Sign	Verify	**(B)**	**(B/Sig)**
EdDSA	18.9	1.1	35.6	53	28	64	0
DSig	0.7	2.0	5.1	131	193	1,584	33

on low-security signatures, or on tiny messages (§9).

Using hybrid signatures to achieve low latency and high throughput in data centers poses a number of challenges. First, the traditional part of hybrid signatures is compute-heavy and can impact latency. Second, hybrid signatures involve frequent key pair generation, which can exhaust compute resources and impact throughput. Third, the actual performance of hybrid signatures deviates from their theoretical models, as real performance requires careful consideration of microarchitectural effects (e.g., caching, prefetching) rather than simply the amount of computation. Fourth, to perform well, hybrid signatures need to be configured with an appropriate HBSS whose thousands of parameter combinations provide complex trade-offs between number of hash computations, signature size, and frequency of key pair generation; most parameter choices exceed our performance goals, the available network bandwidth, the computational resources, or all of the above.

We address these challenges as follows. First, we use hints about who will verify a message in the common case to preprocess the compute-heavy traditional signatures. Second, we use traditional signatures to sign and verify batches of HBSS public keys, thus amortizing the cost of their authentication, while hiding the latency introduced by batching from the critical path. Third, we study the real performance of HBSSs to determine the best schemes to use, and we discover non-intuitive cases where fewer hash computations actually harm performance. Fourth, we identify two promising HBSSs to use in DSig, W-OTS$^+$ [46] and HORS [84], and we explore their parameters in depth to understand how they affect latency, throughput, and resource usage; we give a recommended configuration of DSig that strikes a good trade-off.

We integrate DSig with five applications: two key-value stores (HERD [51] and Redis [87]), a financial trading system (based on Liquibook [73]), a BFT broadcast primitive (CTB [3]), and a BFT replication system (uBFT [3]). We use DSig to provide auditability through a signed security log in HERD, Redis, and Liquibook; and to replace the signatures used in CTB and uBFT to thwart Byzantine behavior.

We evaluate DSig and its applications. We find that DSig can sign, transmit, and verify a signature in 7.7 µs total, which is 7.2× faster than EdDSA [50], the fastest traditional signature scheme [17] (Table 1). DSig achieves 2.5× and 6.9× higher throughput than EdDSA for signing and verifying. DSig benefits applications significantly. In HERD, Redis, and Liquibook, DSig brings auditability with an added latency of less than 8 µs per operation, a reduction of 86% in overhead compared to EdDSA. In CTB, DSig reduces the broadcast latency by 73%, from 123 µs to 34 µs. In uBFT, DSig reduces the replication latency by 69%, from 221 µs to 69 µs.

The price for using DSig is as follows. First, to get the best performance, DSig needs a priori knowledge of where and when signatures are verified (DSig still works without such knowledge, but it is slower). Second, DSig requires extra bandwidth and space to transmit and store its larger ≈1.5 KiB signatures. This is a small cost in data-center systems, which have low-latency high-bandwidth networks and abundant storage, but DSig could be ill-suited for other settings, such as some wide-area systems.

In summary, our contributions are the following:

- We propose DSig, a new digital signature system targeted at microsecond-scale applications in data centers. DSig combines hash-based signatures, traditional signatures, and novel techniques to reduce latency in the critical path while achieving high throughput.
- We analyze and evaluate DSig's large parameter space for low latency at high throughput, and identify a configuration that best fits most scenarios.
- We implement DSig and integrate it into several applications: two key-value stores, a financial trading system, BFT broadcast, and a BFT replication system.
- We evaluate DSig and its applications. DSig significantly improves signature performance compared to EdDSA, the state of the art. These enhancements directly benefit the applications by providing better end-to-end latency and throughput, and by bringing auditability to the microsecond scale.

DSig is open source, available at `https://github.com/LPD-EPFL/dsig`.

2 Setting and Goals

Setting. We target microsecond-scale applications [2, 3, 11, 19, 31, 41, 76, 80–82] with a few tens of servers within the same data center—a scale that addresses the computing needs of many enterprises. These systems have a network with low latency (≈1 µs) and high bandwidth (100s of Gbps or even Tbps [74]).

Goals. Our goal is to achieve faster digital signatures to broaden their usability. We do not seek general-purpose signatures for all settings (wide area networks, mobile networks, embedded systems), but rather seek schemes that provide the right trade-offs in modern data centers. We seek signatures that provide the industry-standard level of security (128 bits).

Digital signatures are important because they are *transferable*: if Alice signs a message to Bob, Bob can prove to Carol that Alice indeed signed it (§3). This property makes signatures more powerful than mechanisms such as SSL/TLS or MACs, which provide only symmetric authenticated channels between Alice and Bob [54], which do not suffice for the applications we consider (§6). Signatures help tackle distrustful parties in distributed protocols for a wide variety of use cases: securing transactions, enabling digital certificates, bootstrapping authentication of users and services [30, 89], verifying software authenticity [28, 57], providing integrity of audit logs [22, 71], tolerating Byzantine failures [4, 6, 18], etc. Many of these use cases apply to microsecond-scale applications, as such systems increasingly bring together mutually distrustful users on shared infrastructure [48, 88]. For example, microservices can benefit from Byzantine fault tolerance [3]; signed transactions can provide auditability in high-frequency trading systems; and signed logs can provide a legal trail in high-stakes settings.

Threat model. Our threat model is standard for digital signatures [54]. Malicious entities can intercept, store, inject, delay, and alter messages. We assume the security of standard cryptography building blocks: traditional digital signature schemes (Ed25519 [50]), hash-based signature schemes (W-OTS$^+$ [46] and HORS [84]), and cryptographic hashes (SHA256 [77], Haraka (v2) [55], and BLAKE3 [75]).

3 Background

3.1 Digital Signature Schemes

A digital signature scheme (DSS) has a key pair consisting of a public key *PK* and a secret key *SK*. A signer *s* uses *SK* to sign a message *m*, producing a signature σ for *m*. The signature σ allows a party who knows *PK* (and knows that *PK* belongs to *s*) to verify that *m* was signed by *s*.

DSSs provide *authenticity*, *integrity*, *public verifiability* and *non-repudiation* of messages [54]. Authenticity means that a party with a message and its signature can verify the identity of the message's signer. Integrity means that the party can verify that the message matches the message that was signed. Public verifiability means that only *m*, σ, and *PK* are needed to verify the authenticity and integrity of *m*. Signatures are transferable: a party can use σ and *m* to convince anyone who knows *PK* that *m* is authentic (and typically *PK* is published, so everyone knows *PK*). This property differentiates digital signatures from other mechanisms, such as message authentication codes (MACs), vectors of MACs, authenticated channels (e.g., SSL/TLS), and symmetric encryption (e.g., AES). Non-repudiation means that *s* cannot deny the signing of *m* once its signature σ is known. Non-repudiation implies that signatures are non-forgeable: without knowing *SK*, it is computationally infeasible to produce a signature σ which passes verification with *PK*.

3.2 The Cryptographic Barrier

After DSSs were proposed [33], many schemes followed: RSA [85], ECDSA [49], EdDSA [50], etc. These schemes rely on the hardness of certain problems (factoring, discrete logarithms) under sophisticated arithmetic (e.g., modular on elliptic curves). They seek to provide strong security and small time to sign and verify. For example, the state-of-the-art 128-bit-secure EdDSA takes 19 µs to sign and 36 µs to verify a small message on modern CPUs (Table 1).

State-of-the-art DSSs are too slow for modern data centers: even the fastest schemes are an order of magnitude slower than network latencies [17] due to the use of sophisticated arithmetic, which consumes CPU and cannot be parallelized. This slowness makes traditional DSSs prohibitive for distributed protocols, microservices, and applications that run at the microsecond scale, which need to check the signatures of messages before acting upon them. For example, signature-based BFT protocols must check signatures before taking safety-critical steps such as computing a quorum intersection, voting, vouching for a message, deciding on a majority value, etc; similarly, auditable applications must check signatures before executing requests to provide auditability (§6).

Signing or verifying messages in batches can improve the throughput of DSSs, but batching further increases latency and is thus ill-suited for latency-critical applications.

3.3 Hash-Based and Hybrid Signatures

Hash-based signature schemes (HBSSs) were proposed by Lamport [59]. They are DSSs that avoid advanced arithmetic by using only cryptographic hashes. Hashes are advantageous because they can be computed quickly: modern algorithms (e.g., Haraka [55] and BLAKE3 [75]) can hash a small message in less than 100 ns on modern CPUs. In some HBSSs (e.g., HORS [84], W-OTS$^+$ [46]), signature generation and verification execute at the microsecond scale, as they require few hash computations.

To explain HBSSs, we overview the HORS scheme (Figure 2), which, whilst simple, illustrates the key ideas of HBSSs. The secret key *SK* for signing is an array of *t* random secrets (*t* is a parameter), while the public key *PK* for verify-

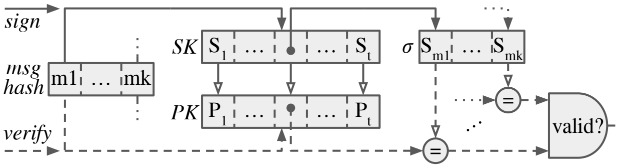


Figure 2: The HORS hash-based signature scheme. Solid lines convey the path taken to sign a message, while dashed lines convey the path to verify a signature. Hollow arrows indicate cryptographic hashes.

ing is the concatenation of the hash of each secret in SK. To sign a message, the signer hashes it with a salt into a string m, splits m into k substrings (k is a parameter), treats each substring as an index into SK, and concatenates the indexed secrets to obtain the signature σ. A verifier hashes the message with the salt and uses the substrings to index into PK. Then, the verifier hashes each secret in σ and checks that they match the indexed elements of PK. This scheme is secure because it is hard to (1) find messages that index the same secrets or (2) reveal secrets without being the signer. Other fast HBSSs, such as W-OTS$^+$ [46], are similar to HORS in that they sign by revealing a subset of the private key; as a result, they are limited to signing one or a few messages.

To overcome this limitation and sign an unlimited number of messages, hybrid signature schemes [37] combine HBSSs with traditional schemes. To sign a message m, a hybrid scheme concatenates an HBSS signature on m with the HBSS public key signed using a traditional signature. To verify a signature, the scheme verifies the HBSS signature of m and the traditional signature of the HBSS public key.

3.4 Challenges

Hybrid signatures were studied extensively in theory, but their application focused either on improving throughput in low-compute low-bandwidth devices, or on low-security signatures, or on tiny messages with only a few bits (§9). To use them in a high-performance data center setting, we must tackle several challenges.

Efficient signature verification. To verify a hybrid signature, we must check both its HBSS signature and its traditional signature. Traditional signatures, however, have high verification latency. We need new mechanisms to avoid the traditional signature verification in the critical path.

Frequent key generation. Because an HBSS key pair can be used only once or a few times, hybrid schemes need to frequently generate and sign new HBSS key pairs. This can become a bottleneck as it impairs signature throughput and, ultimately, its latency. We need new techniques to improve throughput while minimizing latency on the critical path.

Practical performance. We evaluate the performance of hybrid schemes and find that it does not match their theoretical analysis. The latter is based on simple metrics, namely the size of signatures and the number of hash calculations in the critical path. However, due to microarchitectural effects (e.g., CPU cache size, prefetching), optimal configurations in theory perform suboptimally in practice, and optimizations that target solely the theoretical metrics sometimes do not work.

Large parameter space. Hybrid signature schemes have many configuration options: the choice of the traditional scheme, choice of the HBSS, and their respective parameters. As a result, we are faced with thousands of options that provide different trade-offs in network bandwidth, computational resources, throughput, and latency characteristics.

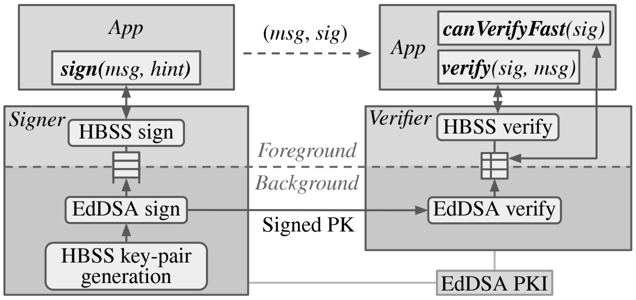


Figure 3: Architecture of DSig.

4 Design of DSig

We present the architecture of DSig (short for "Data center Signatures"), highlighting its extended interface and computing planes (§4.1). We then describe DSig's hybrid signature scheme (§4.2), its security (§4.3), and throughput optimizations (§4.4).

4.1 Architecture

Figure 3 depicts the architecture of DSig. Each process has a public-private key pair of a traditional signature scheme, where the public key is made available to other parties via a public key infrastructure (PKI). For the traditional signature scheme, we choose EdDSA [50] because it is the fastest such scheme [17]. The PKI can be as simple as an administrator pre-installing the keys, or it can be a full-fledged system.

DSig augments the interface of digital signatures (*sign* and *verify* functions) in two ways. First, *sign* takes a hint with the set of processes that will likely verify the signature. The hint is optional: if omitted, it defaults to all known processes. The hint does not restrict who can verify a signature—parties not indicated in the hint can still verify the signature, albeit at a lower performance. Second, a new *canVerifyFast* function returns whether verification of a given signature will be fast. This function can mitigate denial-of-service attacks by letting applications prioritize the handling of fast signatures (§6).

Internally, DSig has two planes: *foreground* and *background*. The foreground plane provides the user library with a synchronous API to sign and verify messages, while the background plane asynchronously pre-generates and propagates new HBSS keys to be used by the foreground plane. DSig's general design can be used with a wide range of HBSSs (e.g., Lamport's [59], HORS [84], W-OTS [34], W-OTS$^+$ [46]). We provide a specific recommendation based on an extensive performance study (§5).

Foreground plane. To sign a message, the signer uses a fresh HBSS key pair and returns to the application a DSig signature, which includes the resulting HBSS signature and the

Algorithm 1: Signers' Pseudocode

```
1  # Signer's setup
2  verifier_groups = { ... } # Provided
3  for group in verifier_groups:
4    signed_keypairs[group] = Queue()

6  # Signer's background plane
7  whenever $\exists$ group | signed_keypairs[group].size < $S$:
8    sk, pk = hbss.generate_keypair()
9    pk$_\sigma$ = {pk: pk, sig: eddsa.sign(pk)}
10   multicast <HBSS_PUBKEY, pk$_\sigma$> to group
11   signed_keypairs[group].push((sk, pk$_\sigma$))

13 # Signer's foreground plane
14 def sign(msg, hint):
15   group = smallest group containing hint
16   sk, pk$_\sigma$ = signed_keypairs[group].pop()
17   hbss_sig = hbss.sign(msg, sk)
18   return <SIG, hbss_sig, pk$_\sigma$>
```

Algorithm 2: Verifiers' Pseudocode

```
19 # Verifier's setup
20 verified_pks = Cache()

22 # Verifier's background plane
23 upon deliver <HBSS_PUBKEY, pk$_\sigma$> from process p:
24   if eddsa.verify(pk$_\sigma$, eddsa_pk_of(p)):
25     verified_pks.add((pk$_\sigma$, p))

27 # Verifier's foreground plane
28 def verify(msg, <SIG, hbss_sig, pk$_\sigma$>, p):
29   if (pk$_\sigma$, p) not in verified_pks:
30     if not eddsa.verify(pk$_\sigma$, eddsa_pk_of(p)): # Slow
31       return false
32   return hbss.verify(msg, hbss_sig, pk$_\sigma$.pk)

34 def canVerifyFast(<SIG, _, pk$_\sigma$>, p):
35   return (pk$_\sigma$, p) in verified_pks
```

corresponding HBSS public key signed with EdDSA. Signing with EdDSA is slow, so the HBSS public key is pre-signed in the background plane. On the other side, the verifier checks the authenticity of the message using the HBSS signature and the included HBSS public key. The authenticity of the HBSS public key is checked by the background plane.

Background plane. The signer generates HBSS key pairs and EdDSA-signs them before forwarding them to the foreground plane. It also sends the EdDSA-signed HBSS public keys to the background plane of the likely verifiers. The latter verifies the authenticity of the HBSS public keys.

The background plane hides the latency of two slow steps: (1) HBSS key pair generation, and (2) EdDSA-signing and EdDSA-verifying the HBSS public keys.

Note that DSig preserves the transferability of signatures irrespective of the background plane. Because DSig hybrid signatures are self-standing (as they include the EdDSA-signed HBSS public key), the only requirement for signature verification is knowledge of the signer's EdDSA public key. The background plane merely boosts performance when a hint is correct, by letting a verifier pre-check an HBSS public key before it receives a signature that includes it.

4.2 Signing and Verifying in DSig

The logic of a DSig signer is shown in Algorithm 1. Each signer is configured with a list of *verifier groups*—groups of processes that are likely to verify the same signatures on their critical path (line 2). This list has a default group that contains all the processes in the system, but is otherwise application-dependent. In the applications we examined (§6), the list was small and obvious (e.g., individual groups containing one process each). Each verifier group is associated with a key-pair queue (lines 3–4).

In the background plane, whenever a group's queue size is below a threshold S (line 7), the signer generates a new HBSS key pair (line 8), and signs the public key using EdDSA (line 9). Empirically, we found that S=512 works well while consuming only 3 MiB of memory per group. Then, the signer multicasts the signed public key to the processes in the group (line 10). The signer next appends the private key with the EdDSA-signed public key to the queue for consumption in the foreground plane (line 11).

To sign a message, the signer chooses the verifier group that matches the hint; if no group matches the hint, the signer picks the smallest group containing the hint (line 15). Then, it gets a new HBSS key pair from this group's queue (line 16). Next, the signer computes an HBSS signature of the message using the private key obtained from the queue (line 17). The DSig signature comprises the HBSS signature of the message together with its EdDSA-signed HBSS public key (line 18).

The logic of a DSig verifier is shown in Algorithm 2. In the background plane, the verifier receives EdDSA-signed HBSS public keys (line 23), which it verifies (line 24) and stores in a cache (line 25). In our applications, we found that having the cache store the latest $2{\times}S{=}1024$ HBSS public keys from each signer worked well while consuming only 100 KiB of memory per signer. In the foreground plane, the verifier first consults its cache to see if it has a verified entry for the HBSS public key (line 29). If so, the verifier proceeds with checking the HBSS signature using the HBSS public key. In this code path, the verifier checks signatures as fast as the underlying HBSS verify (line 32), which is fast. Otherwise, the verifier also checks the EdDSA signature of the HBSS public key (line 30), so the verifier can operate even without the background plane. The verifier's *canVerifyFast* function (§4.1) simply checks whether a signed HBSS public key has already been verified (lines 34–35).

As with other signature schemes, DSig can support key revocation through revocation lists that applications check prior to signing or verifying messages [54].

4.3 Security

For preciseness of argument, we show the security of our recommended DSig configuration, which uses W-OTS$^+$ [46] as its underlying HBSS (§5). Specifically, we show that this configuration of DSig is Existentially Unforgeable under Chosen-Message Attacks (EUF-CMA) [39] and that it provides 128-bit security, which is safe by today's standards [7].

EUF-CMA security. We consider Chosen-Message Attacks (CMA) in which the attacker can query the target to sign arbitrary messages. More precisely, we consider *adaptive* CMA in which the attacker can query the target based on public key(s) and previously obtained signatures. We consider Existential Unforgeability (EUF), which means it should be computationally infeasible for an attacker to forge a signature on any message, except for messages that have already been signed by the target.

DSig is as secure as its parts. To forge a signature in DSig, an attacker must find a combination of message (not previously signed), W-OTS$^+$ public key, EdDSA signature, and W-OTS$^+$ signature that passes the `verify` function (Algorithm 2). We assume that EdDSA provides EUF-CMA security, as proved by Brendel *et al.* [20], and show how DSig's security reduces to the security of W-OTS$^+$:

1. The verifier's background plane caches only correctly EdDSA-signed public keys (Alg. 2 lines 23–25). From the EUF-CMA security of EdDSA, and since a correct signer EdDSA-signs only its own public keys, (Alg. 1 lines 8–9), for any correct signer s, the verifier caches only the W-OTS$^+$ public keys s generates.

2. If a public key is not cached, the verifier EdDSA-verifies it on the critical path (Alg. 2 lines 29–31). As in (1) above, for any correct signer s, this verification only succeeds for public keys s generates. Thus, for any correct signer s, `verify` cannot return *true* for public keys s does not generate.

3. Since, for any correct signer s, an attacker can only use a W-OTS$^+$ public key generated by s, forging a DSig signature reduces to forging a W-OTS$^+$ signature.

W-OTS$^+$ with Haraka and BLAKE3. Hülsing proved that W-OTS$^+$ is EUF-CMA-secure when using a hash-function family that is second-preimage resistant, undetectable, and one-way [46]. Like SPHINCS+ [16], we pick the Haraka [55] hash-function family which satisfies those properties and relies on the battle-tested AES round function. Similarly to SPHINCS+, we reduce the signed messages to 128-bit digests by hashing them salted with the W-OTS$^+$ public key and a random nonce. We do so using the well-established BLAKE3 [75] hashes. Finally, we tune W-OTS$^+$'s parameters to provide 128 bits of security when signing said 128-bit digests. More precisely, we set the size of secrets and public key elements to 144 bits, which, together with a W-OTS$^+$ depth of 4 (§5), provides a security level of 133.9 bits [46].

DSig's security level. Breaking DSig can be reduced to breaking either EdDSA, W-OTS$^+$, Haraka, or BLAKE3. The EdDSA signature scheme Ed25519 provides 128-bit security under practical attacks [14], and our configuration of W-OTS$^+$ provides 133-bit security. The security of both Haraka and BLAKE3 relies on well-studied components [7, 55] and to date, no attack has compromised their security.

4.4 Optimizing Throughput

DSig has a few throughput optimizations that do not significantly impact latency.

Speeding up key pair generation. Generating an HBSS key pair requires producing hundreds to thousands of secrets for the private key, and then hashing each secret for the public key. To produce secrets quickly, DSig collects entropy from the hardware at startup to get a truly random 256-bit seed, which DSig then salts with the key index and hashes using BLAKE3 to generate a digest with the size of the private key. To produce the public key quickly, DSig hashes the secrets using Haraka, which has a high-throughput implementation that optimizes instruction pipelining to compute multiple hashes efficiently.

Amortizing the cost of EdDSA signatures. EdDSA-signing every HBSS public key is slow and becomes a throughput bottleneck as each EdDSA sign-verify computation takes ≈55 µs [17]. DSig EdDSA-signs batches of HBSS public keys [86]. However, batching naively would increase the size of signatures, since the entire batch of HBSS public keys should be included in every DSig signature to make their EdDSA signature self-standing. DSig addresses this issue by arranging the batch of HBSS public keys into a Merkle tree [67] and EdDSA-signing its root. As a result, a DSig signature is composed of an HBSS signature, an HBSS public key, a Merkle inclusion proof, and the EdDSA signature of the Merkle root. The Merkle proof is a space-efficient way of proving that the included HBSS public key is part of the EdDSA-signed batch. As Merkle proofs require collision-resistant hashes, we use (the efficient) BLAKE3. The space efficiency of Merkle proofs comes at the computational cost of generating and verifying them. DSig moves most of this cost to the background plane of both signers and verifiers, which precompute and cache the full Merkle tree associated with a batch. Then, on the critical path, signing a key merely requires copying the subset of the tree that constitutes the Merkle proof, while verifying the Merkle proof consists of simple string comparisons. The efficiency of this scheme depends on the batch size, which we determine in §8.7.

Speeding up bulk verification. Verifying many signatures with no assistance from the background plane (e.g., when checking an audit log) requires checking the same EdDSA signatures many times. To speed up this process, EdDSA signatures verified in the foreground plane are cached. A cache entry takes only ≈33 bytes, a tiny overhead, but saves ≈36 µs of computation on our hardware (Table 3).

Reducing background bandwidth. Sending signed public keys both ahead of time and within signatures consumes significant networking bandwidth. To nearly halve the bandwidth usage, DSig batches, EdDSA-signs, and sends only BLAKE3 digests of the public keys in the background plane. This optimization requires computing the public key digest during signature verification, which adds only ≈1.3 µs of latency.

5 Choice of HBSS

DSig can be used with many HBSSs, but its performance heavily depends on the actual HBSS used and how this HBSS is configured. In this section, we explain which HBSS we choose for DSig and how we configure it.

5.1 HBSS Requirements

Our choice of HBSS is guided by the following requirements.

Security. To provide 128-bit security, DSig needs an HBSS with the same or stronger security.

Low sign and verify latency. DSig executes HBSS *sign* and *verify* operations on the critical path. These operations must have microsecond-scale latency. This latency depends on the choice of the hash function, on the number of hashes they involve, and on microarchitectural effects.

Short signatures. At the microsecond scale, signatures cannot exceed a few KiB in length, as larger signatures incur significant transmission latency: we experimentally find that when sending small messages each extra KiB takes approximately an extra microsecond on a 100 Gbps network. Furthermore, large signatures significantly increase the bandwidth consumption when applications send small messages.

Compressible public keys. Recall that DSig signatures must include an HBSS public key in order to be self-standing (§4.2). However, the combination of an HBSS signature and its public key can be in the KiB range, which is undesirable. We thus seek HBSSs for which this combination can be compressed, leading to smaller DSig signatures.

Lightweight key generation. HBSS key generation mainly involves hash computations, the number of which depends on the HBSS and can bottleneck DSig's throughput. HBSS that use few hashes for key generation are thus desirable.

5.2 Analysis

HBSSs can be grouped in two classes: HBSSs with keys that can sign only one or a few messages [34, 46, 59, 84], and HBSSs with keys that can sign many messages [9, 15, 16, 21, 47, 58]. Only the first class provides low latency (the second focuses on quantum resistance). Within that class, we focus on the fastest HBSSs: HORS [84] and W-OTS$^+$ [46].

HORS. Recall that a HORS signature reveals a subset of its private key secrets determined by the message being signed (§3.3). Verifying a signature requires hashing the revealed secrets, checking that they match the public key, and checking that all the secrets mandated by the signed message were revealed. HORS has two parameters: the number k of secrets revealed in a signature and the number of times r that a key pair can be used. The size of HORS keys is proportional to r, so picking $r \geq 2$ presents no benefits and we set $r=1$. Smaller values of k, however, lead to fewer hash computations, and thus to lower latency in theory, but they require larger HORS public keys for the same security level. Large HORS public keys require compression to fit our signature size budget. We thus devise two compression techniques, described next.

The first technique shortens the embedded HORS public key by removing the elements that can be deduced from the HORS signature (top of Figure 4). We analyze this approach in the first part of Table 2, which shows that configurations with few hashes ($k<32$) have large signatures (tens of KiB) that exceed our budget.

To use HORS signatures with small k while staying within our signature size budget, we devise another compression technique inspired by SPHINCS [15]. This technique is based on the observation that verifying a HORS signature merely requires checking that the few revealed secrets match the public key; knowledge of the full public key is unnecessary. We enable such checks using Merkle inclusion proofs: we arrange all public key elements in a Merkle forest, and EdDSA-sign its roots. Such DSig signatures replace their HORS public

Table 2: Analytical comparison of a DSig signature using either HORS or W-OTS$^+$ as its HBSS for various configurations with EdDSA batches of 128 public keys.

Conf	# Critical Hashes	Signature Size (B)	# BG Hashes	BG Traffic (B/Verifier)
Using HORS with factorized PKs				
k=8	8	8Mi	512Ki	33
k=16	16	64Ki	4Ki	33
k=32	32	8,552	512	33
k=64	64	4,456	256	33
Using HORS with merklified PKs				
k=8	8	4,712	1Mi	8Mi
k=16	16	4,968	8Ki	64Ki
k=32	32	5,480	1Ki	8Ki
k=64	64	6,504	510	4Ki
Using W-OTS$^+$				
d=2	≈68	2,808	136	33
d=4	≈102	1,584	204	33
d=8	≈161	1,188	322	33
d=16	≈263	990	525	33
d=32	≈434	864	868	33

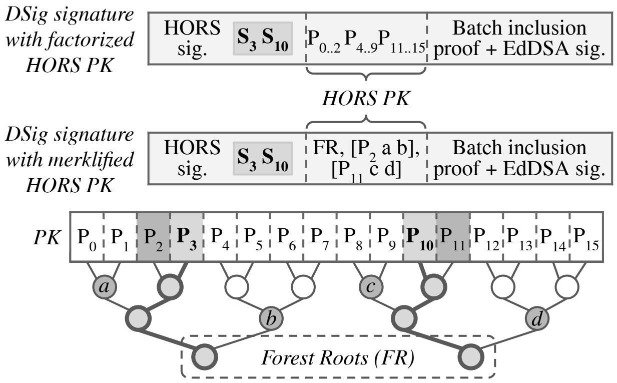


Figure 4: DSig signatures when using HORS with either factorized or merklified public keys. For illustration purposes, we depict a toy configuration of HORS with 2-secret signatures and 16-element keys.

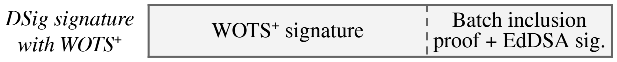


Figure 5: Layout of DSig signatures when using W-OTS$^+$.

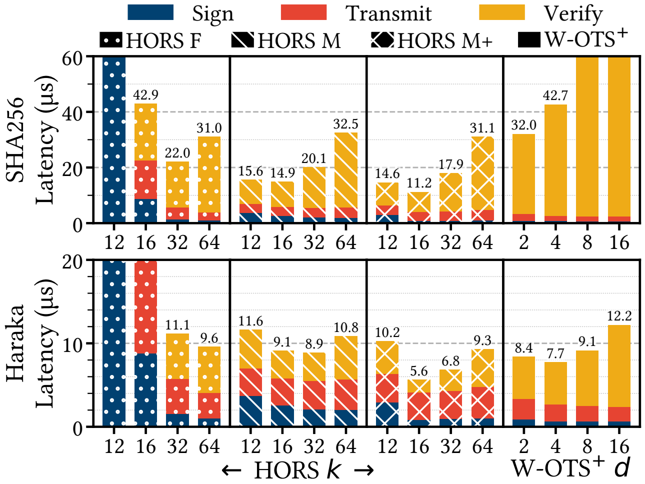


Figure 6: Sign-transmit-verify latency of DSig for 8 B messages using different hashing schemes (SHA256 (top) and Haraka (bottom)), and different HBSS configurations: HORS with factorized PK (HORS F), HORS with merklified PK (HORS M) and prefetched keys (HORS M+), and W-OTS$^+$.

key with the forest roots and the inclusion proofs of the revealed secrets (bottom of Figure 4). To avoid the overhead of checking Merkle proofs (i.e., computing around a hundred BLAKE3 hashes) on the critical path, we use a latency-hiding technique similar to the one in §4.4: signers send their complete public keys ahead of time to verifiers (by disabling background bandwidth reduction (§4.4)); verifiers precompute Merkle forests in their background plane, so Merkle proof verification on the critical path becomes mere string comparisons. We analyze this approach in the second part of Table 2, which shows that configurations with very few hashes ($k\leq16$) have tractable signature sizes, but come at the expense of significant background traffic and many background hashes.

W-OTS$^+$. W-OTS$^+$ differs from HORS in two main ways. First, W-OTS$^+$ secrets are hashed $d-1$ times to obtain the public key, where d is a depth parameter. Second, to sign, each secret is hashed a different number of times, as determined by the message to be signed, before being included in the signature. We lower sign latency by caching these hashes upon computation of the public key so that signing reduces to string copying. To verify a signature, we hash each element the required number of times to get to depth d, as determined by the signed message, and verify that the obtained results match the public key. Note that W-OTS$^+$ does not require embedding the public key in the DSig signature (Figure 5). A downside of W-OTS$^+$ versus HORS is that W-OTS$^+$ needs many more hashes on the critical path.

We analyze W-OTS$^+$ within DSig in the last part of Table 2, which shows that W-OTS$^+$ configurations with small d satisfy our requirements regarding signature size, background processing, and bandwidth consumption. Moreover, although they all require more hashes than HORS on the critical path, their signatures are smaller than the smallest HORS ones. Note that as d gets bigger, the gain in signature size is outweighed by the drastic increase in hash computations.

Conclusion. Our analysis points to three general configurations for further experimental evaluation: (1) HORS with factorized PK and k close to 64, (2) HORS with merklified PK and k close to 16, and (3) W-OTS$^+$ with d close to 4.

5.3 Evaluation

We measure the latency of signing an 8 B message, transmitting it along with its DSig signature, and verifying the signature for the sensible HBSS configurations presented in §5.2. Our experimental setup is detailed in §8. We consider three hash functions: SHA256 [77] (the slowest), BLAKE3 [75], and Haraka [55] (the fastest). Figure 6 shows the results for SHA256 and Haraka (BLAKE3 stands in between).

When using Haraka (bottom of Figure 6), HORS with factorized public keys (denoted *HORS F*) achieves its best end-to-end latency for k=64. For k<64, its latency increases in spite of having fewer hashes on the critical path due to the transmission time of larger signatures.

Surprisingly, HORS with merklified public keys (*HORS M*) signs and verifies only marginally faster despite its far lower number of hashes. This disappointing performance is a microarchitectural effect of the size of the Merkle forests. Indeed, for the assembly and verification of precomputed Merkle proofs to be fast, the relevant elements of their associated Merkle forest should be present in local cache (L1/L2).

However, CPU prefetching is inefficient due to the inherent randomness of Merkle proofs.

To demonstrate the benefit of having the Merkle proofs in the local cache, we modify DSig to prefetch public and private keys into the processor cache right before signing and verifying. The modified system (*HORS M+*) achieves an end-to-end latency of as little as 5.6 µs (signing in 0.9 µs, verifying in 1.5 µs and transmitting in 3.3 µs) for k=16. For smaller k, the keys do not entirely fit in the local cache, hence the performance degradation.

W-OTS$^+$ achieves its best latency of 7.7 µs for d=4 (signing in 0.7 µs, verifying in 5.1 µs and transmitting in 2.0 µs) where it strikes a good balance between few hashes (low d) and short signatures (high d). We omit prefetching W-OTS$^+$ keys, as it has negligible latency benefit (not shown).

When using SHA256 (a slower hash function, top of Figure 6), HORS with factorized public keys (HORS F) sees its verification time vastly increase for large k, while small k still has long transmission time. HORS with merklified public keys (HORS M) has better latency for smaller k, which differs from HORS M with Haraka. Indeed, it is preferable to have fewer SHA256 computations (small k) than smaller Merkle forests (large k) since SHA256 is considerably more expensive than cache misses, which is not the case of Haraka. Finally, the large number of hashes (68 in expectation) of W-OTS$^+$ makes it perform worse than the best configuration of each presented HORS variant.

5.4 Recommended Configuration

From our analytical (§5.2) and experimental (§5.3) studies, we recommend using W-OTS$^+$ with d=4 and Haraka. This configuration offers single-digit sign-transmit-verify latency, tractable 1,584 B signatures, and requires little background computation and networking. Although HORS with merklified keys can achieve lower latency, it is too costly (in compute, bandwidth, and CPU cache pollution) and its superior performance requires modifying applications to prefetch keys into the processor cache, which can be impractical. Note, however, that the choice of HBSS depends on the relative performance of hardware and software: in the future, if cache misses become cheaper and hashing becomes relatively more expensive, low-k HORS configurations could be appealing.

6 Applications

We apply DSig to key-value stores, a financial trading system, BFT broadcast, and BFT replication.

Key-value stores (HERD [51] and Redis [87]). State-of-the-art key-value stores provide microsecond-level performance and form the backbone of many data center applications, microservices, and cloud services. Key-value stores are used to keep security-sensitive information such as service configuration, session management data, cached queries, access control data, chat sessions, etc. Yet, most key-value stores lack *auditability*—the ability for a third-party to check a log of all operations (reads and writes) executed on the key-value store. More precisely, in an auditable key-value store, the server keeps a log of executed operation such that, for any operation *op* in the log, the server can prove to a third party that *op*'s client requested its execution. For example, the third party may be a forensics specialist or a prosecutor, who wants proof that a client requested access or modification to some data. The threat model is that clients may wish to bypass the audit (i.e., execute an operation undetected), while the server is honest. The proofs provided by the server are operations signed by clients and the key-value store must ensure that (a) if an operation signed by client C is in the audit log, then it was executed by the key-value store for client C, and (b) if an operation is executed by the key-value store for client C, then it appears in the audit log as an operation signed by C.

To provide auditability, a key-value store requires all requests to be signed by clients and logged by the server. The server must check the client signature before executing a request (otherwise a client could send a request with a bogus signature, which the server would execute without later being able to prove it), so traditional digital signatures significantly increase the latency of microsecond-scale key-value stores.

We use DSig to add auditability to two key-value stores: HERD and Redis. HERD is highly optimized for the RDMA networks present in data centers, while Redis is popular among web application developers. HERD provides simple `GET` and `PUT` operations on key-value pairs, while Redis also provides higher-level operations on common data structures, such as lists, maps, sets, etc. We modify both systems so that clients use DSig to sign all operations, and servers log and verify the signed operations before executing them. This logging requires 1.5 KiB of storage per operation due to DSig's signatures. Key-value stores have predictable signing and verifying processes: clients simply set their signatures hints to the server process. Logs can be persisted at the microsecond scale using persistent memory. This is not currently implemented due to lack of hardware, but data from Yang *et al.* [91] indicate that persistence would take less than 4 µs, and this latency can be masked by storing in parallel with signature verification. Vanilla HERD takes ≈2.5 µs to `GET` or `PUT` a key-value pair, while vanilla Redis takes ≈12 µs.

Financial trading system (Liquibook [73]). Liquibook provides an order-matching engine for financial trading—it matches buy and sell limit orders from clients. We consider a trading system built using Liquibook and RDMA for fast client-server communication. We use DSig to enhance the system and provide auditability, as we did for key-value stores. Signature hints are identical to key-value stores. Without auditability, the trading system takes ≈3.6 µs to process orders, of which ≈2 µs are spent on communication.

BFT broadcast (CTB [3]). Byzantine Fault Tolerance (BFT) is becoming more relevant in data centers, due to the need

Table 3: Configuration details of machines.

CPU	2× 8c/16t Intel Xeon Gold 6244 @ 3.60GHz
NIC/Switch	Mlnx CX-6 MT28908 / MSB7700 EDR 100 Gbps
Software	Linux 5.4.0-167-generic / Mlnx OFED 5.3-1.0.0.1

to tolerate failures beyond simple crashes [42, 44, 45, 69, 70]. Consistent broadcast is a core BFT primitive that prevents equivocation [23] and has many uses, as in money transfer [27, 40], consensus [1, 13, 25], multi-party computation [10] and decentralized learning [35, 38] protocols. We consider uBFT's state-of-the-art implementation of Consistent Tail Broadcast (CTB) and replace its signatures with DSig's to improve performance. Signing hints are simple, as each signature is verified by all processes running the protocol.

BFT replication (uBFT [3]). State machine replication (SMR) is the standard approach for fault-tolerance [2, 18, 66]. We consider uBFT, a microsecond-scale BFT SMR system for data centers. BFT SMR protocols, including uBFT, often employ signed messages to guard against Byzantine replicas. uBFT recognizes the high cost of digital signatures and follows a fast/slow path approach. The fast path avoids signatures and has a latency of 5 µs. The slow path uses signatures, with a latency of ≈220 µs. The slow path is triggered even without Byzantine behavior (e.g., due to process slowness), leading to latency fluctuations between its two modes of operation. We use DSig to replace the digital signatures in uBFT and improve its performance. Signing hints are simple, as each signature is verified by all processes running the protocol. Moreover, we use DSig's DoS-mitigation mechanism (§4) to prevent a malicious attack from triggering superfluous EdDSA verifications. More precisely, because uBFT is a quorum system, it can make progress with $n-f$ responses (n is the number of replicas, f is the maximum number that can be Byzantine). We make a small modification to uBFT to use the *canVerifyFast* function to prioritize handling of messages that do not incur the EdDSA signature check. As a process gets at least $n-f$ messages from non-Byzantine processes, it ignores the slow-to-check messages from Byzantine players.

7 Implementation

Our implementation of DSig has 3,019 lines of C++17 (CLOC [32]). We use our own implementation of HORS [84] and W-OTS$^+$ [46], the official implementations of BLAKE3 [75] and Haraka [55], and Dalek's implementation of EdDSA (Ed25519 [65]). BLAKE3 and Dalek's EdDSA use AVX2 for high performance. We use uBFT's framework [3], which provides fast point-to-point communication.

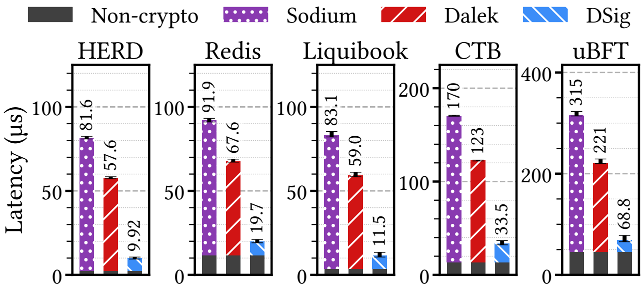


Figure 7: End-to-end latency of different applications using Sodium, Dalek or DSig for signatures. Printed values show the median latency; whiskers show the 10th and 90th %-iles.

8 Evaluation

We evaluate the performance of DSig and verify its suitability as a microsecond-scale signature system. We aim to answer the following:

- How do microsecond-scale applications that use signatures benefit from DSig's low latency (§8.1)?
- How does DSig's signing and verification latency compare to traditional signatures (§8.2, §8.3)?
- What is the throughput of DSig (§8.4)?
- How do DSig's higher bandwidth requirements impact its scalability (§8.5, §8.6)?
- How do we set DSig's EdDSA batch size (§8.7)?

Testbed. Our testbed is a cluster with 4 servers configured as shown in Table 3. The dual-socket machines have an RDMA NIC attached to the first socket. Our experiments execute on cores of the first socket using local NUMA memory. We accurately measure time using the TSC [43] via `clock_gettime` with the `CLOCK_MONOTONIC` parameter.

DSig configuration. We configure DSig per §5: in all experiments, we use W-OTS$^+$ with a depth d=4 as its HBSS. We dedicate a single core to DSig's background plane, which provides a high-enough throughput for our applications (§8.4). Unless specified otherwise, we use an EdDSA batch size of 128 (§8.7) and provide correct verifier hints when signing.

Baselines. We compare DSig against two well-known signature libraries: Sodium [8] (written in C) and Dalek [65] (written in Rust). Both implement the EdDSA signature scheme Ed25519—the fastest traditional scheme to date [17].

8.1 Application Latency

We configure applications with different signature schemes (Sodium, Dalek, DSig) and measure their latency. For the key-value stores, we use 16 B keys and 32 B values, 20% of `PUT` requests and 80% of `GET`s, where 90% of `GET`s return a value. For Liquibook, 50% of the requests are `SELL`s and 50% are `BUY`s. For CTB, we broadcast 8 B messages. Finally, for

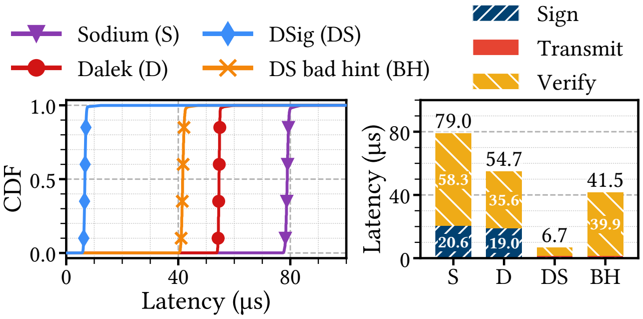


Figure 8: (Left) CDF of latency when signing, transmitting, and verifying the signature of 8 B messages using Sodium, Dalek and DSig with/without correct hints. (Right) Median latency breakdown. Transmission overhead is barely visible.

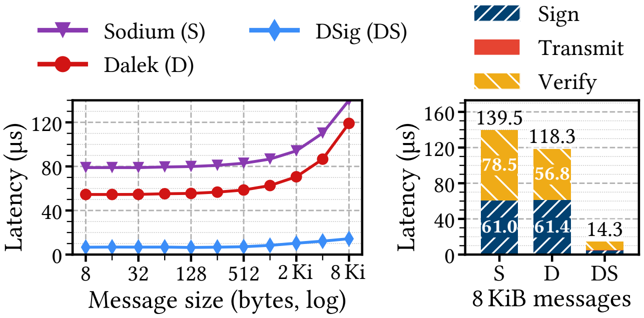


Figure 9: (Left) Latency to sign, transmit, and verify the signature of various-size messages using Sodium, Dalek and DSig with correct hints. (Right) Median latency breakdown for 8 KiB messages. Transmission overhead is invisible.

uBFT, we consider SMR operations of 8 B. We issue 10,000 requests one at a time to each application, measure the end-to-end latency, and report the 10th-, 50th-, and 90th-percentiles.

Figure 7 shows the results. For the three applications on the left, DSig provides auditability for a small cost: an increase of less than 7.9 µs in end-to-end latency. Sodium and Dalek add ≈79 µs and ≈55 µs, respectively, which is 10× and 7.0× DSig's overhead. In CTB, replacing Sodium (resp. Dalek) with DSig reduces the median cryptographic overhead by 87% (resp. 82%), and reduces the median end-to-end latency by 80% (resp. 73%). In uBFT, DSig reduces the median cryptographic overhead by 91% (resp. 87%), and reduces the median end-to-end latency by 78% (resp. 69%) compared to Sodium (resp. Dalek). DSig provides similar latency gains at the 90th percentile. In summary, across the tested applications, DSig significantly reduces cryptographic overheads and improves latency over the state of the art.

8.2 Latency of DSig

We study the latency to sign a message, transmit a signature, and verify a signature using DSig. We also consider the latency of incorrectly hinted DSig signatures for which EdDSA signatures are verified on the critical path. This represents the worst-case scenario for DSig. In each experiment, a process signs an 8 B message and transmits the signed message to a second process, which verifies the signature. We run each experiment 10,000 times for each signature scheme. The signature transmission latency is the incremental cost of adding the signature to a message, computed as the difference between transmitting a message with and without a signature. We estimate message transmission time as half of the round-trip time for ping-ponging the message.

Figure 8 shows the results. All signature schemes have stable latency up to the 99.9th percentile. Sodium and Dalek have similar signing median latency of 20.6 µs and 18.9 µs, respectively. While Sodium verifies in 58.3 µs, Dalek verifies in only 35.6 µs (39% faster) thanks to the use of AVX2 instructions. The (incremental) network latency is less than 100 ns for both since their signatures are merely 64 B. With correct hints, DSig takes 0.7 µs to sign and 5.1 µs to verify. This is 27× and 7.0× faster than Dalek, respectively. Interestingly, even though DSig's larger signatures lead to a 1.0 µs transmission overhead (more than 10× Dalek's), it has limited impact on its latency which is dominated by verification. Overall, DSig is 8.2× faster than Dalek. With incorrect hints, DSig's signature verification requires verifying both HBSS and EdDSA signatures, so verification latency increases to 39.9 µs (4.3 µs more than Dalek's). Signature generation, however, is not impacted as signers still benefit from background EdDSA precomputation and the total latency, although rising to 41.5 µs, is still 24% lower than Dalek's. Even with incorrect hints, DSig has much lower combined sign-transmit-verify latency than the state of the art.

8.3 Effect of Message Size on Latency

We study the effect of message size on the latency of DSig by running the experiments of §8.2 with varying message sizes.

Figure 9 (left) shows the results. With larger messages, DSig's total latency increases gradually but remains below 15 µs. Sodium's and Dalek's latencies are much higher. They also increase faster because they use a slower hash function than DSig (SHA256).[1] Figure 9 (right) shows the latency breakdown for the largest size (the breakdown for the smallest size is in §8.2). In all schemes, the split is about half-half to sign and verify, with negligible transmission time.

8.4 Throughput

We study DSig's throughput. In an experiment, a process signs small 8 B messages repeatedly with either a constant or

[1] Most signature schemes hash the input to operate on a fixed-size string.

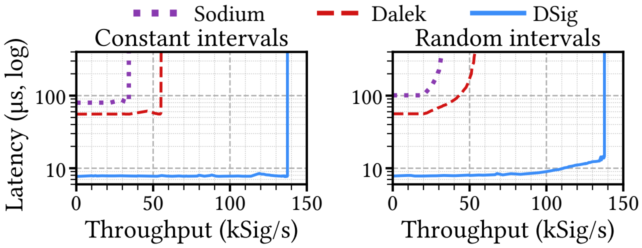


Figure 10: Latency-throughput graphs for Sodium, Dalek and DSig. Signatures are issued at constant or exponentially distributed intervals. All three use two cores on both sides; DSig dedicates one of them to its background plane.

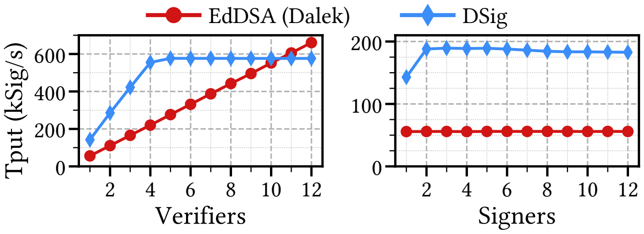


Figure 11: (Left) DSig's throughput with a signer sending the same signature to multiple verifiers. (Right) DSig's throughput with a verifier receiving different signatures from multiple signers. The NICs' bandwidth is limited to 10 Gbps.

an exponentially distributed random interval between signatures. The signer transmits the signature over the network to the verifier, which verifies it. We measure the total latency (sign plus transmit plus verify) and throughput of DSig, and compare it against Sodium's and Dalek's. In all experiments, the signer and the verifier use two cores each. DSig dedicates one core to each of its planes so that background events minimally impact foreground operations, while Sodium and Dalek use all cores to handle multiple messages in parallel.

Figure 10 shows the results as latency-throughput graphs with median latency and average throughput. With constant signature interval, all three systems exhibit stable latency until reaching maximum throughput. Sodium maintains a latency of ≈80 µs up to a throughput of 34 kSig/s where it is bottlenecked by verification time (58 µs). Dalek maintains a latency of ≈56 µs up to a throughput of 56 kSig/s where it is also bottlenecked by verification time (36 µs). DSig maintains a latency of ≈7.8 µs up to a throughput of 137 kSig/s where it is bottlenecked by the signer's background plane, which takes 7.4 µs to generate a new public key. We separately measured the verifier's background plane; it achieves a throughput of 3.6 MSig/s, so it is not a bottleneck. With a random signing interval, queuing occurs gradually, so the respective bottlenecks are less abrupt, causing a smoother latency degradation.

We run another experiment to measure the per-core throughput of DSig by running both of its planes on one core, and compare it to the per-core throughput of Dalek. While Dalek achieves 53 kSig/s signature generations (resp. 28 kSig/s verifications) per core, DSig achieves 131 kSig/s signature generations (resp. 193 kSig/s verifications) per core.

In summary, DSig sustains significantly higher throughput at much lower latency than EdDSA-based systems.

8.5 One-to-Many, Many-to-One Performance

We now study DSig's scalability and bottlenecks in one-to-many and many-to-one scenarios. In *one-to-many*, one signer signs a message and sends the signature to many verifiers; this pattern is common in distributed protocols. In *many-to-one*, many signers sign different messages and send them to the same verifier; this pattern is common in client-server applications. We run experiments where the signer(s) and verifier(s) use one foreground and one background core to work as fast as possible. We measure the aggregate verification throughput, and report the average. We consider a scenario where most of the network bandwidth (≈90%) is consumed by other activities, by limiting our NICs' bandwidth to 10 Gbps. This of course makes it harder for DSig to operate since it consumes more network bandwidth than other schemes. We compare the scalability of DSig to a two-core system based on Dalek.

Figure 11 shows the results. In one-to-many (left of figure), DSig's throughput increases until 577 kSig/s with 5 verifiers; at this point, the signer saturates its link to the verifiers, with the 1,584 B signatures and their 33 B background data accounting for ≈7 Gbps. Dalek scales more slowly than DSig with the number of verifiers, but it is not affected by bandwidth, as it continues to scale beyond 11 verifiers, at which point it surpasses DSig's throughput with 603 kSig/s using merely ≈300 Mbps to transmit 64 B signatures.

In many-to-one (right of figure), two signers are enough to achieve DSig's maximum throughput of 190 kSig/s as they saturate the verifier's foreground plane, which we set to run on a single core. As signing with Dalek is faster than verifying, Dalek does not scale beyond 1 verifier and achieves a maximum throughput of 53 kSig/s.

Overall, DSig's main scalability bottleneck compared to Dalek is its larger signatures.

8.6 Effect of Larger Signatures

We study how DSig's larger signatures affect application performance. In each experiment, we run a synthetic application where a server receives signed requests of a given size, checks their signature, spends some given processing time, and sends a 16 B unsigned reply. Similarly to §8.5, we limit the NICs' bandwidth to 10 Gbps, and compare the same application when using DSig or EdDSA. For fairness, EdDSA uses Dalek and pre-hashes messages with BLAKE3. In addition, we run an experiment with signatures disabled. The application runs with 4 cores: DSig uses 1 core for the background plane and

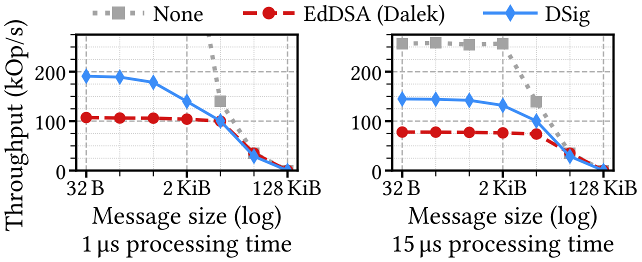


Figure 12: Request throughput of an application using signatures when NICs' bandwidth is constrained to 10 Gbps, for different request sizes and request processing times.

3 cores to handle requests, while the others use 4 cores to handle requests. We run enough clients to saturate the server. We consider 7 request sizes (32 B, 128 B, 512 B, 2 KiB, 8 KiB, 32 KiB and 128 KiB) and 2 processing times (1 µs and 15 µs).

Figure 12 shows the results. For both processing times, DSig outperforms EdDSA up to 8 KiB, after which it performs similarly to EdDSA. For small messages (32 B–512 B), the limited bandwidth has no impact on either scheme, so DSig significantly outperforms EdDSA thanks to its lower computational cost. With 2 KiB messages and 1 µs processing time, bandwidth impacts DSig while EdDSA is almost unaffected. Relative to 512 B messages, DSig's throughput decreases by 22%, while EdDSA's decreases by only 1.9%. Higher processing time offsets DSig's bandwidth bottleneck closer to 8 KiB messages. Beyond these points, the throughput of both DSig and EdDSA converges to that of the application that does not use signatures, as network bandwidth bottlenecks all three systems, making the overhead of signatures negligible.

In summary, DSig's higher per-core throughput lets applications reach higher throughput than with EdDSA even with limited network bandwidth, up to moderate-size messages.

8.7 EdDSA Batch Size

To set the size of EdDSA-signed key batches (§4.4), we run the same experiment as in §8.2 for different batch sizes, and we measure the latency and the per-core throughput. To take into account the impact of larger batches on low-end networks, we limit our NICs' bandwidth to 10 Gbps, as in §8.5 and §8.6.

Figure 13 shows the results, where a batch size of 1 means no batching. We see that batch sizes do not affect latency much (left of figure).[2] Throughput is different (right of figure): initially, batching improves throughput a lot for both signing and verifying. The gain dwindles when the amortized EdDSA cost per signature becomes a diminishing fraction of the overall computation as batches get larger. The best signing throughput is 135 kSig/s for batches of 32 keys, while the best verifying throughput is 206 kSig/s for batches of 4,096 keys. We pick a batch size of 128 as a balance.

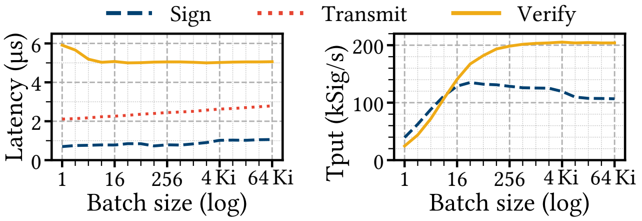


Figure 13: (Left) Median latency to sign, transmit, and verify a signature on an 8 B message with DSig for different EdDSA batch sizes. (Right) Single-core throughput of signing and verifying for different EdDSA batch sizes.

9 Related Work

HBSSs. HBSSs are well studied and prior work has proposed different implementations of them, many of which are variants of HORS [60,63,72,90]. Li *et al.* [63] proposed a variant targeted at smart grids with limited storage that reduces key and signature size but increases computation costs. Wang *et al.* [90] proposed a scheme with small signatures and microsecond performance, but it is limited to providing low ≈ 50-bit security. HORSE [72] reduces the cost of few-time signatures by repeatedly hashing the private key secrets, creating a matrix whose last row is the public key; however, it restricts the order in which applications can reveal public keys. W-OTS$^+$ [46] was proposed by Hülsing as a variant of W-OTS [34] with reduced signature and key sizes.

Online/offline signature schemes. The concept of online/offline digital signatures, in which heavy computation is done prior to knowing the message to sign, was first introduced by Even *et al.* [37]. So far, practical applications of the theoretical concept (including hybrid signature schemes) have targeted low-compute devices and/or wide area networks, with a focus on improving signature throughput or reducing bandwidth [53,62,64,78,92,93]. Recently, Esiner *et al.* [36] also recognized the importance of low-latency signatures, yet their solution is tailored for industrial control systems with tiny messages (25 bits), and does not provide self-standing signatures. No prior work addresses hybrid signatures in data centers with microsecond-scale performance.

Merkle-based signatures. Prior work proposes schemes that rely exclusively on HBSSs to sign (virtually) infinitely many messages, with the goal of attaining quantum resistance. Most of this work is based on XMSS [21], such as SPHINCS [15] and variants [16,47,58]. Instead of distributing keys regularly, these schemes efficiently pack an infinitude of one-time public keys using Merkle inclusion proofs [68]. These proofs need to be checked during signature verification, thus making the performance of such schemes be in the milliseconds.

Signature-like schemes. The cost of signatures has fueled alternatives for different scenarios. Message authentication codes (MACs) provide authentication and integrity of mes-

[2] The transmission latency differs from §8.2 due to the 10 Gbps NIC limit.

sages, but lack transferability, as parties use a shared secret to communicate. While MACs are widely used in networked systems to provide authenticated channels between two parties, they are not substitutes for signatures, as they provide weaker properties, they are harder to use, and they are more susceptible to protocol mistakes. In particular, using MAC-based mechanisms in BFT protocols has several drawbacks: (1) These mechanisms are ad-hoc and highly dependent on the protocol: some require MAC vectors [24] others require MAC matrices [5]; others explicitly prefer or mandate signatures over MACs for critical messages [3,4,26]. (2) These mechanisms add complexity to the BFT protocols, e.g., by requiring a fast-slow path approach where the fast path avoids signatures but the slow path (or view change) still uses them [3,4,24,26,29,56]; this added complexity increases their attack surface [26]. (3) These mechanisms often add messages and roundtrips to the protocols [5,29,79], and/or lower their resilience to failures [5,79].

Some systems make extra assumptions to provide MAC with some form of transferability. TESLA [79] assumes clock synchrony and has time windows during which MACs are generated and transmitted; afterward, the MAC secrets are revealed to check previously seen MACs. This idea provides only a limited form of transferability and increases the latency of verification. Using trusted hardware [12, 52, 61], such as trusted execution environments (TEEs), one can provide MACs with transferability by hiding the secret and computing the MACs in the TEE so that every TEE owner can verify the MACs but only a designated TEE can create them.

10 Conclusion

DSig is the first digital signature system for microsecond-scale applications. DSig achieves single-digit microsecond latency for signing and verifying messages—27× and 7× faster than the prior state of the art—while achieving higher throughput. To achieve that, DSig introduces a new hybrid signature scheme that uses knowledge of where signatures are issued and verified in the common case. DSig can bring auditability to latency-critical applications with a small latency overhead, or replace other signature schemes in applications that use them. Ultimately, we believe that DSig makes digital signatures fast enough to broaden their use in data centers as a powerful security building block.

Acknowledgments

We thank the anonymous reviewers and our shepherd Steve Hand for their valuable comments, as well as the anonymous artifact evaluators for reviewing our implementation. We also thank Dariia Kharytonova, Ed Bugnion, Jean-Philippe Aumasson, Khashayar Barooti, Phillip Gajland, Pierre-Louis Roman, and Serge Vaudenay for their feedback.

References

[1] Ittai Abraham, Marcos K. Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing*, pages 4–19, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, Berkeley, CA, USA, 2020. USENIX Association.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. uBFT: Microsecond-scale BFT using disaggregated memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 862–877, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal Byzantine computing. In *Proceedings of the 35th International Symposium on Distributed Computing*, volume 209 of *DISC '21*, pages 3:1–3:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. Matrix signatures: From MACs to digital signatures in distributed systems. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 16–31, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Jean-Philippe Aumasson. Too much crypto. *Cryptology ePrint Archive*, Paper 2019/1492, 2019.

[8] Jean-Philippe Aumasson, Daniel J. Bernstein, Alex Biryukov, Tung Chou, Frank Denis, Daniel Dinu, Romain Dolbeau, Jason A. Donenfeld, Niels Duif, Adrien

Gallouet, Jack Grigg, Mike Hamburg, Dmitry Khovratovich, Tanja Lange, Adam Langley, Isis Lovecruft, Andrew Moon, Samuel Neves, Yoav Nir, Colin Percival, Alexander Peslyak, Thomas Pornin, Bart Preneel, Peter Schwabe, George Tankersley, Henry de Valence, Filippo Valsorda, Zooko Wilcox-O'Hearn, Christian Winnerlein, Hongjun Wu, and Bo-Yin Yang. The Sodium cryptography library, 2022. Accessed: April 14, 2023.

[9] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In *Topics in Cryptology – The Cryptographers' Track at the RSA Conference*, CT-RSA '18, pages 219–242, Cham, Zug, Switzerland, 2018. Springer International Publishing.

[10] Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. Asynchronous MPC with a strict honest majority using non-equivocation. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 10–19, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[12] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, page 222–237, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Naama Ben-David, Benjamin Y. Chan, and Elaine Shi. Revisiting the power of non-equivocation in distributed protocols. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, page 450–459, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. Ed25519: high-speed high-security signatures, 2017. Accessed: April 14, 2023.

[15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In *Advances in Cryptology – Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '15, pages 368–397, Berlin, Germany, 2015. Springer-Verlag.

[16] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems, 2022. Accessed: April 14, 2023.

[18] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State Machine Replication for the masses with BFT-SMART. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, page 355–362, NW Washington, DC, USA, June 2014. IEEE Computer Society.

[19] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, page 645–650, Berkeley, CA, USA, 2018. USENIX Association.

[20] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, SP '21, pages 1659–1676, New York, NY, USA, 2021. IEEE.

[21] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS – A practical forward secure signature scheme based on minimal security assumptions. In *Proceedings of the 4th International Workshop on Post-Quantum Cryptography*, PQCrypto '11, pages 117–129, Berlin, Germany, 2011. Springer-Verlag.

[22] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: April 14, 2023.

[23] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, Berlin, Germany, 2nd edition, 2011.

[24] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[25] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, page 301–308, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 153–168, USA, 2009. USENIX Association.

[27] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, New York, NY, USA, 2020. IEEE.

[28] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, Paper 2016/086, 2016.

[29] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 177–190, USA, 2006. USENIX Association.

[30] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '19, page 35–48, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Al Danial. CLOC: Count Lines of Code, 2022.

[33] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[34] Chris Dods, Nigel P. Smart, and Martijn Stam. Hash based digital signature schemes. In *Proceedings of the 10th IMA International Conference on Cryptography and Coding*, IMACC '05, pages 96–115, Berlin, Germany, 2005. Springer-Verlag.

[35] El Mahdi El-Mhamdi, Sadegh Farhadkhani, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Collaborative learning in the jungle (decentralized, byzantine, heterogeneous, asynchronous and nonconvex learning). In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 25044–25057, New York, NY, USA, 2021. Curran Associates, Inc.

[36] Ertem Esiner, Utku Tefek, Hasan S. M. Erol, Daisuke Mashima, Binbin Chen, Yih-Chun Hu, Zbigniew Kalbarczyk, and David M. Nicol. LoMoS: Less-Online/More-Offline signatures for extremely time-critical systems. *IEEE Transactions on Smart Grid*, 13(4):3214–3226, July 2022.

[37] Shimon Even, Oded Goldreich, and Silvio Micali. On-Line/Off-Line digital signatures. In *Advances in Cryptology – Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, page 263–275, Berlin, Germany, 1989. Springer-Verlag.

[38] Sadegh Farhadkhani, Rachid Guerraoui, Nirupam Gupta, Lê Nguyên Hoang, Rafael Pinot, and John Stephan. Robust collaborative learning with linear gradient overhead. In *Proceedings of the 40th International Conference on Machine Learning*, ICML '23, jmlr.org, 2023. The Journal of Machine Learning Research.

[39] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[40] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 307–316, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *Proceedings of the 2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 101–120, Berkeley, CA, USA, 2022. USENIX Association.

[42] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage*, 14(3), October 2018.

[43] Red Hat. RHEL for real time timestamping, 2020. Accessed: April 14, 2023.

[44] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Andreas Hülsing. W-OTS+ – Shorter signatures for hash-based signature schemes. In *Progress in Cryptology – Proceedings of the 6th International Conference on Cryptology in Africa*, AFRICACRYPT '13, pages 173–188, Berlin, Germany, 2013. Springer-Verlag.

[47] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward secure signatures on smart cards. In *Proceedings of the 19th International Conference on Selected Areas in Cryptography*, SAC '12, pages 66–80, Berlin, Germany, 2012. Springer-Verlag.

[48] Illumina. Advancing research and clinical genomic data solutions, 2023. Accessed: April 14, 2023.

[49] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, August 2001.

[50] Simon Josefsson and Ilari Liusvaara. Edwards-Curve digital signature algorithm (EdDSA). RFC 8032, January 2017.

[51] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[52] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th European Conference on Computer Systems*, EuroSys '12, page 295–308, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Jayaprakash Kar. Provably secure online/off-line identity-based signature scheme for wireless sensor network. *Cryptology ePrint Archive*, Paper 2012/162, 2012.

[54] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, Boca Raton, FL, USA, 2nd edition, 2014.

[55] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 – efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology*, 2016(2):1–29, February 2016.

[56] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, oct 2007.

[57] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: Abuse in authenticode code signing. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 465–478, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Mikhail A. Kudinov, Andreas Hülsing, Eyal Ronen, and Eylon Yogev. SPHINCS+C: Compressing SPHINCS+ with (almost) no cost. *Cryptology ePrint Archive*, Paper 2022/778:48, 2022.

[59] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, Menlo Park, CA, USA, October 1979.

[60] Jaeheung Lee and Yongsu Park. HORSIC+: An efficient post-quantum few-time signature scheme. *Applied Sciences*, 11(16), August 2021.

[61] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[62] Fagen Li, Di Zhong, and Tsuyoshi Takagi. Practical identity-based signature for wireless sensor networks. *IEEE Wireless Communications Letters*, 1(6):637–640, December 2012.

[63] Qinghua Li and Guohong Cao. Multicast authentication in the smart grid with one-time signature. *IEEE Transactions on Smart Grid*, 2(4):686–696, December 2011.

[64] Joseph K. Liu, Joonsang Baek, Jianying Zhou, Yanjiang Yang, and Jun Wen Wong. Efficient online/offline identity-based signature for wireless sensor network. *International Journal of Information Security*, 9(4):287–296, August 2010.

[65] Isis Agora Lovecruft and Henry De Valence. ed25519-dalek: Fast and efficient rust implementation of ed25519 key generation, signing, and verification in rust, 2022.

[66] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, page 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[67] Ralph C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, April 1978.

[68] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology – Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, pages 218–238, Berlin, Germany, 1989. Springer-Verlag.

[69] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, page 177–190, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the 2018 Internet Measurement Conference*, IMC '18, page 393–407, New York, NY, USA, 2018. Association for Computing Machinery.

[71] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Accessed: April 14, 2023.

[72] William D. Neumann. HORSE: An extension of an r-time signature scheme with fast signing and verification. In *International Conference on Information Technology: Coding and Computing, Volume 1*, ITCC '04, pages 129–134 Vol.1, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[73] Eric Newhuis. Liquibook: Open source order matching engine, 2022.

[74] NVIDIA. NVIDIA ConnectX-6 DX datasheet, 2022. Accessed: April 14, 2023.

[75] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3, 2022.

[76] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, page 361–377, Berkeley, CA, USA, 2019. USENIX Association.

[77] Wouter Penard and Tim van Werkhoven. *On the secure hash algorithm family*, pages 1–18. Wiley, New York, NY, USA, 2008.

[78] Cong Peng, Min Luo, Li Li, Kim-Kwang Raymond Choo, and Debiao He. Efficient certificateless online/offline signature scheme for wireless body area networks. *IEEE Internet of Things Journal*, 8(18):14287–14298, September 2021.

[79] Adrian Perrig and J. D. Tygar. *TESLA Broadcast Authentication*, pages 29–53. Springer, Boston, MA, USA, 2003.

[80] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[81] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355, New York, NY, USA, 2015. Association for Computing Machinery.

[82] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, page 145–160, Berkeley, CA, USA, 2018. USENIX Association.

[83] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, Boston, MA, USA, 2001.

[84] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *Proceedings of the 7th Australian Conference Information Security and Privacy*, ACISP '02, pages 144–153, Berlin, Germany, 2002. Springer-Verlag.

[85] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[86] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, CCS '99, page 93–100, New York, NY, USA, 1999. Association for Computing Machinery.

[87] Salvatore Sanfilippo. Redis, 2022.

[88] Synopsys. Synopsys and Amazon Web Services, October 2023. Accessed: April 14, 2023.

[89] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, WOEC '96, Berkeley, CA, USA, 1996. USENIX Association.

[90] Qiyan Wang, Himanshu Khurana, Ying Huang, and Klara Nahrstedt. Time valid one-time signature for time-critical multicast data authentication. In *Proceedings of the 28th IEEE International Conference on Computer Communications*, INFOCOM '09, pages 1233–1241, New York, NY, USA, 2009. IEEE.

[91] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST '20, page 169–182, Berkeley, CA, USA, 2020. USENIX Association.

[92] Andrew Chi-Chih Yao and Yunlei Zhao. Online/Offline signatures for low-power devices. *IEEE Transactions on Information Forensics and Security*, 8(2):283–294, February 2013.

[93] Ping Yu and Stephen R. Tate. Online/Offline signature schemes for devices with limited computing capabilities. In *Topics in Cryptology – The Cryptographers' Track at the RSA Conference*, CT-RSA '08, pages 301–317, Berlin, Germany, 2008. Springer-Verlag.

Ransom Access Memories: Achieving Practical Ransomware Protection in Cloud with `DeftPunk`

Zhongyu Wang[*], Yaheng Song[*], Erci Xu[†], Haonan Wu, Guangxun Tong, Shizhuo Sun,
Haoran Li, Jincheng Liu, Lijun Ding, Rong Liu, Jiaji Zhu, and Jiesheng Wu
Alibaba Group

Abstract

In this paper, we focus on building a ransomware detection and recovery system for cloud block stores. We start by discussing the possibility of directly using existing methods or porting one to our scenario with modifications. These attempts, though failed, led us to identify the unique IO characteristics of ransomware, and further drove us to build `DeftPunk`, a block-level ransomware detection and recovery system. `DeftPunk` uses a two-layer classifier for fast and accurate detection, creates pre-/post-attack snapshots to avoid data loss, and leverages log-structured support for low overhead recovery. Our large-scale benchmark shows that `DeftPunk` can achieve nearly 100% recall across 13 types of ransomware and low runtime overhead.

1 Introduction

Ransomware has become increasingly prevalent in recent years, posing a major threat to data security. Recent reports show that ransomware attacks have caused billions of dollars in losses to individuals and organizations [27, 29]. Typically, attackers exploit system vulnerabilities to access and encrypt users' data for ransom.

Cloud services are also facing increasingly aggressive ransomware threats. In ALIBABA cloud, our cloud block store (a.k.a., Elastic Block Storage, or EBS) has been under constant attacks. In Q3 of 2022 alone, we have received nearly one thousand reports from our EBS Virtual Disk (VD) users, yielding a 118% increase over the entire year of 2021.

To combat ransomware, practitioners have proposed various detection and recovery methods. At the application layer, users can use antivirus software and firewalls [21, 24, 31] to identify suspicious behaviors by monitoring file access and checking malware signatures. At the OS layer, recent works have demonstrated their ability to detect ransomware activities via behavioral analysis [33, 41, 45, 57]. Third, at the hardware layer, prior works have shown that, with hardware-assistance from customizable devices, the log-structured design of SSDs can be leveraged to detect and roll back ransomware activities [34, 35, 43].

Unfortunately, these traditional methods cannot be directly applied for our EBS. First, the application/OS layer methods require strong user cooperation. As a cloud vendor, we are unable to enforce our users to use specific software and/or OS. In addition, field statistics suggest that tenants may not always keep the software up to date, leaving potential vulnerabilities. Second, cloud providers usually use commodity hardware for cost efficiency and portability. Thus, it is impractical for us to build protection that only works on specialized devices. Third, certain vendors, including us, have already provided (periodical) VD snapshot for data recovery. Simply relying on such mechanisms may not be favorable for users, as they may lose the data between two snapshots due the hour-level interval and high CapEx led by the extra storage space

Existing solutions, while failing to work directly, inspire us to use the IO characteristics of ransomware (e.g., excessive Write-After-Read access) for detection, and leverage the garbage collection (GC) of log-structured design for recovery. This is because the block store service, by default, can monitor and analyze the block-level IO traffic. Moreover, our EBS is built on an append-only distributed file system (DFS), and thus supports similar mechanism to roll back data that have not been reclaimed by GC (such as SSD). The benefits of such design are two-fold. First, it does not rely on certain software or hardware support. Second, it can be easily deployed with little extra overhead imposed.

Therefore, we started exploring the possibility of building this solution. However, real world data show that such a preliminary attempt fails to deliver satisfactory accuracy in detection. We assume the main reason is that the existing detection algorithms—bounded by their prerequisites (e.g., need to use SSD's weak SoC) and lack of real-world access/analysis—can not effectively and efficiently distinguish ransomware traffic from normal IOs. Additionally, we find that the existing recovery methods can not always guarantee lossless data recovery due to the limited time of multi-version support.

In this paper, we propose `DeftPunk`, a block-level ransomware detection and recovery system for the cloud. Based on extensive comparisons between ransomware and normal workloads, `DeftPunk` constructs a rich set of features and uses a two-layer classifier for fast and accurate ransomware detection. To avoid data loss, `DeftPunk` creates pre- and post-attack snapshots to "lock in" the effects of the attack, and persists all modifications in between. For recovery, `DeftPunk` follows a "undo-redo" strategy to roll back the data to the pre-attack state and only redo the users' writes.

[*] Equal contributions.
[†] Corresponding author.

Specifically, we first assemble a large-scale ransomware dataset by collecting more than 140 hours of block-level IO traces from 13 mainstream ransomware (e.g., Wannacry [30] and Mallox [18]) and 16 types of real-world workloads from our EBS. The comprehensive comparison shows that, apart from the well-known Write-After-Read (WAR) pattern, ransomware also exhibits other characteristics, such as read to write ratio, access offset distribution and frequent access on the system disk. These observations help us to extend the feature set of `DeftPunk` by including IO statistics, dependency, working set size, offset and certain LBA spatial access.

The above extended features help `DeftPunk` to achieve higher accuracy. But, for deployment efficiency under the sheer volume of VDs and their IOs, we also need to consider the runtime overhead. Therefore, `DeftPunk` adopts a two-layer model. The first layer, focusing on eliminating false positives (i.e., filtering normal IOs), uses a straightforward decision tree algorithm with simple features (only requiring O(1) computation). The positive cases classified by the first layer will be further sent to the second layer for a double check. The second layer emphasizes on exposing all ransomware cases, and thus uses a more sophisticated algorithm (i.e., XGBoost [38]) with the complete set of features.

Based on the output of the two-layer model, `DeftPunk` can create a pair of snapshots right before and after the attack. Moreover, with multi-version support by the EBS log-structured design and GC pausing, `DeftPunk` can persist all data modifications during the attack. For recovery, `DeftPunk` follows a "undo-redo" strategy to roll back the data to the pre-attack state, and, based on a rule-based model, only redo the writes made by the users.

Based on our assembled dataset, we benchmark `DeftPunk` with a set of state-of-the-art ransomware detection methods. The results show that `DeftPunk` can always achieve nearly 100% recall with 95.8% precision, outperforming all other peers. Moreover, `DeftPunk` only uses 7 vCPU cores for processing 1 million IOPS for detection and can recover valid data at 4.62 GB per second. We have deployed `DeftPunk` in our EBS service for a few invited users, and it has successfully prevented 2 attacks with data fully recovered.

The contributions of this paper are summarized as follows:

- We assemble and release a large-scale ransomware benchmark with real-world traces[1].
- We build `DeftPunk`, a practical ransomware detection and recovery system for the cloud EBS.
- We extensively evaluate `DeftPunk` and the results show that `DeftPunk` outperforms peers by up to 95.77% in precision and nearly 100% in recall.

The rest of the paper is organized as follows. §2 gives a brief overview of our EBS and background of ransomware. §3 discusses the existing solutions and related work. §4 identifies the goals and challenges. §5 presents the design of `DeftPunk` and §6 shows the evaluation of `DeftPunk` in depth. We end this paper with a discussion on `DeftPunk`'s limitations in §7 and a short conclusion in §8.

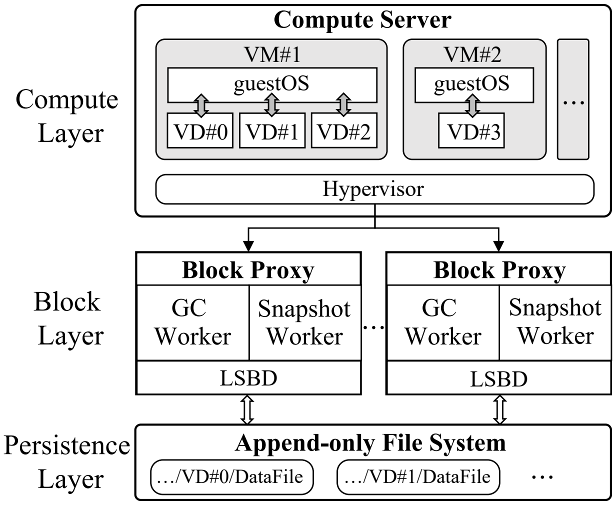


Figure 1: Overview of EBS Architecture. **VD**: *Virtual Disk*; **LSBD**: *Log-Structured Block Device*. **GC**: *Garbage Collection*.

2 Background

2.1 EBS in ALIBABA Cloud

Elastic Block Storage (EBS) serves as a cornerstone in today's cloud. To provide virtual block devices to users with high flexibility and availability, our EBS, similar to other vendors' architecture [9–11, 17], follows a "compute-to-store" philosophy. With this setup, the storage clusters are physically disaggregated—interconnected by data center network—from the compute servers (and subsequently the virtual machines running on them).

Figure 1 illustrates an overview of the EBS architecture. On the compute end, each server can host multiple Virtual Disks (VDs) and also embeds a client within the hypervisor to forward VD's requests to backend storage clusters. Once the *Block Proxies* receive the VDs' requests, they will then persist/fetch the data to/from the corresponding file in the distributed file system (DFS).

Like Google [49], Azure [37], and Alibaba Cloud [48], we too adopt a log-structured distributed file system (DFS) as the storage backend. The Block Proxy employs a Log-Structured Block Device (LSBD) to transform VDs' IO to an appending-only DataFile provided by the underlying DFS. One key functionality is to register the mapping (i.e., from VD LBA to location in DataFile) in a per-VD IndexMap to track latest location of data in the DataFile. Also, this requires a garbage collection (GC) for reclaiming space from stale data. In addition, we also set up a Snapshot Worker to allow users to create snapshots of their VDs and store them as binaries.

[1] The dataset is at: https://tianchi.aliyun.com/dataset/177511?lang=en-us

Layer	Mechanism	Schemes
App.	Anti-virus real-time scan	Fortinet [24], Kaspersky [21], Windows defender [31]
OS	File system behavioral analysis	CM&CB [33], UNVEIL [45], WaybackVisor [41], Towards. [57]
HyperV.	Scheduled snapshot	AWS [3], HUAWEI [4]
HW.	Detection with real-time rollback	FlashGuard [43], SSD-insider [34], SSD-insider++ [35], RSSD [53]

Table 1: Common protections against ransomware. **App.**: *Application*; **HyperV.**: *Hypervisor*; **HW.**: *Hardware*.

2.2 Ransomware

Ransomware, such as WannaCry [30], has been rampantly spreading across the globe, leading to billions of dollars financial losses. The typical procedure of a ransomware attack has three steps: (1) the attacker infects the victim's machine via weak password and/or system vulnerabilities; (2) the ransomware encrypts the victim's data; (3) the victim is instructed to pay a certain amount of ransom (e.g., cryptocurrency) for decryption.

Both industry and academia have been working on ransomware detection and protection. In Table 1, we summarize the commonly used methods by layers. First, at the Application layer, users can install antivirus software and firewalls [21,24,31] to monitor the suspicious behaviors (e.g., frequent access to unauthorized files and encryption) or match the signatures in the viruses database. Second, by intercepting system calls and file access patterns within the OS, recent studies have also proposed to detect ransomware activities via behavioral analysis [33,41,45,57]. Third, users or vendors could ask the hypervisor to take snapshots of the whole runtime to directly recover the data. This method is widely available by major cloud service providers [3,4]. Finally, many prior works have shown that the log-structured design of SSDs can be leveraged to detect and rollback ransomware activities [34,35,43]. The key idea is that valid data which are overwritten and encrypted by the ransomware will be marked as stale. But, the Flash Translation Layer (FTL) usually would not immediately reclaim their space, providing an opportunity to recover users' data before garbage collection.

3 Motivation and Related Work

The cloud is no stranger to ransomware attacks, especially when an increasing number of users are migrating their sensitive data to the public cloud [5–7,13,14,26–29]. A recent report by Zscaler cloud [7] states that ransom attacks have increased by 38% from April 2022 to April 2023, and they predict that attackers are likely to develop new types of ransomware and campaigns optimized for targeting cloud services and workflows. Sophos report [27] also indicates that the average ransom has increased from $812,380 in 2022 to $1,542,333 in 2023, not to mention the cost of the data recovery process and the losses due to downtime. In our cloud, we have also been witnessing a growing number of ransomware attacks on users. In just one quarter (2022 Q3), our cloud has recorded nearly one thousand ransom incidents, leading to an increase of 118% compared to the previous year.

One might wonder, with all the protection approaches available (see Table 1) and vendors' high emphasis on data security, why ransomware attacks are still so prevalent in the cloud. Here, we summarize the key reasons and challenges based on our observations and statistics from the field.

Human mistakes. User awareness is the first, and often the weakest, line of defense against ransomware attacks. To be human is to err, so it is not uncommon for users to fall victims to phishing or malicious emails and other malware. A recent survey by Fortinet [28] reports that phishing remains the top ransom tactic (56%). Other ransomware reports, such as those from Sophos [27] and SpyCloud [29], have likewise emphasized the vulnerability of humans in ransomware defense. Similarly, in our cloud, we discover that nearly all ransomware attacks start with a negligence being exploited (e.g., outdated software and weak password).

Lack of protection in VM. Normally, with antivirus software properly installed and security patches regularly updated, VM should be able to operate safely even under the threats of ransomware. However, in the cloud, VMs, can often run in an under-protected environment due to the following reasons.

First, only a small fraction of VMs are under sufficient antivirus software protection. For instance, Zscaler [2] reports that 17% of organizations are running workloads on unprotected virtual machines which is consistent with our observations from the field. Even for ones who have, they can still be at risks of latest attacks as 28% of VMs attacked by ransomware are running with vulnerable outdated OS and software. Third, while we provide OS images with built-in security support and automatic updates, only 19.4% of users opt in. One main reason, after discussion with multiple users, is that they tend to reuse their own OS images for consistency and compatibility after migrating to cloud.

Snapshot protection is expensive and coarse-grained. Our EBS allows users to persist the Virtual Disk (VD) as a snapshot and later use it to restore to a certain point in time. By taking periodical snapshots, users can conveniently recover from a ransomware attack by simply restoring the VD to the most recent checkpoint before the attack. However, in practice, this is not the case. First, the snapshot-based protection can be expensive. Even with our latest incremental snapshot service, the cost incurred from snapshots can easily 2.5 times more than the monthly rental of the VD, assuming VD is under normal traffic and snapshots are taken on hourly basis. Moreover, even if users are willing to pay the price, recovery by periodical checkpoint is still coarse-grained in the cloud EBS. For example, the highest specification VD in our cloud can achieve a throughput of 4000 MB/s and 100M IOPS. In this case, even if we use minute-level periodic snapshots, it would still result in a significant amount of data loss.

Hardware-based protection is impractical. Academia have proposed multiple approaches for defending against ransomware attacks via hardware assistance [34, 35, 43, 53]. However, these attempts would fall short for large-scale cloud deployment, such as EBS. First, previous works are based on specialized and/or prototype hardware, such as Open-Channel SSDs [34] or FPGAs, yielding a small chance for large-scale deployment. In addition, even if manufacturers manage to produce such hardware, it is still impractical as ransomware evolves rapidly. At the same time, frequently updating the firmware or providing backward compatibility for legacy devices would be a huge challenge.

4 Goals, Opportunities and Challenges

To this end, we have shown that the existing ransomware protection mechanisms can not be shoehorned to cloud services such as EBS. In this section, we first list the design goals of an ideal ransomware protection mechanism for EBS. Then, we discuss the potential opportunities for a practical detection approach based on intrinsic properties of EBS and the ransomware attack patterns. Finally, we present the insights we gained and challenges we faced from building and exploring this preliminary design.

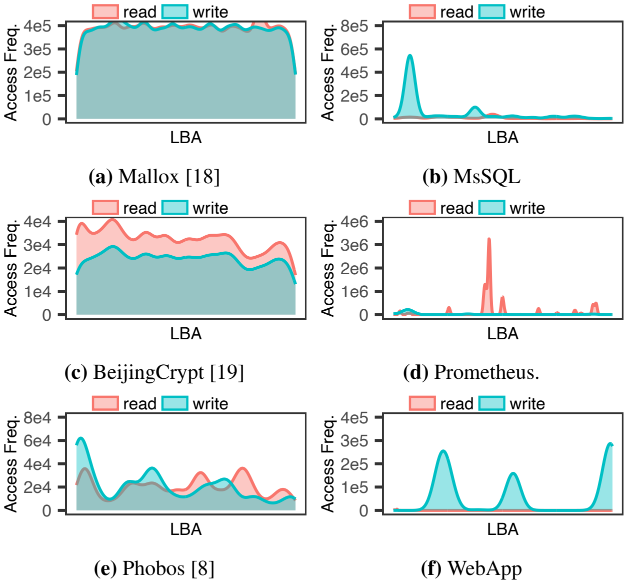
4.1 Design Goals

The role of a cloud vendor profoundly limits our ability and choices in applying existing techniques. Therefore, to develop a practical ransomware detection for EBS, we start by identifying the key requirements.

- *No data lost/tainted after recovery.* The most fundamental requirement for ransomware protection is to ensure that, once detected, no ransomed data is lost during recovery. In other words, all user issued IOs—before, during and after the attack—should be preserved and no tainted data left. On top of that, we need recover data in a timely manner with minimal effort from users.
- *Transparent to users.* The monitoring and detection should not rely on users' awareness or cooperation with the vendor, such as installing certain softwares, using specific OS, or updating patches under certain schedules. In addition, to obey privacy and security protocols, we also cannot directly control the VMs and/or insert (kernel) modules to proactively defend ransomware attacks.
- *Hardware independent to vendors.* As a cloud vendor, our solution should be based on commodity products, instead of depending on features only provided by specialized hardware (e.g., customizable Open-Channel SSDs) or prototype devices (e.g., FPGAs). Moreover, even for the commodity products, the ideal solution should provide backward compatibility, meaning that it can support legacy devices such as early models of SSDs or even HDDs.
- *Low runtime overhead.* The proposed solution should run with low resource consumption (e.g., CPU, memory, space, and network bandwidth). This is because, given the destructive impacts and the increasing prevalence of ransomware, the occurrence—compared to the massive volumes of VDs—is still rare in the wild. For example, on average, less than 0.001% of the VDs in our cloud are subjected to ransomware attacks on a daily basis. Hence, the high CapEx led by high resource consumption would not be acceptable to vendors or tenants.

4.2 Opportunities

While previous ransomware prevention techniques fail to work directly in EBS, we discover that the log-structured block device (LSBD) design bears great similarities with SSD internals. This motivates us to explore the possibility of borrowing ideas from the existing hardware-based methods. Next, we discuss the two similar opportunities our EBS shares with the hardware-based ransomware protection.



(a) Mallox [18] **(b)** MsSQL

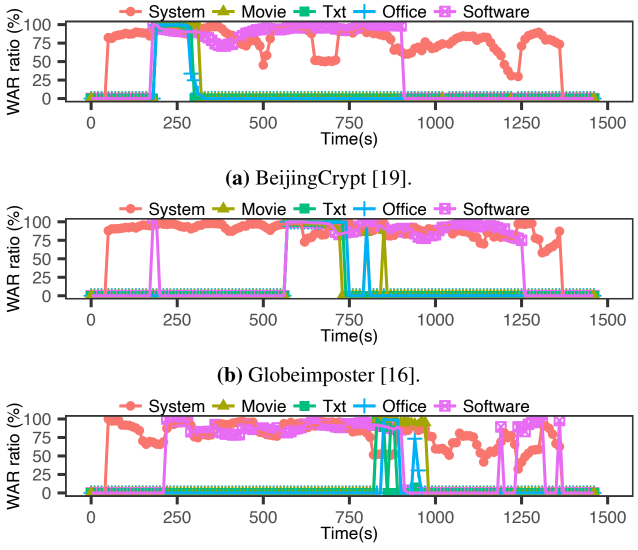
(c) BeijingCrypt [19] **(d)** Prometheus.

(e) Phobos [8] **(f)** WebApp

Figure 2: Comparation of LBA access frequency between ransomware and regular applications

Block-level ransomware access pattern. Ransomware usually follows a read-encrypt-write procedure on users' data. For example, a recent survey indicates that up to 76% of ransomware employed encryption-based attack mode [27]. Previous studies have shown that this access pattern holds across different ransomware families and can be caught at file system level (e.g., directory traversal, file type change, access frequency, etc [46, 54]) and device (i.e., SSD) level (e.g., statistics of erasure IO in [34, 35]). As Block Proxies handle VD requests in the format of Logical Block Address (LBA), we further explore the possibility of detecting ransomware attacks at the block level.

We start with the spatial and temporal patterns of typical ransomware. For the spatial pattern, Figure 2 (a)(c)(e) and



(a) BeijingCrypt [19].

(b) Globeimposter [16].

(c) Babuk [12].

Figure 3: Write-After-Read IO ratio by time on ransom attack

(b)(d)(f) respectively illustrate the LBA access patterns of a VD under ransom attacks and normal workloads. The x-axis represents the normalized LBA, and the y-axis shows the frequency of access within 10 minutes. By comparing the two, we can see an outstanding difference. The ransomware tends to have a similar amount of reads and writes to the same LBA. This is consistent with observations from previous studies [34, 35]. Namely, the "read, encrypt, and write back" pattern of ransomware is manifested as erasure IO (i.e., EIO) or Write-After-Read IO (i.e., WAR IO) at the block-level.

We then examine the temporal pattern of ransomware attack. Figure 3 shows the variations in the WAR ratio (i.e., the ratio of WAR IO to the total write requests) of three VMs (each with five VDs) under various ransomware attack. In each VM, the five VDs consist of one system drive and four data drives loaded with different types of data.

For example, Figure 3(a) illustrates the variations in the WAR ratio of the VM when it is under a BeijingCrypt attack. The WAR ratio of the system drive rapidly rises from 0 to 100% around 60s and maintains a relative high level until the attack ceases at 1370s, at which point the WAR ratio falls back to 0 rapidly. A similar pattern of “climb-maintain-drop” in the WAR ratio can be observed on other data drives, albeit with variations in the start or end time. Figure 3 (b) and (c) illustrate the WAR ratio dynamics for the VM during attacks by Globeimposter and Babuk, respectively, revealing patterns similar to those depicted in Figure 3(a). Hence, we can conclude that cloud-based ransomware also exhibit a distinct pattern of WAR IO at the block level, which is consistent with insights from ransomware detection on hardware level.

To sum up, we can conclude that ransomware attacks at the block level exhibit distinct patterns that can be detected by analyzing the LBA access records.

Method	Precision	Recall	F1-score
SSD-insider++	63.05%	87.53%	73.26%
RanSAP	84.83%	94.37%	89.23%
WaybackVisor	71.36%	93.06%	80.66%
Combine Model	90.74%	92.10%	91.42%

Table 2: Comparison of four ransomware detection algorithms.

Multi-version nature of LSBD. Recall that one key feature of EBS is adopting the log-structured block device (LSBD) in the Block Proxy (See §2.1). In this setup, all writes from the front-end VDs are appended to the end of the log, and the Block Proxy maintains a mapping table—called IndexMap—to track the latest metadata of data. Periodically, Block Proxy reclaims the space with garbage collection (GC).

This design is similar to the SSD’s append-only internal architecture, which serves as one of the prerequisites for hardware-based ransomware protection. The knack here is that, in both EBS and SSD, the stale data are usually *not immediately* reclaimed by GC and overwritten with new data. Instead, it can survive for a certain period of time which essentially enables multiple versions of data to co-exist. As a result, in the face of ransomware attacks, we can leverage this multi-version nature to conveniently roll back to early versions by altering the IndexMap.

4.3 A Preliminary Exploration

Based on the above two opportunities, a potential design for ransomware protection in EBS arises. In short, we can actively monitor all incoming IO from each VD and check whether the LBA access matches the ransomware patterns. If detected, we can conveniently roll back the data to a previous clean version. The benefits of this design are that: (1) it only relies on block-level IO records, thereby being transparent to users and hardware independent; (2) it can leverage the multi-version nature of LSBD, hence no extra storage space is required.

In this prototype, there are mainly three components. (1) *IO Trace Collector*: Collect the block-level IO records from Block Proxy including the operation code (read, write or trim), LBA, and length; (2) *Ransomware Monitor*: Periodically analyzing the IO records of all VDs to determine if any are targeted by ransomware attacks. The monitor essentially functions as a feature extractor combined with a classifier. The former extracts handcrafted features from IO records, while the latter is a machine learning-based classifier; (3) *Data Recovery*: Upon detection of an attack, the data recovery mechanism is activated, restoring the data to a pre-attack state by changing the IndexMap.

To validate the effectiveness, we test this prototype on our benchmark, which includes around 150 hours of Block IO traces from 13 common ransomware and 16 types of real-world workloads (see §6.1). In addition, we have implemented four variations by adopting three state-of-the-art detection methods from previous work including SSD-insider++ [35], RanSAP [40] and WayBackVisor [42], and a combined one (i.e., all features included). Table 2 demonstrates the precision, recall, and F1-score results of the four methods. From the results, we can observe that all four left room for improvement and combining features certainly help (i.e., highest F1-score).

Such solutions still do not meet our goals as they incur data loss or leave tainted data unhandled. First, false negatives is unacceptable as the valid copies of data may be lost during later GC. Second, having a less satisfied precision (i.e., identifying normal activity as ransomware) is also consequential as users' normal IO can be wrongly interrupted and rolled back.

4.4 Challenges

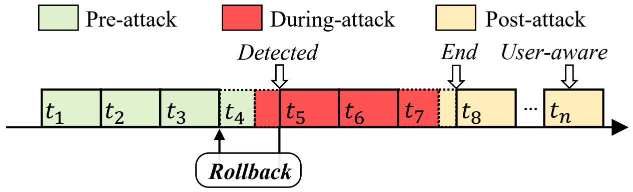


Figure 4: A typical process of ransomware attack, detection, and recovery.

Insufficient features. The straightforward design is based on the SSD and thus bounded by its limited on-chip resources (e.g., computational power and memory space). As a result, the feature extraction and classification algorithms become inherently straightforward (e.g., simple statistics of erasure IO in [35]), thereby yielding low accuracy. This also applies to the solutions based on kernel modules as they, too, need to be lightweight to avoid interfering with normal IO activities.

In EBS, the Block Proxies can have much more resources at hand. In addition, we can also set a standalone machine to run the detection module. Therefore, it is possible for our solution to include more features and adopt complex models for improvement. However, introducing features is helpful only when they reflect the unique characteristics of ransomware attacks, which requires careful analysis on the traces. Moreover, though EBS may have more computational power, spending too many resources on ransomware detection is still not economically acceptable due to the relatively low occurrence rate of ransomware attacks. In summary, we need to strike a balance between accuracy and overhead by identifying the unique characteristics and including them for feature construction.

Impermanent multi-version support vs. data loss. Even if we can achieve a high-accuracy model through the above endeavors, it is unlikely such a solution can always detect the exact timing of the ransomware attacks with no false positives/negatives. Hence, we may still run into data loss when rolling back is wrongly invoked.

Here, we use Figure 4 to illustrate a rundown on typical cases that might lead to data loss. In the figure, each rectangle represents a sample, and the background color indicates the three phases of the attack: pre-, during-, and post-attack. Assume that ransomware initiates an attack at the t_4 window (as indicated by the dashed line) and continues until t_7 window (also indicated by the dashed line). The detection model identifies the attack at t_5 and rolls back the data to t_4.

First, if the proposed solution fails to detect the attack at t_5 (i.e., false negatives), the user's data may become irretrievable if garbage collection (GC) has already kicked in. Note that data recovery can only issued by users but users may not be aware of the attack in time. Second, if the detection model wrongly labels normal IO as ransomware attack (i.e., false positives), it might cause user panic and even lead to incorrect data rollback. Third, even if the ransomware attack is alarmed, the user fails to stop normal IO until t_n which might be several hours after attack, the rollback-based recovery would result in greater data loss that all the user's normal IO during- and post-attack would be reverted.

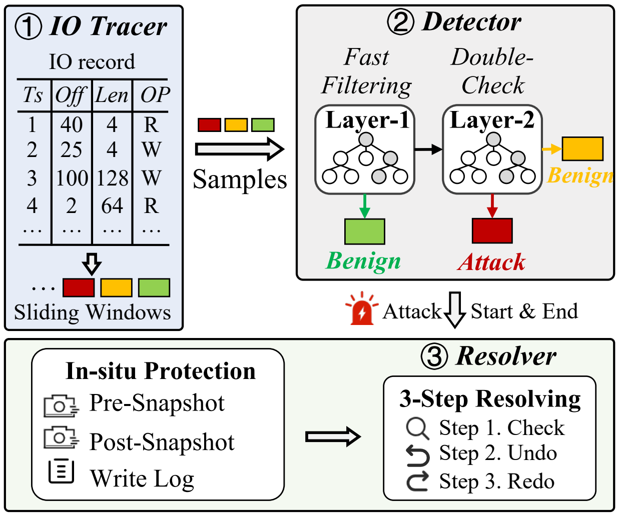
To solve above issues, the easiest way is to provide the multi-version support for all data modifications permanently. However, this is not feasible as it would quickly deplete the user's purchased disk space within months or even days due to the constant accumulation of data. The same reason also applies to vendors. Therefore, we need to find a solution that can efficiently persist all IO records from right before a ransomware attack until the end of it.

5 `DeftPunk` Design

We now introduce the design of `DeftPunk`, a practical ransomware detection and data recovery framework for cloud EBS. `DeftPunk` does not require users' cooperation, designated software support or customization on hardware in detection, and can guarantee no data loss during the recovery. The key to `DeftPunk` success is employing a two-layer machine learning model to efficiently detect attacks and create snapshots. Then, `DeftPunk` can notify the user and perform the subsequent data recovery. In this section, §5.1 presents an overview of the framework and workflow of `DeftPunk`, followed by a detailed discussion on the three main components: the feature engineering (§5.2), two-layer classifier (§5.3) and data recovery (§5.4).

5.1 Overview

Data preprocessing. Figure 5 demonstrates the high-level procedures and interactions between components in `DeftPunk`. First, the IO Tracer, embedded in the Block Proxies (BP), constantly monitors all incoming IO record (i.e., formatted as `<timestamp, offset, length, operation>`) and group them as IO records by VDs. Then, IO tracer generates samples with a sliding window of 10 seconds. For example, assume a BP that serves three VDs. For a minute



Ts	Off	Len	OP
1	40	4	R
2	25	4	W
3	100	128	W
4	2	64	R
…	…	…	…

Figure 5: Overview of `DeftPunk`.

of monitoring, IO Tracer would generate 6 samples of IO records for each VD.

Ransomware detection. Upon receiving the IO record samples, `DeftPunk` first uses the *Layer-1 Classifier*, based on simple features (e.g., IO count) and a decision tree, to perform fast scanning with an emphasis on high recall and low computational cost. If the sample fails to pass the first layer (i.e., suspicious of a ransom attack), it will be further checked by the *Layer-2 Classifier*. The second model uses a superset of Layer-1's features by including more complex ones, (e.g., statistics of entropy, Working Set Size, and IO offset), and employs a more sophisticated model (i.e., XGBoost), aiming at high precision. If the sample is again labeled as positive, `DeftPunk` would trigger snapshot generations.

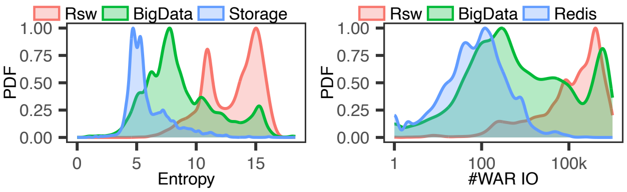
Creating snapshots. When a positive sample arrives, the snapshot worker would first create a Pre-attack Snapshot and a Post-attack Snapshot at the beginning and the end of it, respectively. As a ransom attack may span across multiple 10-second samples, the snapshot worker continues to create new Post-attack Snapshot to replace the previous one until the incoming sample becomes negative. Meanwhile, during the attack, `DeftPunk` would also pause the EBS garbage collection for this VD to prevent the accidental deletion of the valid data until the post-attack snapshot is finalized.

Data recovery. When the user has been notified of the attack (e.g., via our alert message or discovering certain data become inaccessible), one needs to issue a recovery request. `DeftPunk` would follow a three-step recovery process. (i) *checking*: `DeftPunk` first checks and lists all tainted LBAs (i.e., 4KB long each) during the attack. If a LBA is further modified (by the user) after the post-attack snapshot, then data in that LBA would not be recovered. (ii) *undo*: `DeftPunk` would revert all data in tainted LBAs to the version of the pre-attack snapshot. *redo*: `DeftPunk` would run another rule-based model to identify IOs made by the users and redo them.

5.2 Feature Engineering

In §4.4, we mentioned that one key reason for suboptimal performance of existing solutions is the insufficient feature engineering. To construct features for `DeftPunk`, we first conduct an extensive study on real-world VDs' IO traces to identify the fundamental differences between ransomware and normal user behaviors (§5.2.1). Based on the insights, we then devise the extended features for the two-layer model.

5.2.1 Characterizing IO Behavior



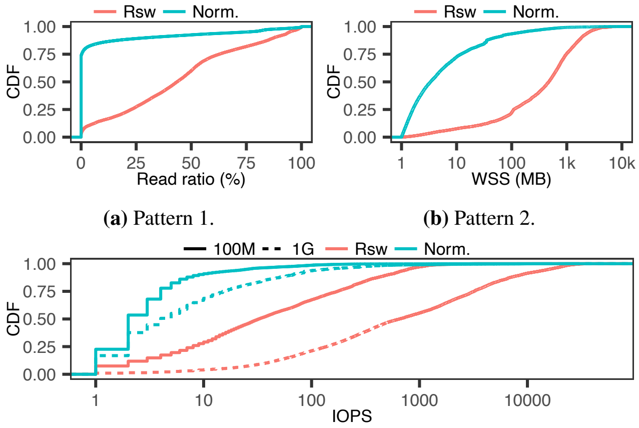
(a) Offset entropy **(b)** WAR IO count

Figure 6: Opportunities of detecting ransomware at block-level.

We first reason why typical features used in existing works, such as *offset entropy* in [40,42] and *WAR IO count* in [34,35], can be ineffective. In Figure 6(a), we present the probability density function (PDF) of the offset entropy for samples in our dataset. "Rsw" indicates the PDF of all ransomware samples. "BigData" and "Storage" represent two different types of common workloads. We can observe a distinct difference in the PDF of offset entropy between Rsw and Storage, while there exists large PDF overlap between BigData and Rsw. Figure 6 (b) exhibits a similar pattern on Bigdata and Rsw.

This comparison clearly shows that normal workloads can be easily mislabeled (or the other way around) if we solely rely on a selected few simple features. We believe that one major root cause is that previous work mostly focus on positive samples (i.e., ransom attack IOs) without noticing or studying the similarities they share with normal traffic. Hence, we study both the patterns of ransomware and normal I/O behaviors and propose three patterns for better feature engineering.

- **Pattern 1.** Ransomware typically exhibits nearly equal amounts of read and write in bytes, whereas the read-to-write ratio of normal IOs is often skewed. Figure 7(a) presents the cumulative probability density (CDF) of the read-ratio (i.e., the proportion of read requests to total requests). We can see a notable gap between the CDF of normal workloads (Norm.) and ransomware (Rsw), with the average value for Norm. being X and the average value for Rsw being Y.
- **Pattern 2.** The IO offsets of ransomware are distributed more broadly across the LBA space, and each offset is typically accessed only once. In contrast, the IO offsets under



(**a**) Pattern 1.

(**b**) Pattern 2.

(**c**) Pattern 3.

Figure 7: Three unique patterns that can distinguish between ransomware and normal workloads.

normal workloads tend to be concentrated. Figure 7(b) shows the CDF of the working set size (WSS) for the two categories of samples, and we can similarly observe a significant difference between Norm. and Rsw.

- **Pattern 3.** Ransomware displays obvious WAR on the system disk, particularly in the Master Boot Record (MBR) region at the beginning of the LBA, whereas such operations are rarely performed by normal users. Figure 7(c) displays the IOPS (Input/Output Operations Per Second) for the two categories of samples within the first 100MB and the first 1GB of the LBA.

5.2.2 `DeftPunk` Feature Engineering

Based on the findings above, we construct the following as `DeftPunk`'s enhanced feature engineering for ransomware detection. Table 3 presents the 43 features we employed along with their corresponding meanings. For ease of understanding, we categorize these features into 5 classes, which include:

- **IO Dependency**. This set of features characterizes the behavior of adjacent I/O operations in a temporal context. In addition to including Write-After-Read (WAR), which has been adopted by other works [34, 35], we also incorporate three other types of read-write sequences, such as Read-After-Write (RAW), Read-After-Read (RAW), and Write-After-Write (WAW).
- **IO Statistics.** This part constitutes the basic statistics of VD IO, including the IO count, total bytes, and IO size. Here, features are made separately for read (R), write (W), and the sum of read and write (RW). The design of this feature set is inspired by **Pattern 1.**
- **Working Set Size (WSS)**. Inspired by **Pattern 2**, we track the working set size (WSS) for 6 types of I/O. Here, WSS refers to the proportion of the LBA space that is accessed by a specific type of IO.
- **Offset Statistics**. Compared to WSS, this category of features characterizes the distribution of I/O across the LBA at a finer spatial granularity, such as the mean, variance, standard deviation, coefficient of variation (CoV) [32], and entropy [56] of the IO offset. The design of this type of feature is also inspired by **Pattern 2**.
- **Access on LBA Head Region**. As mentioned in **Pattern 3**, ransomware may tamper with data in the head region of the LBA. Bearing this in mind, we conduct statistics on IO that fall within the first 100MB and 1GB of the LBA, including the I/O count and total bytes.

In Table 3, we also list the computational complexity (in Big O notation) for each class. Note that the traditional definition of Write-After-Read (WAR) is to check whether there are sequential read and write requests accessing the same LBA offset. However, we extend the definition of WAR by relying on both the offset and the length to decide whether the LBA accessed by the sequential read and write requests are overlapped. Therefore, our check for WAR requires a time complexity of O(logn) rather than O(1). The reason to do this is that we find that ransomware can read a large volume of data and then writes it in smaller chunks, exhibiting the behavior of "read-write-...-write". According to the traditional definition of WAR, this behavior would be characterized as 1 WAR and 2 WAWs, which can be similar to normal user IO. However, our definition of WAR would describe this behavior as 3 WARs and 2 WAWs, thereby distinguishing it from the I/O behavior of normal users.

5.3 Two-layer Model

The enhanced feature engineering can improve the precision/recall of the detection. However, simply building a classifier based on the entire set of features would not be practical for production deployment. This is because certain features such as IO dependency, offset entropy, and WSS would have high computational complexity and thus yield a high overhead in detection. For example, given a VD with 1 million IO records, the processing time for IO dependency can be 14 seconds for one thread. In this case, blindly applying the entire set of features would consume nearly 56 virtual CPU cores for 1 million IOPS online only for feature calculation.

Therefore we propose a two-layer model to balance the trade-off between overhead and detection accuracy. First, we use a simple feature set in the first-layer model for fast filtering the majority of non-ransom activities. We apply the complex feature set with second classifier to scrutinize the suspected cases labeled by the first model. Specifically, the two-layer classifier is implemented as follows.

Layer-1: Fast and Broad Filtering. This layer aims to achieve a high recall and rapid initial scanning of ransomware using features with O(1) complexity. Hence, we evaluate the detection performance of three commonly-used simple classifiers, including k-Nearest Neighbors (kNN) [39], Logistic Regression (LR) [50], and Decision Trees (DT) [51], with

Type	Description	Complexity	#Features
IO Dependency on Block	**IO count / Bytes** of (RAW / WAR / RAR / WAW) IO	O(logn)	8
IO Statistics	**IO count / Bytes / Size / Bps** of (R / W / RW) IO	O(1)	11
Working Set Size (WSS)	**WSS** of (R / W / RAW / WAR / RAR / WAW) IO	O(logn)	6
IO Offset Statistics	**Var / CoV** of (RW) IO	O(1)	2
	Entropy of (R / W / RAW / WAR / RAR / WAW) IO	O(logn)	6
Access on LBA Head Region	**IO count / Bytes** on first (100M / 1G) of (R / W) IO	O(1)	8

Table 3: Feature engineering of `DeftPunk`. *In the second column "Description", the bold text separated by slashes ("/") represents different metrics, while the parts within parentheses separated by slashes represent different IO types. For example, the meaning of the first row is to calculate the IO count and Bytes for each of the four IO types: RAW (Read-After-Write), WAR, RAR, and WAW. Therefore, there are a total of* $2 \times 4 = 8$ *features. R, W and RW IO refers to read, write, and read-write, respectively..* **Var**: *variance*; **WSS**: *working set size*; **CoV**: *coefficient of variance*.

Layer	Model	Precsion	Recall	F1-score
Layer-1	KNN	73.1	82.5	77.5
	LR	32.7	91.1	48.1
	DT	**87.5**	**95.9**	**91.5**
Layer-2	RF	96.2	97.6	96.9
	lightGBM	95.1	97.1	96.1
	CatBoost	95.4	98.5	96.9
	XGBoost	**95.8**	**98.6**	**97.1**

Table 4: Two-tier model selection. **KNN**: *k-nearest neighbors*; **LR**: *logistic regression*; **DT**: *decision tree*; **RF**: *random forest*.

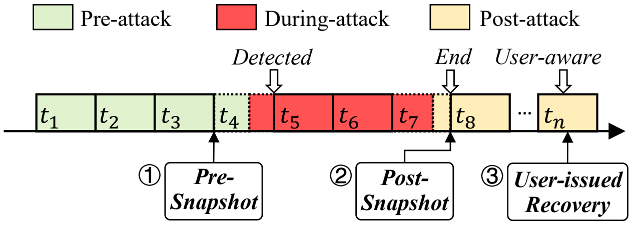


Figure 8: `DeftPunk`'s timeline for lossless data recovery.

the results depicted in Table 4. We choose the Decision Tree (DT) as the classifier for layer-1 as it achieves the highest recall (95.9%).

Layer-2: Accurate Double Check. Suspected cases from layer-1 are passed to the layer-2, which employs computationally expensive features that can better distinguish between ransomware and normal behavior. Here, we add all the features mentioned in Table 3 to the model, and similar to layer-1, we test the performance of four commonly-used complex classifiers, including Random Forest (RF) [36], LightGBM [44], CatBoost [52], and XGBoost [38], with the results presented in Table 4. The results indicate that XGBoost achieves the highest F1-score (97.1%) and thus is chosen as our layer-2 classifier. Our further experiments in §6.6 demonstrate that the two-layer model maintains the same detection performance as single-layer model but with much less computation needed.

5.4 Creating Snapshots

Once an attack is detected, `DeftPunk` generates a pair of snapshots, called pre-/post-attack snapshots. The goal is to "lock in" the effects of the ransomware to make sure all data modification during the period are recorded and recoverable. Then, inspired by the "undo-redo" mechanism in database, `DeftPunk` reverts all the LBAs modified by the ransomware. We also employ another rule-based model to identify the LBAs modified by the user and redo them.

Creating pre-attack snapshot. Once notified by the two-layer classifier, the snapshot worker checkpoints the IndexMap and data of the VD at the time of beginning of the labeled batch window. This is enabled by the multi-version nature of LSBD, which allows the snapshot worker work backwards as long as the previous changes have not been garbage collected. This is guaranteed by setting up the length of a sample to be 1 minute and configuring the GC to only collect stale data that are at least older than 30 minutes. Meanwhile, we also pause the GC on this VD.

Creating post-attack snapshot. Generating the post-attack snapshot is different. As a ransomware attack might be longer than a single sample, `DeftPunk` waits until the two-layer classifier labels an incoming as negative. Then, the snapshot worker creates a post-attack snapshot at the end of last labeled sample and resume GC. In addition, we also record all the writes, including data and LBA, between two snapshots as a write log. Note that the writes between pre-snapshot and detection time can be obtained due to the multi-version support. Plus, the modifications after the detection time can be recorded as we have paused the GC. Finally, we compare the IndexMaps between the two snapshots to only keep data pre-snapshot that are modified during the attack and drop others for space efficiency.

5.5 Data Recovery

To this end, we have acquired three pieces of data, including a pre-attack snapshot (i.e., data, LBA and versions at t_3 in Figure 8), a post-attack snapshot (i.e., LBA and versions at t_8), and a write log (i.e., data and LBA between t_1 and t_5). Now, `DeftPunk` follows three steps to recover the data.

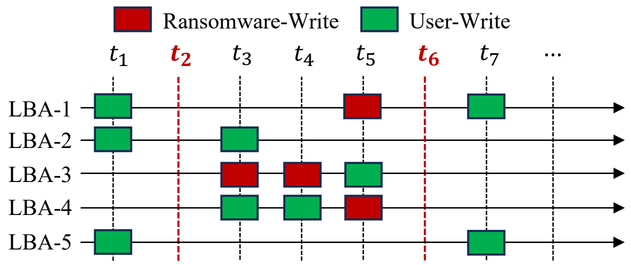


Figure 9: Five cases of LBA data recovery by `DeftPunk`.

Identifying the tainted LBAs. Once the user has been notified of the ransomware attack, they can initiate the request for data recovery. `DeftPunk` would first check if the LBAs listed in the post-attack snapshot are modified by the users after the attack. `DeftPunk` would not try to recover data from such LBAs as they are deemed valid by the users (i.e., LBA-1 and LBA-5 in Figure 9).

Undo. Executing undo is straightforward. `DeftPunk` would revert all involved LBAs (i.e., only modified during the attack but not after) to the version of the pre-attack snapshot.

Redo. Ransomware usually skips files that are already opened (e.g., LBA-2) for writes or kills the user's writing process (e.g., LBA-1) to ensure data integrity of the encrypted files. Still, it is possible that the user's IO exists in the tainted LBAs such as LBA-3 (i.e., user overwrites encrypted LBA block) and LBA-4 (i.e., user's writing is not immediately stopped).

To avoid data loss, `DeftPunk` needs to redo these users' IOs. First, `DeftPunk` employs a rule-based model, which checks the in-place write-after-read (WAR) patterns, to determine whether an IO is indeed modified by the ransomware. Note that, in this case, simply checking the WAR pattern can effectively single out the ransomware IOs because these LBAs have already been filtered by the two-layer model. In other words, the chance of a user's IO coincidentally matching the ransomware's IO pattern is extremely low. We further discuss how to handle such a corner case in §5.6. Then, `DeftPunk` would drop the tainted IOs and apply the normal ones in order.

5.6 Corner cases.

Missing the start/end. The ransomware might start or end right around the 10-second sliding window splitting. Hence, a few IOs made by the ransomware might evade the two-layer model checking and taint the users data. In this case, we further include IOs from one more sample before and after the labeled sample to the write log to avoid data loss.

Mislabeled writes. During the attack, the users might coincidentally write to the LBAs with the similar pattern as the ransomware, e.g., encrypting or rewrite the files. Consequently, the rule-based model may incorrectly mark the behaviors as ransomware and revert them. Note that experiencing this type of mislabeling is rather unlikely in practice. This is because all known ransomware would choose to terminate the user's writing process targeting the same LBAs or simply avoid these addresses (i.e., files) to ensure the integrity of ransomed data. So far, we have not observed any such cases in the field.

Nevertheless, we employ a manual checking process if the user is unsatisfied with the automatic undo and redo. We provide the users with a tool to associate files to LBAs. Then, for the files (and their corresponding LBAs) that are mistakenly reverted, the users can choose to drop/apply the writes on an individual basis. Note that simply using this tool and asking the users to manually check all the LBAs would be impractical given the sheer volume of IOs during runtime.

Data inconsistency. `DeftPunk` is a block-level solution and hence does not provide file/application-level consistency guarantees. It is possible that recovery process can lead to data inconsistency. For example, applying a user's overwrites on the encrypted data may result in a corrupted file (i.e., LBA-3). To resolve this, restoring to the pre-attack snapshot guarantees a clean start. In addition, users can again use our manual checking tool for a finer control of what IOs to apply or drop on tainted LBAs.

Multiple attacks. As the data recovery is only triggered by the user after the notification, it is rare but still possible that the same LBAs have been attacked multiple times. Now, we further discuss the case of two consecutive attacks. The analysis and solution shall apply to the case of multiple attacks (e.g., three or more). Depending on the distance between the two attacks, we can have the following cases:

- *One pair of snapshots.* When the the second attack closely follows the first one (i.e., within 2 samples), `DeftPunk` would treat the two as a single attack as the post-attack snapshot would include an extra sample (see corner case 1). Then, `DeftPunk` performs data recovery as usual. If there is a user's IO between the two attacks, `DeftPunk` can use the rule-base model to check and only redo user's IOs.
- *Two pairs of snapshots.* If the two attacks are far apart (e.g., one day away), `DeftPunk` creates a pair of snapshots for each attack. Then, `DeftPunk` performs data recovery on each attack separately in time order. In this case, the user's IOs between the two attacks would not be affected as the two are treated as independent attacks.

5.7 Runtime Modes

To avoid mislabeling, `DeftPunk` runs in a per-batch mode in the field. Specifically, a batch is 10-minute long and thus includes 60 10-second records. `DeftPunk` labels a batch as positive as long as one sample has been flagged.

Further, users can enable/disable Deftpunk on a Viritual Disk (VD) basis. For example, if a user has mounted multiple VDs, the user can choose to only protect important VDs (e.g., data drives) with Deftpunk but not ones for caching. In addition, we initially allowed system administrators to use only a part of the feature set or just Layer-1 of the two-

App.	#Batch	#IO($1e5$)	App.	#Batch	#IO($1e5$)
Prometheus	980	893.5	MongoDB	128	197.2
Storage	938	277.6	Postgress	100	48.4
MsSQL	921	459.2	Oracle	77	112.7
MySQL	900	949.6	RabbitMQ	55	82.1
WebAPP	739	159.5	ElasticSearch	40	32.2
Redis	435	70.1	etcd	29	7.4
MessageBus	370	90.6	influxDB	6	0.9
BigData	220	749.0	Kafka	5	18.1

Table 5: Negative samples in `DeftPunk` benchmark.

Config	Content	#Types
Ransom family	loki [22], BeijingCrypt [19], makop [20] Sodinokibi [25], babuk [12], VoidCrypt [15] phobos [8], GlobeImposter [16] wannacry [30], mallox [18]	13
OS version	WinServer 64-bit 2016, 2019, 2022 (w/ and w/o container), CentOS	6
App.	Copying, Massive Write, Massive Read, ZIP CRYPT, MySQL	5

Table 6: Configurations of simulated ransomware attack.

layer model to achieve lightweight detection. However, we later found that such a trade-off is not efficient as it leads to a significant drop in detection performance with minor reduction in overhead. We present an experiment on this in §6.6. We recommend that users enable the full feature set and adopt the complete two-layer model.

6 Evaluation

Our evaluations intend to answer the following questions:

- What is the composition of the benchmark? (§6.1)
- How does `DeftPunk` perform against other methods on a per-sample basis? (§6.2)
- How does `DeftPunk` perform in zero-shot scenario? (§6.3)
- How does `DeftPunk` perform in per-batch setup? (§6.4)
- Does the feature engineering work? (§6.5)
- What is `DeftPunk`'s runtime overhead? (§6.6)
- How is `DeftPunk` in deployment? (§6.7)

6.1 Ransomware Benchmark

The dataset is composed of two parts:

Negative samples (normal IO). Benign samples consist of I/O records from EBS virtual disks (VDs) running real-world workloads from the field. Table 5 shows the types of workloads, the number of batches, and the volume of IO records. Note that each batch refers a group of IO records of a VD within a 10-minute span. Each IO record is quadruple, formatted as `<timestamp, offset, length, operation>`. To generate negative samples, we reuse the sliding window mechanism. Therefore, a 10-minute batch can generate 55 1-minute long negative samples. In total, we have gathered nearly 2 million samples.

Postive samples (ransomed IO). We generate positive samples by simulating ransomware attacks atop the normal IOs. Specifically, we follow a "ransomware-OS-application" configuration to generate batches. For each configuration, as shown in Table 6 we mix and match the ransomware families (1 in 13), operating systems (1 in 6), and background applications (1 in 5). Each generated batch is around 10 minutes long and we use the same sliding window methodology to generate positive samples. Note that a ransomware attack can be shorter than 10 minutes. Therefore, we also discard samples that do not have any ransom activities (i.e., samples before/after the ransomware in the generated batch). In total, we have 52 thousands positive samples.

Metric. Ransomware detection is a typical binary classifier, determining whether a sample is positive (i.e., ransomed) or not. Therefore, we use *precision* (i.e., the proportion of positive identifications that are actually correct), *recall* (i.e., the proportion of actual positives that are correctly identified), and the *F1-score* (i.e., the harmonic mean of precision and recall) for measurement.

6.2 Per-Sample Test

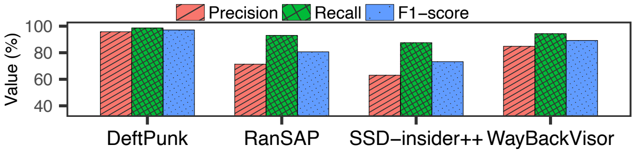


Figure 10: Overall performance of `DeftPunk`.

We compare `DeftPunk` with three other ransomware detection methods including SSD-insider++ [35], WaybackVisor [42] and RanSAP [40]. We split the dataset as 90% for training and 10% for testing. We use 10-fold cross-validation which means we train and test the models 10 times and each time with a different 10% as test set.

We calculate the average recall, precision, and F1-score of each candidate from the 10 tests. In Figure 10, we can observe that `DeftPunk` clearly outperforms all others with a 98.6% in recall, 95.8% in precision, and 97.1% in F1-score. This validates the effectiveness of `DeftPunk`'s feature engineering and two-layer model.

We further analyze the false positive/negative cases. First, it is possible `DeftPunk` incorrectly flags normal IOs as ransomware activities (i.e., precision is not 100%). Normally this is because users' IOs exhibit the same write-after-read patterns as ransomware activities, such as data encryption, in-place compression, and format conversion (e.g., changing a mp4 video file to a mkv one). `DeftPunk` alleviates this issue by leveraging multiple patterns instead of just write-and-read for detection. Still such coincidences may occur and lead to the mislabeling. Note that having a few false positives is acceptable. Recall that DeftPunk follows a detect-notify-rollback process. Therefore, if DeftPunk mislabels normal activities, the notified users can just ignore the alerts, and inform us to delete these snapshots. No rollback would be executed unless the users confirm an attack has occurred.

Second, it is possible DeftPunk misses some ransomware activities (i.e., recall is not 100%). In the per-sample test, the main reason is that the ransomware attack, which usually lasts several minutes, can span multiple samples. Certain samples (e.g., the very first or last one) may not contain enough ransomware activities to be successfully identified, thereby lowering the recall. Note that, in practice, DeftPunk is deployed in a per-batch manner (i.e., 10-minute window), where the recall is near 100% (see §6.4).

6.3 Zero-shot Detection Performance

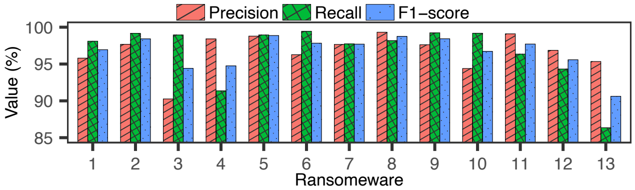


Figure 11: Zero-shot performance of DeftPunk.

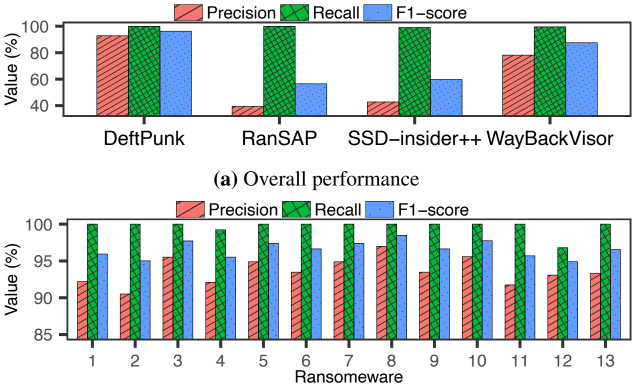
Given that ransomware can be fast evolving, we further measure DeftPunk's detection performance on "unseen" ransomware. Each time, we remove one type of the 13 ransomware families from the training dataset but still keep samples from that family in the testing set (i.e., zero-shot).

Figure 11 displays the results where each group of three bars show the performance of a specific ransomware family were removed from the training set. We can see that DeftPunk is capable of effectively detecting most unseen ransomware, maintaining at least 95% recall and 90% precision. This indicates that our feature engineering captures the common characteristics of ransomware.

The only exception is Wannacry. Further analysis suggests that Wannacry is intentionally slowing down its operation (e.g., CPU utilization is below 25%, and disk throughput is below 10%) to evade detection. Similar to the aforementioned discussion on false negatives, the lower recall is caused by the scant amount of ransomware activities in some samples due to the intentionally *diluted* IOs. Note that such an evasion strategy—while useful in the per-sample test—is not effective in our deployment scenario (i.e., per-batch experiment, see §6.4) for more details.

6.4 Per-Batch Experiment

Note that in the previous two experiments we calculate the performance on a per-sample basis. While they validate the overall designs of DeftPunk in a microbenchmark fashion, DeftPunk in practice is deployed on a per-batch checking basis (see §5.4). In other words, as long as one sample within a 10-minute batch is marked as positive, the entire batch is marked as positive. Note that per-batch testing is closer to real-world setup due to the streaming of IOs.



(a) Overall performance

(b) Zero-shot performance

Figure 12: Per-batch performance of DeftPunk.

In Figure 12, we rerun the experiments in §6.2 and §6.3 but with a per-batch basis. We can see that DeftPunk can always achieve near 100% recall across all situations with a minor (around 2% on average) decrease in precision. Surprisingly, with per-batch detection, DeftPunk can even successfully identify Wannacry ransomware in zero-shot experiment which intentionally dilutes its IO for evasion. On stark contrast, other methods while also have an increasing recall, can experience considerable precision loss, yielding low practicality (i.e., frequently issuing false alarms).

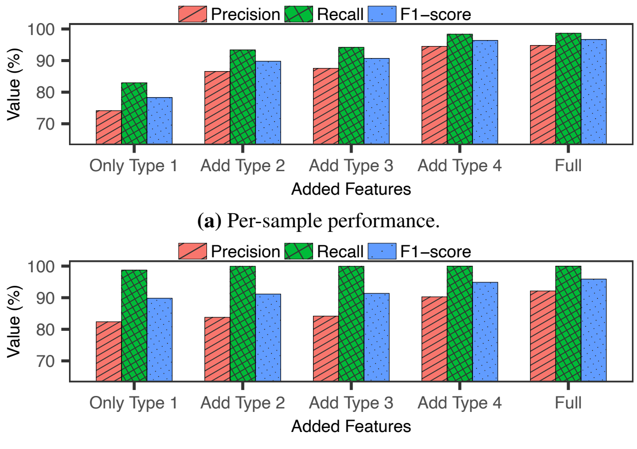
Obviously, compared to experiments in §6.2 and §6.3, the precision and recall in per-batch experiments are much higher. This is because the per-batch test reduces the chances of mislabeling by taking multiple samples (i.e., 1 batch = 55 samples) into consideration. Nevertheless, DeftPunk may still fail to identify certain ransomware activities. One representative case is the Babuk ransomware (i.e., the 12th in Figure 12(b) being undetected in a VD with mostly text files. Further analysis reveals that Babuk is designed to not encrypt text files, thus being latent (i.e., not encrypting files) and avoiding the detection.)

6.5 Effectiveness of Features

Now, we measure the effectiveness of DeftPunk's feature engineering. Specifically, from only one type of features, we add on another set of features and evaluate the overall performance of DeftPunk on both per-sample and per-batch basis. In Figure 13, we can see that DeftPunk's performance steadily increases with the addition of more features. This validates the effectiveness of DeftPunk's feature engineering.

6.6 Runtime Overhead

Monitoring and preprocessing in DeftPunk incur negligible overhead as the IO Tracer only checks and packs the quadru-



(a) Per-sample performance.

(b) Per-batch performance.

Figure 13: Ablation study of `DeftPunk`'s feature engineering. *In the figure, the Type 1 features corresponds to all features associated with the 1st row (IO Dependency on Block) in Table 3, with Type 2 through Type 4 following in a similar fashion.*

Model	Precision	Recall	F1-score	Time (s)
Layer-1 Only	87.3%	95.9%	91.5%	**3.72**
Layer-2 Only	94.9%	98.6%	96.7%	**56**
DeftPunk	95.8%	98.6%	97.1%	**4.85**

Table 7: Comparison of `DeftPunk` v.s. Layer-1/2 only.

ple metadata in an asynchronous fashion (i.e., non-blocking).

For detection, we evaluate the speedup made by the two-layer model by comparing it against the Layer-2 only model. Table 7 shows that, on processing 1 million IOs with one thread, the two-layer model only takes 4.85 seconds, yielding a 11.5× speedup over the Layer-2 only model. Including the overhead for detection, as well as data processing and transfer, we are able to process data at 140,000 IOPS using a 2.7GHz vCPU, which means that on processing 1 million IOs per second we need around 7 vCPUs.

We can enforce `DeftPunk` to run with only Layer-1 enabled for lower overhead. In Table 7, we can see that, with only first layer, `DeftPunk` can speed up 24.3% (i.e., from 4.85 to 3.72 seconds). However, both precision and recall decrease significantly (e.g., from 95.8% to 87.3% in precision). By weighing the trade-off between performance and speedup, we believe the two-layer model is the best choice for `DeftPunk`.

For recovery, `DeftPunk` introduces additional storage space because of the snapshots and write logs. For a typical 100GB VD under ransom attack, our experiments show that the storage overhead is around 150MB on average, much less than the periodical snapshots.

6.7 Deployment

Currently, we have deployed `DeftPunk` in our EBS service for limited internal users. For each internal EBS cluster (hosting more than 30K VDs), we deploy one machine (8 virtual CPU(2.7 GHz), 32 GiB Memory) to run `DeftPunk`'s detection and recovery components. `DeftPunk` has already successfully prevented two attacks with data fully recovered within 240 minutes after internal users issued reports. We expect to release `DeftPunk` for public review in the near future.

7 Potential Limitations

7.1 EBS-specific Solution

The success to `DeftPunk` is based on two properties of ALIBABA EBS, the log-structured design and block-level IO support. However, this does not mean `DeftPunk` can only be applied to ALIBABA EBS. First, the log-structured design is widely adopted in many cloud storage systems (e.g., HDFS [55]) and file systems (e.g., F2FS [47]). Practitioners can also leverage the multi-version support, snapshots and GC pausing to secure a full-copy of data modifications during attack. Second, block-level IO monitoring and analysis can be achievable for cloud vendors and system administrators. Moreover, with our open dataset and users' own traces, they can also train their own models to detect ransomware activities following our feature engineering practice.

7.2 Effectiveness on Unknown Ransomware

In this paper, `DeftPunk` shows its effectiveness on 13 types of well-known ransomware. There can be others that are not covered or even still under development. However, we believe this does not pose a great threat to `DeftPunk`'s validity. First, the zero-shot experiments show that the features of `DeftPunk` can generalize well to unseen types of ransomware. Second, `DeftPunk` can be quickly adapted to new ransomware by introducing new features and deployed to production systems as it is a pure software solution.

7.3 Threats of Mimicry Attacks

The features of `DeftPunk` arise from the three ransomware IO characteristics (i.e., Pattern 1-3 in §5.2.1). First, to encrypt, ransomware exhibit unique in-place write-after-read LBA patterns. Second, to quickly attack multiple files, ransomware would touch many files and encrypt a small proportion in each of them, yielding a wide range of sporadic LBA accessing. Third, to increase the impacts, ransomware would especially favor important areas (e.g., head region).

It is possible that future attackers may choose to evade `DeftPunk` detection by avoiding the above behaviors in their ransomware. However, we believe such efforts can be ineffective or even against the interests of ransomware (i.e., becoming less stealthy or unprofitable). First, using different access patterns (e.g., Wannacry slowdown) can still be caught by `DeftPunk` even under zero-shot setup (see §6.4). Moreover, further slowing down the IOs can backfire as it takes much longer time to finish file encryptions and can easily alert the users. Additionally, countermeasures for this slowdown attack in `DeftPunk` is straightforward—condensing the workloads (e.g., include a set of features to treat 10-minute workloads as 1-minute ones).

Second, we have seen "pseudo-ransomware" (e.g., wiperware [1]) choose not to encrypt but to directly destroy data (e.g., filling zeroes), thereby not showing write-after-read patterns (only writes, no read). But, they are no longer active as few victims would pay the ransom and current ransomware usually allows victims to decrypt a small proportion of data to show validity. In addition, we notice certain early ransomware (e.g., Gpcode.ak) do not follow in-place write-after-read but to delete original files and create new ones with encryption. However, they, too, soon have died out due to showing unusual patterns (i.e., frequent file deletion) and excessive IO traffic (creating a mass amount of files), which can be easily singled out and restored (e.g., PhotoRec for Gpcode.ak [23]).

Third, we can see that even if ransomware avoid Pattern 2 and 3 (i.e., "Only Type 1" in Figure 13), `DeftPunk` still shows high recall (98.8%). But, by doing this, ransomware would impact less files and/or target less important ones, thus being unattractive to the attackers.

In other words, Patterns 1-3 persist across all 13 families of ransomware are as a result of such patterns reflecting the fundamental nature of ransomware, especially after generations of evolutions. Admittedly, future attacks may evade detection in a different fashion or combine multiple strategies together. For example, ransomware can encrypt a selected set of important files with a more “diluted” pattern to minimize footprint.

To sum up, two lessons we have drawn from the above analysis are: (1) Closely monitoring the latest development. Propagation of new ransomware, while fast, still takes time to spread, which renders a window for us to analyze. (2) Building adaptive solutions. The security "arms race" is often inevitable and never-ending. One important advantage of `DeftPunk` is software-based and thus offers high flexibility, which further enables us to quickly adapt to emerging threats.

8 Conclusion

In this paper, we revisit a pressing problem in data security: defending against ransomware attacks. With a large-scale study on the IO characteristics of ransomware, we identify a rich set of features and leverages the log-structured multi-version properties to build `DeftPunk`, a block-level ransomware detection and recovery system. Our extensive evaluation shows that `DeftPunk` can achieve nearly 100% recall with 95.8% precision with minor overhead.

Acknowledgments

The authors would like to thank our shepherd Prof. George Amvrosiadis and anonymous reviewers for their meticulous reviews. We are also grateful for the support from the EBS team, and feedbacks by Amber Bi, Keely Xu, and Qingke X.F. on early drafts of this paper. This research was partly supported by Alibaba ARF/AIR program and NSFC(62102424).

References

[1] NotPetya Technical Analysis by LogRhythm Labs. https://gallery.logrhythm.com/threat-intelligence-reports/notpetya-technical-analysis-logrhythm-labs-threat-intelligence-report.pdf, 2017.

[2] 2022 Cloud (In)Security Report. https://www.zscaler.com/blogs/security-research/2022-cloud-security-report, 2022.

[3] AWS Backup Anomaly Detection for Amazon EBS Volumes. https://aws.amazon.com/cn/blogs/storage/aws-backup-anomaly-detection-for-amazon-ebs-volumes/, 2022.

[4] Storage Anti-Ransom Solution. https://e.huawei.com/cn/solutions/storage/oceanprotect/ransomware, 2022.

[5] 2023 Ransomware Trends Report. https://www.veeam.com/ransomware-trends-report-2023, 2023.

[6] 2023 State of the Cloud Report. https://info.flexera.com/CM-REPORT-State-of-the-Cloud?lead_source=Website%20Visitor&id=Flexera.com-PR, 2023.

[7] 2023 ThreatLabz State of Ransomware. https://info.zscaler.com/resources-industry-reports-2023-threatlabz-ransomware-report-old, 2023.

[8] A deep dive into Phobos ransomware. https://www.malwarebytes.com/blog/news/2019/07/a-deep-dive-into-phobos-ransomware, 2023.

[9] AliCloud - Elastic Block Storage. https://www.alibabacloud.com/zh/product/disk, 2023.

[10] Amazon Elastic Block Store. https://aws.amazon.com/cn/ebs, 2023.

[11] Azure Disk Storage. https://azure.microsoft.com/zh-cn/products/storage/disks, 2023.

[12] Babuk Ransomware: In-Depth Analysis, Detection, Mitigation, and Removal. https://www.sentinelone.com/anthology/babuk/, 2023.

[13] Batched, Fileless, Highly Adversarial | Annual Report on Cloud Ransomware Attacks in 2022. https://developer.aliyun.com/article/1150967, 2023.

[14] Cloud Ransomware | Understanding And Combating This Evolving Threat. https://www.sentinelone.com/cybersecurity-101/cloud-ransomware-understanding-and-combating-this-evolving-threat, 2023.

[15] Dark - VoidCrypt (.dark) ransomware virus removal and decryption options. https://www.pcrisk.com/removal-guides/24606-dark-voidcrypt-ransomware, 2023.

[16] GlobeImposter. https://malpedia.caad.fkie.fraunhofer.de/details/win.globeimposter, 2023.

[17] Google Cloud - Persistent Disk. https://cloud.google.com/persistent-disk?hl=zh-CN, 2023.

[18] How to eliminate the Mallox ransomware from a computer? https://www.malwarebytes.com/blog/news/2019/07/a-deep-dive-into-phobos-ransomware, 2023.

[19] How to remove Beijing ransomware. https://www.pcrisk.com/removal-guides/19222-beijing-ransomware, 2023.

[20] How to remove Makop ransomware and prevent further file encryption? https://www.pcrisk.com/removal-guides/16848-makop-ransomware, 2023.

[21] Kaspersky Anti-Ransomware Tool. https://www.kaspersky.com.cn/, 2023.

[22] Loki Locker (.Loki or .Rainman) ransomware virus - removal and decryption options. https://www.pcrisk.com/removal-guides/21572-loki-locker-ransomware, 2023.

[23] PhotoRec. https://www.cgsecurity.org/wiki/PhotoRec, 2023.

[24] Ransomware Protection Solutions. https://www.fortinet.com/solutions/enterprise-midsize-business/ransomware-protection, 2023.

[25] REvil / Sodinokibi: The Crown Prince of Ransomware. https://www.cybereason.com/blog/research/the-sodinokibi-ransomware-attack, 2023.

[26] Securing Your Amazon Web Services Cloud Environment Against Ransomware. https://aws.amazon.com/cn/campaigns/disaster-recovery-form/, 2023.

[27] The state of ransomware 2023. https://www.sophos.com/en-us/content/state-of-ransomware, 2023.

[28] The 2023 Global Ransomware Report. https://www.fortinet.com/content/dam/fortinet/assets/reports/report-2023-ransomware-global-research.pdf, 2023.

[29] The 2023 SpyCloud Ransomware Defense Report. https://spycloud.com/resource/2023-ransomware-defense-report/, 2023.

[30] WannaCry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2023.

[31] Windows Defender. https://www.microsoft.com/en-us/windows/comprehensive-security, 2023.

[32] H. Abdi. Coefficient of variation. Encyclopedia of research design, 1(5), 2010.

[33] M. M. Ahmadian, H. R. Shahriari, and S. M. Ghaffarian. Connection-monitor & connection-breaker: A novel approach for prevention and detection of high survivable ransomwares. In 2015 12th International iranian society of cryptology conference on information security and cryptology (ISCISC), pages 79–84. IEEE, 2015.

[34] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang. SSD-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pages 875–884, 2018.

[35] S. Baek, Y. Jung, D. Mohaisen, S. Lee, and D. Nyang. SSD-assisted ransomware detection and data recovery techniques. IEEE Transactions on Computers (ToC), 70(10):1762–1776, 2020.

[36] M. Belgiu and L. Drăguţ. Random forest in remote sensing: A review of applications and future directions. ISPRS journal of photogrammetry and remote sensing, 114:24–31, 2016.

[37] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), pages 143–157, 2011.

[38] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (KDD), pages 785–794, 2016.

[39] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer. Knn model-based approach in classification. In On The Move to Meaningful Internet Systems (CoopIS), pages 986–996, 2003.

[40] M. Hirano, R. Hodota, and R. Kobayashi. RanSAP: An open dataset of ransomware storage access patterns for training machine learning models. Forensic Science International: Digital Investigation, 40:301314, 2022.

[41] M. Hirano and R. Kobayashi. Machine learning based ransomware detection using storage access patterns obtained from live-forensic hypervisor. In 2019 sixth international conference on internet of things: Systems, Management and security (IOTSMS), pages 1–6. IEEE, 2019.

[42] M. Hirano, T. Tsuzuki, S. Ikeda, N. Taka, K. Fujiwara, and R. Kobayashi. WaybackVisor: Hypervisor-based scalable live forensic architecture for timeline analysis. In Security, Privacy, and Anonymity in Computation,

Communication, and Storage (SpaCCS), pages 219–230, 2017.

[43] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi. FlashGuard: Leveraging intrinsic flash properties to defend against encryption ransomware. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (CCS), pages 2231–2244, 2017.

[44] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. Advances in neural information processing systems (NIPS), 30, 2017.

[45] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In 25th USENIX security symposium (USENIX Security), pages 757–772, 2016.

[46] A. Kharraz and E. Kirda. Redemption: Real-time protection against ransomware at end-hosts. In Research in Attacks, Intrusions, and Defenses (RAID 2017), pages 98–119, 2017.

[47] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In 13th USENIX Conference on File and Storage Technologies (FAST), pages 273–286, 2015.

[48] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, et al. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In 21st USENIX Conference on File and Storage Technologies (FAST), pages 331–346, 2023.

[49] M. K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward: A discussion between Kirk McKusick and Sean Quinlan about the origin and evolution of the Google File System. Queue, 7(7):10–20, 2009.

[50] S. Menard. Applied logistic regression analysis. Number 106. Sage, 2002.

[51] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown. An introduction to decision tree modeling. Journal of Chemometrics: A Journal of the Chemometrics Society, 18(6):275–285, 2004.

[52] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. CatBoost: unbiased boosting with categorical features. Advances in neural information processing systems (NIPS), 31, 2018.

[53] B. Reidys, P. Liu, and J. Huang. RSSD: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 726–739, 2022.

[54] N. Scaife, H. Carter, P. Traynor, and K. R. Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In 2016 IEEE 36th international conference on distributed computing systems (ICDCS), pages 303–312, 2016.

[55] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), pages 1–10, 2010.

[56] A. Wehrl. General properties of entropy. Reviews of Modern Physics, 50(2):221, 1978.

[57] C.-Y. Yang and R. Sahita. Towards a Resilient Machine Learning Classifier–a Case Study of Ransomware Detection. arXiv preprint arXiv:2003.06428, 2020.

Secret Key Recovery in a Global-Scale End-to-End Encryption System

Graeme Connell[*]
Signal Messenger

Vivian Fang[*]
UC Berkeley

Rolfe Schmidt[*]
Signal Messenger

Emma Dauterman
UC Berkeley

Raluca Ada Popa
UC Berkeley

Abstract

End-to-end encrypted messaging applications ensure that an attacker cannot read a user's message history without their decryption keys. While this provides strong privacy, it creates a usability problem: if a user loses their devices and cannot access their decryption keys, they can no longer access their message history. To solve this usability problem, users should be able to back up their decryption keys with the messaging provider. For privacy, the provider should not have access to users' decryption keys. To solve this problem, we present Secure Value Recovery 3 (SVR3), a secret key recovery system that distributes trust across different types of hardware enclaves run by different cloud providers in order to protect users' decryption keys. SVR3 is the first deployed secret key recovery system to split trust across heterogeneous enclaves managed by different cloud providers: this design ensures that a single type of enclave does not become a central point of attack. SVR3 protects decryption keys via rollback protection and fault tolerance techniques tailored to the enclaves' security guarantees. SVR3 costs $0.0025/user/year and takes 365ms for a user to recover their key, which is a rare operation. A part of SVR3 has been rolled out to millions of real users in a deployment with capacity for over 500 million users, demonstrating the ability to operate at scale.

1 Introduction

End-to-end encrypted messaging applications like Signal [85], WhatsApp [24], and Messenger [58] are used by hundreds of millions to billions of users. They provide end-to-end encryption: user devices (the "ends") encrypt user messages so application servers receive only encrypted messages without decryption keys. Only the users in a conversation can decrypt the messages locally on their devices. This paradigm protects user messages even if the application provider or cloud infrastructure is compromised.

To provide this guarantee, end-to-end encrypted messaging application providers must ensure that their users' secret keys are protected against a wide range of attacks by malicious employees, cloud provider administrators, or other privileged agents. Unfortunately, this creates a usability problem: if a user loses their secret keys, for example by losing their devices, the user loses access to their account and message history because these keys are necessary to decrypt the user's chat history and metadata (e.g., address book, social graph). The application provider cannot directly store user secret keys because it could then decrypt user messages, violating the core principle of end-to-end encryption. Therefore, users who lose their devices should be able to recover their secret keys without the provider getting access to their secret keys.

Shortcomings of many existing key recovery systems. A potential strawman is to allow the user to download their secret keys (e.g., print them on a piece of paper) and store them in a safe place [40, 46, 59], but this places extra burden on the user [76]. A more user-friendly approach to this problem is to allow a user to use a password or a PIN to encrypt their key [33]. Unfortunately, these are often vulnerable to brute-force dictionary attacks [82, 83]. Furthermore, standard safeguards (e.g., forcing the attack to be performed online) can easily be circumvented by the application provider.

Current deployed systems [4, 43, 51, 88, 96, 98] prevent brute-force attacks by using secure hardware to limit the number of PIN guesses. This approach provides a strong protection against service provider administrators and cloud providers. While these systems all represent significant advances in password-based key recovery, they rely on the security guarantees of a *single* type of secure hardware. Although secure hardware is a powerful tool for enhancing the security of systems, it can eventually be subverted—attackers have extracted user secrets from secure hardware in the past [12, 14, 31, 35, 62, 75, 79, 87, 90, 91, 94, 95]. In these systems, compromising just one type of secure hardware enables an attacker to recover many users' secret keys, which is a catastrophic scenario for any popular encrypted system.

[*] Equal contribution.

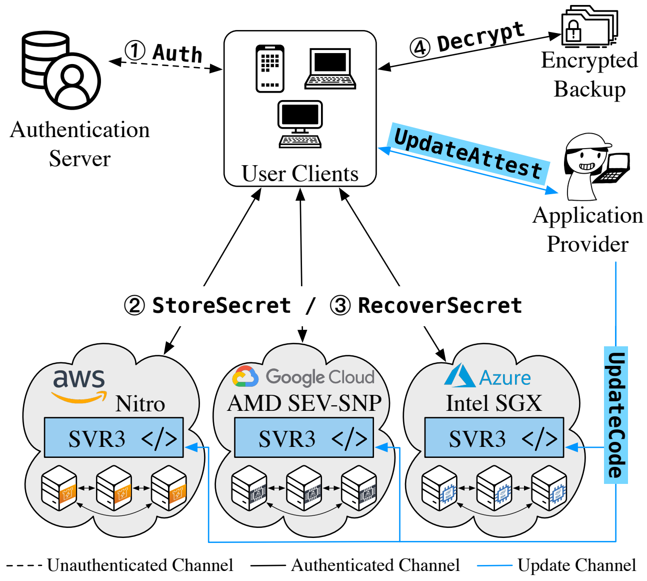


Figure 1: System architecture for $n = 3$ enclave clusters, with each cluster using a different type of hardware enclave.

Key recovery without a single point of security failure. In this paper, we contribute **S**ecure **V**alue **R**ecovery **3**[1], a PIN-based secret key recovery system that prevents any one type of enclave or cloud provider from becoming a central point of attack. Our security properties are informed by the observation that many vulnerabilities are quickly patched, and so it is challenging for an attacker to find vulnerabilities *simultaneously* on different enclave architectures. SVR3 proposes a layered architecture, illustrated in Figure 1, consisting of a tailored cryptographic multi-server key recovery protocol that distributes trust across three different enclaves from three distinct hardware vendors on three major clouds: Intel SGX in Microsoft Azure, AMD SEV-SNP in Google Cloud, and Nitro in AWS. SVR3 ensures that even if an attacker simultaneously compromises two of these enclave types and the respective clouds, the attacker cannot reconstruct the user's secrets due to the cryptographic protocol. The attacker needs to simultaneously compromise the security of all of the clouds and all of the enclave types to reach user secrets.

We implemented SVR3 as a production-ready system embedded in Signal Messenger [85], an end-to-end encrypted messaging application with tens of millions of users. We have already deployed an initial version of SVR3's implementation to millions of users globally, and the fully featured system is in the process of deployment at the time of publication. A third-party auditor, NCC Group, audited the deployment of Signal's SVR2, a predecessor system currently in production and using SVR3's consensus protocol on a single trust domain. SVR3 is open source [84] and can be used by any end-to-end encrypted system that needs secret key recovery (e.g., encrypted messaging [24, 85], email [72, 74], or storage [99]). To the best of our knowledge, SVR3 is the first deployed cross-enclave, cross-cloud secret key recovery system. The servers for SVR3 cost only \$0.0025/user/year and it takes 365ms for a user to recover their key, which is a rare operation.

Design decisions. Our design choices were guided by the goal of developing a real-world PIN-based key recovery system that prevents dictionary attacks, is easy and affordable to maintain, and provides security even if a particular enclave or cloud provider is vulnerable. We summarize the key decisions below.

A layered security architecture (§2–§3). We aim to protect users' secrets against three major classes of attackers: cloud attackers, an internal application provider attacker, and external hackers. To achieve this, one strawman is to distribute trust across multiple organizations. However, finding reliable and trustworthy such organizations is difficult and expensive [21, 50]. Instead, we introduce an architecture that layers cryptographic security on top of hardware security by using different types of enclaves in different clouds. The hardware enclaves enable creating three separate trust domains, and the cryptographic tools split secret keys across the trust domains.

PPSS to distribute trust (§4). Password Protected Secret Sharing (PPSS) [5] provides password-based key recovery while distributing trust across multiple backends and limiting attackers to online dictionary attacks. Different PPSS schemes have different deployment consequences, and we select the construction by Jarecki et al. [37] primarily because it requires no cross-trust domain communication and the server design enables clients to use different secret sharing schemes if they wish. We use this protocol to construct our one-round key recovery protocol, where the servers receive no information about whether the PIN guess was correct, and the servers unconditionally delete key material after a fixed number of PIN guesses (which can be refreshed by the clients). This is in contrast to existing works [85, 88, 98], which rely on password-based authentication and require multiple communication rounds.

Rollback protection through enclave memory and consensus (§5). Like Signal's original SVR1 system [85], SVR3 protects against *software* rollback attacks by keeping all data (e.g., guess counts) inside enclave memory. In order to prevent data loss, we replicate data across multiple enclaves in the same cloud. SVR1 uses the original Raft consensus protocol [66], which is not safe under *physical* rollback attacks. In principle, an attacker with physical access (e.g., a DIMM interposer [89]) to a single server in a vanilla Raft replica group could take control of the group and roll back log entries. To defend against such attacks, we develop a modified Raft [66] protocol, Raft$^{\circlearrowleft}$, that provides safety under physical rollback attacks, as specified in §3.2. We prove its safety under a formal TLA+ [45] model in the face of physical rollback attacks.

[1] This is the third generation of Signal's SVR service and succeeds SVR1 [51], which did not distribute trust across multiple types of secure hardware. (SVR2 was a transition system consisting of a partial SVR3 design.)

Secure code updates via auditing (§6). To enable code updates while providing strong security, we allow clients to audit the deployed code and explicitly disallow sharing of data between different (server) binary versions. Data migration between binary versions flows through the client, and clients can determine whether or not to store their secret value on each version of the binary.

Limitations. SVR3 relies on the underlying security guarantees of the enclaves it employs; supporting a new enclave or a new version of an existing enclave would require carefully reasoning about how it fits into the threat model. Splitting infrastructure across multiple cloud providers also incurs higher monetary costs than deploying on a single provider, but offers stronger security assurances. Finally, SVR3 does not support recovering the user PIN that is used in secret key recovery (i.e., if a user forgets their PIN, they cannot recover their key). We mitigate this in practice by periodically prompting the user to re-enter their PIN on the messaging client to prevent permanent lockout.

2 System overview

2.1 System architecture

Figure 1 shows the system architecture for an SVR3 deployment with three cloud providers, with the following entities:

Enclave clusters. The application owner deploys n enclave clusters (in our deployment, $n = 3$). To strengthen security, each enclave cluster should run on a different type of enclave in a different cloud environment (see §3). We will refer to each enclave cluster running on different hardware in a different cloud as a trust domain. Enclave clusters maintain replicated storage and respond to messages from clients. Each enclave cluster consists of a load balancer, a discovery service, and a geographically distributed replica group.

Authentication server. The authentication server establishes authenticated channels between clients and enclave clusters. The authentication server prevents malicious clients from exhausting PIN attempts for honest users because a client needs to authenticate to the authentication server (e.g., via an SMS code) before interacting with the enclave clusters.

Clients. Clients (e.g., mobile phones or laptops) interact with the authentication server and nodes in the enclave clusters in order to back up and recover their secret keys.

Application provider. The application provider will update the software and run monitoring and maintenance to ensure that the system is available and healthy.

2.2 System API

As shown in Figure 1, SVR3 exposes the following client API:

- Auth(client_id, client_cred) → auth_token: Establishes authenticated channel between client and server.

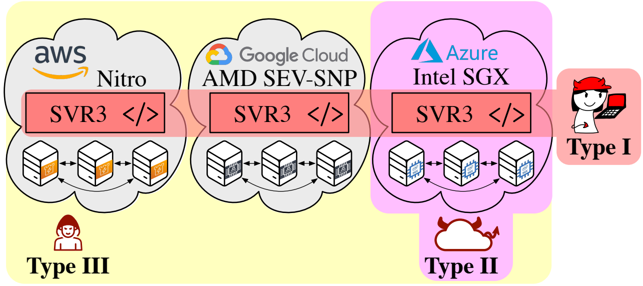


Figure 2: Types of attackers SVR3 protects against.

- StoreSecret(client_id, auth_token, val, pin): Backs up a value val for an authenticated client using a human-memorable PIN value pin and an authentication token auth_token.
- RecoverSecret(client_id, auth_token, pin) → {secret, ⊥}: Recovers the value secret for client if (and only if)
  - auth_token is valid for client_id,
  - pin matches the PIN provided at StoreSecret time for client_id, and
  - the number of unsuccessful RecoverSecret attempts for client_id does not exceed a set guess limit.

  Otherwise, outputs ⊥.

The client can use their recovered secret to locate, authorize access to, and decrypt their encrypted backup.

We describe how the developer updates SVR3 in §6.

3 Threat model and guarantees

SVR3's goal is to protect users's secrets. SVR3 provides different security guarantees against three types of server attackers, shown in Figure 2:

- **Type I (Internal).** This attacker compromises the organization deploying SVR3 (e.g., a malicious employee). This attacker does not have physical access to the cloud deployment, but can freely spin up and bring down machines and modify the software being run.
- **Type II (Cloud).** This attacker represents an entity with control over the physical infrastructure SVR3 is deployed on (e.g., a single cloud provider). While this attacker does not have the same degree of access to the entire multi-cloud system deployment, it can leverage physical access and tamper with the hardware running SVR3.
- **Type III (External).** This attacker is external to the deployment of SVR3 (e.g., a hacker), and attacks all parts of an organization's surface.

We express SVR3's security guarantees at two levels: (1) at the level of trust domains (§3.1), defining security in terms of which trust domains are not compromised, and (2) at the level of enclaves inside a trust domain (§3.2), specifying the conditions under which a trust domain is not compromised.

Like other end-to-end encrypted systems [72,74,98], if a user's device is compromised, SVR3 provides no guarantees to that user. For an uncompromised user device, we rely on the trustworthiness of client code released by Signal; we enable the community to scrutinize the client code and build trust in it by making it open-source [55–57].

SVR3 does not hide the identity of clients or the timing of backup and recovery requests.

3.1 Security across trust domains

SVR3 protects users' secret keys if at most t out of n trust domains are compromised. We assume that the odds of an attacker identifying and exploiting vulnerabilities *simultaneously* across $> t$ trust domains is low, which motivates our threat model. By simultaneous, we mean within the time period it takes to become aware of a vulnerability and replace the enclaves in the trust domain impacted by that vulnerability.

In our deployment of SVR3, we set $t = 2$ and $n = 3$, so we ensure security as long as at least one trust domain is not compromised. We limit PIN guesses by selecting a parameter u, a server *usage limit*.

Theorem 1 (Informal)**.** *In an SVR3 deployment configured with n trust domains, threshold t, and a usage limit u, assuming a password-protected secret sharing scheme (defined in §4.2), if an attacker compromises $\leq t$ trust domains, then SVR3 ensures that, for each secret key, the attacker only has $\lfloor \frac{nu}{t+1} \rfloor$ PIN attempts and, after that, cannot recover the secret key.*

We describe how SVR3 achieves Theorem 1 in §4.2.

3.2 Security within a trust domain

We now describe the threat model we consider when instantiating the trust domains assumed in §3.1. Recall that each trust domain consists of an enclave cluster and that each trust domain should use a different type of enclave.

3.2.1 Enclave threat model

SVR3's design is not tied to some specific enclave implementations. Different enclaves vary in design, so we abstract out the security properties that we require from the enclaves employed for SVR3's security guarantees (§3.2.2) to hold. An *uncompromised enclave* must provide:

(E1) *Application-level attestation*. The enclave can prove that certain code is running before other systems interact with it.

(E2) *Access control*. Enclave memory is encrypted, and access control is hardware-enforced to prevent all non-enclave access.

(E3) *Page-level rollback granularity*. The attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating global memory integrity. We assume that an attacker cannot mount these attacks at a sub-page granularity (e.g., address level) either because the enclave protects this or other protection mechanisms are used in the enclave (see below).

Deviations from enclave threat model. We describe the properties of different enclaves and how they fit our threat model in §A of the full version [17]. Some recent enclaves use AES-XTS, which encrypts in 16B increments [15]. While our design currently targets enclaves that can only be rolled back at the page-level granularity (E3), we can implement atomic regions (regions that are guaranteed to run without interruption by an attacker) by utilizing the interrupt handler introduced by AEX-Notify [18]. We describe how to do so in §5.3. Given the changing landscape of enclave implementations and the possibility that enclaves may not adhere to (E1)–(E3) in the future, we assume that alternative mechanisms like AEX-Notify can be developed to address such discrepancies between real-world enclaves and our enclave threat model.

Attacks on enclaves. Enclaves are susceptible to attacks. We list four categories here and then discuss when SVR3 hardens a trust domain against them.

(A1) *Memory access pattern attacks*. Enclaves do not hide memory access patterns, enabling a large class of side-channel attacks, including but not limited to cache attacks [9,32,61,80], branch prediction [48], paging-based attacks [93, 100], and memory bus snooping [47].

(A2) *Software rollback attacks*. Enclaves are also susceptible to rollback attacks, also referred to as freshness or replay attacks [69]. Software rollback attacks occur from rolling back persisted state outside of the enclave's memory (**Type I** attacker).

(A3) *Hardware rollback attacks*. An attacker with physical access to the system bus can roll back enclave memory at the page level without detection (**Type II** attacker), for example, using a DIMM interposer [89].

(A4) *Other attacks*. Certain physical attacks allow an attacker to break guarantees (E1)–(E3) of enclaves (e.g., leakage due to power consumption [14, 62, 87] or denial-of-service attacks due to memory corruptions [31, 35]). Transient execution attacks [12,75,79,90,91,94,95] exploit speculative execution to leak secret data.

3.2.2 Security guarantees

SVR3 *hardens* a trust domain against a set of attacks, rendering the trust domain uncompromised despite those attacks. We describe the conditions below:

(H1) SVR3's memory-access patterns do not depend on user secret content, and hiding *which* user is recovering their key is a non-goal for SVR3, so it does not suffer from memory-access patterns side-channel attacks (A1).

(H2) SVR3 defends against software rollback attacks (A2).

(H3) SVR3 defends against hardware rollback attacks (A3) as long as $\leq s$ nodes in each cluster are simultaneously rolled back, where s is a fault-tolerance ("supermajority") parameter defined in §5.2.5. In our production deployment, we set $s = 2$.

(H4) Within a trust domain, SVR3 does not guarantee protection against other attacks (A4), which could render the trust domain compromised. In this case, SVR3 still offers the cross-trust domain security guarantees in §3.1.

3.3 Availability

Like other end-to-end encrypted systems [72, 98], Signal prioritizes security over availability of secret key recovery because users' secret keys are extremely sensitive and crucial to safeguard in an end-to-end encrypted system. Nevertheless, SVR3 provides availability to clients when at least $t+1$ trust domains are operating correctly. By correct operation, we mean that enclaves in the trust domain are online and none of the enclaves in the trust domain are under attack. Therefore, we expect the system to be available under normal operation.

SVR3 also does not defend against denial-of-service (DoS) attacks from a **Type I** attacker (since this is the organization that deploys SVR3 itself) or the authentication server.

SVR3 ensures that a malicious client cannot deny availability for an honest user (e.g., by exhausting the number of PIN attempts allowed) assuming that the attacker did not compromise the client credentials or the authentication server (used to Auth in Figure 1), and it did not otherwise compromise the servers beyond the availability threshold above.

It is important to consider what users would experience if trust domain(s) were to fail, leading to secret value loss. While this is a significant event when viewed from the perspective of the application provider, it will not lead to secret value loss for the majority of clients in practice: clients cache their SVR3-protected secret, and so clients can simply create a backup at the new deployment. Thus data loss is only a concern for users who lose their devices after the old deployment fails and before migration to the new deployment completes.

4 Secret key backup and recovery protocols

We now describe the cryptographic protocols in SVR3.

4.1 Establishing enclave sessions

To interact with the SVR3 servers, the client must first authenticate with the authentication server. If the user has lost their devices, then the authentication server sends the client an SMS code, and then the user enters the SMS code to receive a token. This process allows the authentication server to prevent malicious clients from denying service to honest users by exhausting all of their PIN attempts. Notably though, the authentication server does not have any information about user PINs. The client then uses this token to establish a secure channel with a replica in each trust domain. As part of the process of establishing a secure channel, the client runs remote attestation [16] with the enclaves to ensure that it is communicating with the expected enclaves.

4.2 PIN-protected secret sharing

In existing deployed PIN-based backup systems [43,51,96,98], a secure hardware device has access to users' secret keys and PINs or PIN-derived information in order to authenticate users. This design means that an attacker that compromises the secure hardware can, either directly or via a brute-force attack, learn user PINs. This property is particularly problematic when we consider the fact that many users re-use PINs across services.

As a result, when designing our cross-enclave cross-cloud solution, we cannot simply instantiate the above mechanism in each trust domain. Any one compromised trust domain would have access to the PIN, enabling the attacker to recover the user's secret key. Instead, we leverage the class of cryptographic protocols called *password-protected secret sharing (PPSS)* [5] protocols, which ensure that:

- An attacker that compromises $\leq t$ trust domains is still limited to an online dictionary attack.
- If an attacker fully compromises $> t$ trust domains, the attacker does not immediately learn client secrets. The attacker still must perform an offline dictionary attack on user PINs.

Identifying a suitable PPSS scheme for SVR3. Different PPSS schemes have different tradeoffs [1, 5, 36–38], so we worked to identify the most suitable scheme for SVR3 and then tailor it to our setting. Some prior work optimizes for metrics that are not important to our deployment, but sacrifices properties that are important to us.

For example, many of these works aim to reduce the number of exponentiations to improve efficiency [1, 36–38]. However, the number of exponentiations is not a bottleneck in our setting, especially because the number of trust domains (3) is small. The scheme with the fewest exponentiations [38] also requires coordinated server initialization and necessitates choosing secret sharing parameters at deployment time. Coordinated initialization could require us to redeploy all trust domains every time a single trust domain requires a security upgrade, and cross-trust-domain communication with security against **Type I** attackers is difficult. Choosing a secret sharing scheme at deployment time tightly couples PPSS parameters with clients and servers, removing the flexibility to modify client PPSS parameters without also changing the servers.

With these priorities in mind, we identified the PPSS from Jarecki et al. [37] as the most suitable because it is particularly simple: each backend generates a new secret key for a client when the client creates a new backup and then uses this key to evaluate an oblivious pseudorandom function (OPRF) [28] during secret reconstruction. Informally, a pseudorandom function (PRF) is a keyed function $F_k(\cdot)$ that, for a randomly chosen key k, appears to be random (indistinguishable from a function chosen uniformly at random from all functions with the same domain and range), even though it is deterministic and efficiently computable. An *oblivious* PRF is a two-party protocol where the server holds k and the client holds some input x. The protocol enables the client to learn $F_k(x)$ without the server learning anything about x or $F_k(x)$.

This PPSS scheme has several properties that are appealing for a real-world deployment:

- The protocol is one-round and concretely efficient.
- Different trust domains do not communicate with each other.
- Servers need minimal configuration. In particular they do not need any information about the threshold scheme being used, and different clients can use the same server with different threshold schemes.
- The protocol can use a standards-track OPRF with optional verifiability [23].

We note that the WhatsApp key recovery system uses a password-authenticated key agreement (PAKE) scheme [24, 98], and SVR3 does not. While PAKE protocols are a commonly cited application for PPSS schemes, we do not need to establish a session between our client and a server. We only need to recover a secret key, which is a simpler problem. Since branching while fetching secret shares is not sensitive, we do not need to layer oblivious data retrieval on top [22, 60].

Augmenting PPSS with usage limiting. Limiting attackers to a fixed number of password guesses is a core requirement for SVR3. While the application provider can use an authentication server for access control and rate limiting, this only restricts external users. SVR3 must limit powerful attackers with full administrative and physical access to the servers to the same finite number of guesses.

We solve this by leveraging our distributed-trust setting to enforce a *usage quota* on OPRF evaluations. A standard OPRF [28] allows a server with a PRF key to evaluate a PRF on a client input without learning the input. SVR3 allows the client to set a usage limit, u, at registration time, and each honest trust domain will delete that client's OPRF key after u OPRF evaluations. In order to instantiate an honest trust domain, we use enclaves to ensure that the server enforces the usage limit. Note that the security guarantees provided by PPSS and the heterogeneous enclaves are tightly coupled: the enclaves are critical for instantiating trust domains, and PPSS enables splitting a secret value across different trust domains.

In the below proposition, we bound the number of total OPRF evaluations based on the threshold t and trust domains n, providing the protection described in Theorem 1.

Proposition 1. *For a (t, n) instance of PPSS [37] with a usage-limited OPRF configured to allow u evaluations, an adversary has at most $\lfloor \frac{nu}{t+1} \rfloor$ PIN attempts before the secret cannot be recovered.*

Proof. Only nu OPRF evaluations are possible in the system. $t+1$ evaluations are needed to perform one PIN attempt. After $\lfloor \frac{nu}{t+1} \rfloor$ PIN attempts, $(t+1)\lfloor \frac{nu}{t+1} \rfloor$ OPRF evaluations have been used. Only $(t+1)\{nu/(t+1)\} < t+1$ more evaluations are possible, where $\{\}$ denotes the fractional part, that is, $\{x\} = x - \lfloor x \rfloor$. This is not enough to reconstruct the secret. □

5 Building a SVR3 backend

We now describe SVR3's system design within one trust domain. Per our threat model in §3, each uncompromised SVR3 trust domain consists of a cluster of machines, which we assume behave correctly except for possible physical rollback attacks and crash failures within a specified bound.

5.1 Design decisions

We first provide an overview of the design decisions behind SVR3's design to ensure fault tolerance and the security guarantees in §3.2.2.

Use of enclaves. In order to protect server secrets and allow clients to check the code that is processing their data, we run the core part of the service in an attested, confidential enclave.

In-memory database to avoid sealing. Data sealing is a mechanism whereby an enclave can encrypt internal state with a key that is unique to the platform and enclave, persist the encrypted data to disk, and then recover it if the enclave is torn down and restarted. As noted in prior work [26, 97], applications in commercially available enclaves that use data sealing to store state externally and recover from crashes are vulnerable to simple, software-based rollback attacks. Since a core function of SVR3 is to faithfully maintain a per-user OPRF evaluation count, rollback attacks would undermine the system and could allow an attacker unlimited online password guesses. To prevent this and achieve (H2), the enclave that stores the database of client secrets and usage counters is kept entirely in enclave-protected memory; it is *never* sealed and written to untrusted memory or disk. We show that the database fits entirely in memory without sharding users in §8.1.

Distributed consensus. Without a data persistence mechanism (e.g., data sealing), the servers cannot recover from crashes, and data in any failed server will be lost. To ensure that data is not lost, we build the service as a geographically distributed database. To ensure split-brain or other attacks do not allow excess PIN guesses, we use a distributed consensus protocol, modified from Raft [66]. We give a high-level overview of vanilla Raft in §5.2.1. Our modified Raft protocol, Raft$^{\circlearrowleft}$, which we describe in §5.2.3, hardens vanilla Raft against physical rollback attacks and ensures that client requests and usage count changes are committed before responding to client queries. We describe in §5.3 how we use Raft$^{\circlearrowleft}$ to achieve global integrity across the database when assuming page-level rollback granularity of enclaves (E3), achieving (H3).

5.2 Rollback-resistant consensus protocol

SVR3 already protects against the class of rollback attacks that arise from storing state outside of the enclave by keeping all state in memory. However, as discussed, machines can fail, and so in order to tolerate failures without losing data, we use Raft$^{\circlearrowleft}$, a modified version of vanilla Raft across enclaves from a cloud provider. A full TLA+ description of Raft$^{\circlearrowleft}$ is available in §E of the full version [17], and we provide a proof of safety based on the TLA+ specification in §D.

In this paper, we use n to refer to the number of trust domains and m to refer to the number of replica machines *within* a trust domain.

5.2.1 Vanilla Raft background

Raft [66] is a consensus algorithm that manages a replicated log across multiple nodes (replicas). It elects a single leader replica that receives and replicates log entries to the other follower replicas. The leader handles all client requests by appending new log entries and sending an AppendEntriesRequest to each follower for the duration of its *term*. Follower replicas respond to requests from the leader to replicate log entries. If the leader fails, a new leader is elected through a leader election process. Log entries are identified by `<index, term>`, where `index` is the log position and `term` is the current term number. There is at most one leader in any given term. A leader forces the followers' logs to duplicate its own: conflicting entries in follower logs (with some `term` t) will be overwritten with entries from the leader's log if the leader's term t' is $\geq t$. For f crash failures, Vanilla Raft requires $m \geq 2f+1$ replicas in order to provide safety and liveness.

5.2.2 The physical rollback problem

While keeping the database in memory protects against software rollback attacks, an attacker with physical access to the system bus could roll back enclave memory at the page level. Since such an attack is more expensive to perform than software-based rollback attacks, we can significantly improve security by requiring an attacker to perform these attacks simultaneously on multiple enclave replicas. With this context, we note that the vanilla Raft protocol [66], as specified, will allow an attacker who can roll back a Raft leader to make an unlimited number of PIN attempts: the Raft protocol does not look at log contents, so if a leader is rolled back and sends an AppendEntriesRequest for a new `<index, term>` log entry at an old log index, followers will accept it and allow the leader to commit.

Prior work [26, 97] has addressed a problem close to this one, but with important differences. First, they are designed for data-sealing rollbacks, which do not affect SVR3 because we do not use data sealing. Second, Raft$^{\circlearrowleft}$ also defends against physical rollback attacks, which prior works do not consider in their threat model. Physical rollback attacks are more difficult to detect than data-sealing rollback attacks: after a crash recovery, the new enclave has to execute code that decrypts the sealed data to rebuild the internal state and every data-sealing rollback needs to have the enclave go through this code path. The RR protocol [26] takes advantage of this process to detect data-sealing rollback attacks. Finally, existing protocols aim to ensure liveness in the face of rollback attacks, and this is an explicit non-goal for SVR3 as mentioned in §3.3.

5.2.3 Rollback prevention in Raft$^{\circlearrowleft}$

Together, the following additions to the Raft protocol enable us to prove safety of Raft$^{\circlearrowleft}$ in the presence of an attacker who can simultaneously mount physical rollback attacks against $\leq s$ nodes. For m Raft$^{\circlearrowleft}$ servers in a trust domain, s must be strictly smaller than m to ensure safety (§5.2.4). However, to ensure fault tolerance and liveness in the face of crash failures, s should be even smaller (§5.2.5).

Hash chain. Instead of using `<index, term>` to identify a log entry, as in Raft, we use `<index, term, hash`$_{\texttt{index}}$`>` where `hash`$_{\texttt{index}}$ = `Hash(entrydata, index, term, hash`$_{\texttt{index-1}}$`)`, `entrydata` is the contents of the log entry, and `Hash` is a cryptographic hash function. When a follower receives an AppendEntriesRequest, it computes the expected hash chain value and verifies that it matches the value in the request. If the values do not match, the follower rejects the request.

This prevents the simple rollback attack on Raft described in §5.2.1. However, it is still possible for an attacker who can roll back one server to gain unlimited password guesses by triggering an election with a quorum of servers that did not see the log entry for the first client request.

Supermajority. To ensure that an attacker capable of rolling back a single server cannot gain extra password guesses by triggering an election, we require quorums to have a supermajority of replicas so that the intersection of any two quorums contains more than s replicas, where s is a configurable parameter that is included in the server's attestation. This allows clients to be certain of the value of s used by the service and decide whether to accept it. We prove that an attacker must be able to roll back more than s enclaves to roll back a log entry that was committed by this Raft$^{\circlearrowleft}$. This supermajority parameter is comparable to PBFT's Byzantine nodes value [10].

Promise round. We add a *promise round* to the protocol. We discuss reasoning for why we add a promise round in the full version [17]. Once a quorum of servers acknowledges seeing a log entry, the leader will “promise" this entry by advancing its `promise_idx` to the index of this entry. A promised entry is not committed, but no replica will delete an entry that has been promised. This completes the first round.

The leader now sends its `promise_idx` to all followers in its next AppendEntriesRequest, and followers will update their own `promise_idx` to match the leader's when they process the message. From this point, these followers have promised the log entry and will not delete it. The followers send their current `promise_idx` with each AppendEntriesResponse. Once a quorum of replicas has promised an entry, it is committed.

5.2.4 Safety

In order to achieve safety, the number of machines in the enclave cluster must be larger than the number of rollback attacks we want to tolerate ($m > s$). As liveness under rollback attacks is a non-goal for SVR3 (an attacker with physical access can easily deny service), we decouple the constraints on m with respect to rollback attacks (s) and crash failures (f_c). We describe how s impacts liveness under crash failures in §5.2.5. We prove that Raft$^{\circlearrowleft}$ is safe under a bounded number (s) of physical rollback attacks within a trust domain.

Theorem 2 (Informal)**.** *Let M_R be the maximum number of machines in an enclave cluster that can be rolled back and s be our supermajority configuration parameter. If $M_R \leq s$, then under standard cryptographic assumptions, for every log entry <`index, term, hash`$_{index}$> that has been applied to the state machine of a server i, server i will never apply a different log entry at this index.*

Proof sketch. The argument follows the proof of safety in Ongaro [65] and relies on the observation that any two quorums will have an intersection that includes at least one server that has not been rolled back. We must address the fact that in the presence of rollbacks, Lemma 3 in Ongaro [65] does not hold. This poses a significant challenge, and forces us to introduce a new concept of *live committed* entries that is subtly different from the prior notion of committed [65]. With our definition, future leaders may not have a live committed entry in their log, but if they do not then they will be unable to commit new entries, so we retain safety at the expense of liveness. The major point where the argument from Ongaro [65] breaks down in our setting is in points 7.c.ii.B and 7.c.iii.B in the proof of their Lemma 8. Our argument uses the hash chain and promise index to show that there is a voter in the intersection of two quorums that has not been rolled back and will not replace the log entry. The complete proof of safety is in §D of the full version [17].

5.2.5 Liveness

We do not provide liveness for a trust domain under the setting of an attacker mounting physical rollback attacks, as the attacker could trivially deny client requests by taking the entire enclave cluster offline. When assuming no attacks within a trust domain, Raft$^{\circlearrowleft}$ requires $f_c \leq \lfloor (m-s)/2 \rfloor$ *crash* failures to be live under normal connectivity conditions, where m denotes the number of replicas in a trust domain (enclave cluster) and s denotes the supermajority parameter described in §5.2.3. This is due to the quorum size being $\lfloor (m+s)/2 \rfloor + 1$ enclaves. It remains an open problem to prove liveness of Raft in this setting (e.g., by formal verification [34]). Nevertheless, as discussed in §3.3, SVR3 still provides availability to clients when at least $t+1$ trust domains are operating correctly.

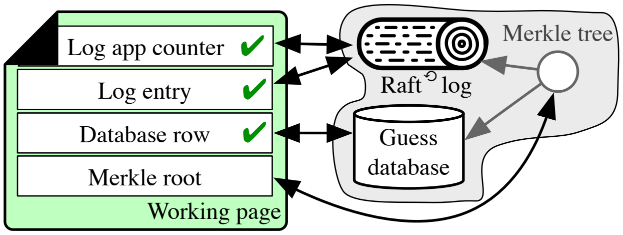


Figure 3: Integrity across database. In order to achieve global integrity, updates are only applied when all state on the working page validates under the same Merkle tree root.

5.2.6 Self-healing for simple maintenance

We implement the process for replica group membership changes described in the Raft paper [65] and add a layer of automation. In Raft$^{\circlearrowleft}$, a replica group has a configured target number of voting members. For a healthy configuration, a replica group in our system will have this number of voting members as well as several non-voting members that stay up to date and service client requests. If some voting member is not seen by the leader after a configurable timeout, the leader will initiate a membership change that demotes the missing replica to non-voting status. After an additional timeout, it will remove the replica from the group entirely.

Furthermore, whenever the number of voting members is below the configured target, the leader will check to see if a non-voting member is present and initiate a membership change promoting a non-voting member to voting status.

With these mechanisms in place, administrators simply need to launch new instances and direct them to the discovery service with group information. The new server will then request to join the group, be brought up to date by a peer, and become a non-voting member. As needed, the voting members may then promote this new replica to voting status.

5.3 Integrity across the database

Raft$^{\circlearrowleft}$ provides protection against rollback attacks on the contents of the log. However, our threat model (§3) assumes *page-level* rollback granularity on memory inside the enclave, which means that the attacker can replace pages of data in the enclave's memory with older pages from the same physical location and can mix and match old and new pages, thus violating *global* memory integrity.

In order to protect against rollback attacks on the backing in-memory database, SVR3 keeps a Merkle tree across the Raft$^{\circlearrowleft}$ log, database, and log application counter.

5.3.1 Merkle tree

The log application counter keeps track of the latest log entry that has been applied to the database. The Merkle tree contains every database row, the hashchain of the most recently committed log entry, and the log application counter. The hashchain of the last committed log entry, as described in §5.2.3, can be used to verify this entry and earlier entries in the log. As shown in Figure 3, the Merkle leaves for database rows and log application counter are updated each time the underlying object changes, and the update only succeeds if the current state of the Merkle tree is consistent with the previous value of that data.

5.3.2 Applying committed log entries

We describe how we process committed log entries in Algorithm 1. The executing thread holds a lock on the database, log, and log application counter throughout execution, so no honest process will have a thread outside this process change the Merkle tree during that execution. When applying a commited log to the local database, a replica will begin by reading the log application counter lac, the log entry at that index entry, and the database row row referenced by that log entry onto a single memory page, which we will call the *working page*. When reading each of these items, it will verify its Merkle proof ($\Pi_{\text{lac}}, \Pi_{\text{entry}}, \Pi_{\text{row}}$) and also copy the root of the Merkle tree for each read onto the working page. After copying this data, we verify that the Merkle roots associated with each read are equal, determine whether the number of uses of this row has surpassed the configured maximum, and update the row by incrementing the usage count and deleting the OPRF secret if the maximum usage count has been exceeded. We then update the row in the database and increment the log application counter, updating the Merkle tree entries for both, then proceed with evaluating the OPRF, if the key is present, and finally respond to the client.

If the attacker rolls back the database row to the contents of a previous timestep, it first has to roll back every entry from the row to the Merkle tree root. However, the root also covers the log entries and log application counter, which are modified when a database row is modified (how SVR3 achieves atomicity of this operation is described above). Thus, the attacker will have to roll back the log as well; rolling back the log is exactly what Raft$^{\circlearrowleft}$ protects against.

Atomic regions. Because all of our working memory fits on a single page, operations are atomic with respect to the attacker's ability to rollback memory at the page granularity. In order to support more modern enclaves that only have cache line granularity (e.g., 16B), we need to implement atomic regions that are guaranteed to run without interruption by an attacker. We describe in detail how to implement atomic regions on SGX and SEV-SNP in §C of the full version [17] by utilizing the interrupt handler in AEX-Notify [18]. AEX-Notify mitigates SGX-Step, an attack framework that makes it possible to single-step enclave programs [92]. It does so by introducing an instruction set architecture extension to support a custom handler on interrupt. The SGX-Step mitigation leverages this handler to speed up the next instruction so that the attacker is statistically unlikely to 'hit' the next instruction's execution with an APIC timer. This mechanism also allows us to implement atomic regions, in a similar fashion to restartable sequences [8]. At a high level, we set a flag in a fixed register when an interrupt occurs, and we check this flag at the end of the atomic region to determine whether to restart the atomic region. If the flag is set, we restart and retry until it runs without any interrupt. We leave optimizing this approach in a secure manner to future work.

Algorithm 1 Applying a committed log entry. We describe in text how we process committed log entries in §5.3.2.

```
$\text{workspace}_R \leftarrow (\text{lac}, \Pi_{\text{lac}}, \text{entry}, \Pi_{\text{entry}}, \text{row}, \Pi_{\text{row}})$
Atomic region.
                                  ▷ Abort on any Verify failure.
    $\text{failure} \leftarrow 0$
    $\text{Verify}(\Pi_{\text{lac}}.\text{root} \overset{?}{=} \Pi_{\text{entry}}.\text{root} \overset{?}{=} \Pi_{\text{row}}.\text{root})$
    $\text{Verify}(\text{entry.clientid} \overset{?}{=} \text{row.clientid})$
    $\text{Verify}(\text{lac}, \Pi_{\text{lac}}); \text{Verify}(\text{entry}, \Pi_{\text{entry}});$
    $\text{Verify}(\text{row}, \Pi_{\text{row}})$
    if $\text{row.guess\_cnt} < \text{max\_guesses}$ then
        $\text{evaluated} \leftarrow \text{OPRFEval}(\text{row.sk}, \text{blinded})$
        $\text{row.guess\_cnt} \leftarrow \text{row.guess\_cnt} + 1$
    else
        $\text{failure} \leftarrow 1$
        $\text{row.sk} \leftarrow 0, \text{row.guess\_cnt} \leftarrow \text{UINT\_MAX}$
    end if
    $\text{workspace}_W \leftarrow (\text{row}, \text{UpdatePrf}(\text{row}, \Pi_{\text{row}}))$
$\Pi'_{\text{row}} \leftarrow \text{UpdatePrf}(\text{row}); \Pi'_{\text{lac}} \leftarrow \text{UpdatePrf}(\text{lac})$
Check that leaves on path in $\Pi'_{\text{row}}, \Pi'_{\text{lac}}$ match $\Pi_{\text{row}}, \Pi_{\text{lac}}$.
if failure then return MISSING
else return (OK, evaluated)
end if
```

6 Operations

Production systems need upgrades. This is a challenge for us because we want to defend against malicious administrators: a secure system can become completely insecure if a malicious administrator can push arbitrary code to the system. At a high level, we defend against malicious code updates by ensuring that users can audit the code that is running; the code is open source, and enclaves attest to the security-relevant server code and configurations running.

Adding new servers. When a new server is launched in a trust domain, it connects to a discovery service and registers a new group if no replica group is registered. If there is an existing replica group, the new server will select a peer in that group, validate that its enclave measurements match, and create an attested connection with that peer. By checking that enclave measurements match, SVR3 ensures that an administrator cannot add a server running different code. The new server then requests to join the group, and the existing server transfers all log entries and database rows to the new server. This is done over a Noise protocol [71] channel with key resetting and hybrid post-quantum forward secrecy [70] to provide robust forward secrecy. Once the transfer is complete, the replica group goes through the membership change process to add the new server (which requires a quorum).

Sometimes security-required microcode updates need to be applied to all servers. Since all data is kept in volatile enclave memory, there is no way to reboot the machine without losing all replica data. In this situation, *all members of the cluster must be replaced.* This can be done by sequentially adding new servers on patched hardware, then terminating old servers.

Clients. Android, iOS, and desktop clients are deployed through app stores with auditable, open-source code. Each client contains hard-coded information about which enclave measurements (for remote attestation), platform versions, and cluster configurations to accept. If a client attempts to connect to a SVR3 cluster and finds unexpected measurements or configuration, it will abort the connection.

Service upgrades and data migration. Since server enclaves can only communicate with peers that share the same enclave measurements, there is no mechanism to migrate data directly from an old version of an enclave-backed service to a new one. Instead, data migration flows through the client. To accomplish this, when a new version of a client is released that contains measurements for the new enclave, this client will recover its secret from the old servers (if it is not cached in local storage), and then it will back up its secret to the next version of the service. It takes approximately 90 days for a new client software release to fully reach the user base, so the new enclave-backed service must run alongside the older version during this 90-day window.

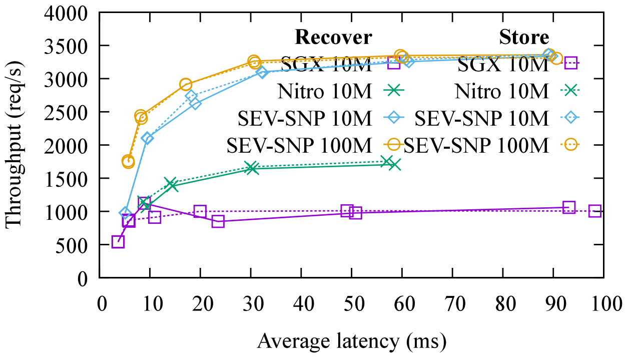


Figure 4: Average latency vs. throughput.

7 Implementation

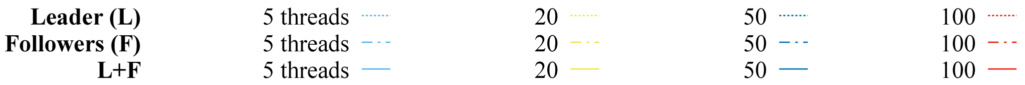
We implemented SVR3 in ~8,800 lines of C++ for the enclave and ~5,300 lines of Go for the untrusted host. For the SGX deployment we use the OpenEnclave framework v0.19 [67] and Intel SGX v2.22. For the Nitro deployment we use the Nitro Security Module library v0.4 [63]. We use a Noise protocol [71] channel on top of TCP for communication between replicas and websockets for communication with clients. We use protobuf [73] to define formats for all wire messages. In addition to handling client and peer requests, the host offers a control interface for administration as well as sophisticated metrics collection that is integrated with our internal monitoring and reporting systems. Our implementation assumes enclave page-level integrity, and we estimate overheads for supporting 16B-level rollback granularity in §8.1. The implementation is open source and the consensus system is already in production use. The full system is being deployed to production at the time of publication. Production deployments use 7 geographically distributed servers and a supermajority parameter of 2. Further details about the production deployment are in §B of the full version [17].

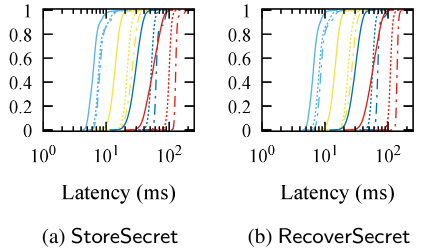
8 Evaluation

We investigate the overheads of running SVR3 (§8.1) and the performance perceived by the end user (§8.2).

Evaluation setup. For the purposes of this paper, we evaluate end-to-end performance on our organization's staging system, configured to handle 10 million users. This limit is due to available enclave memory, not compute. Staging clusters are configured with a supermajority parameter of 1 and consist of 3 environments (trust domains), each with 5 replicas deployed in the same region:

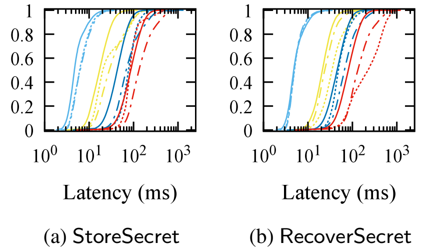
- AWS Nitro: `m5.xlarge` instances with 2 cores and 10 GB RAM per enclave ($142/month/server).
- Intel SGX at Azure: `DC2s_v3` instances with 2 cores and 8 GB EPC RAM per enclave ($140/month/server).





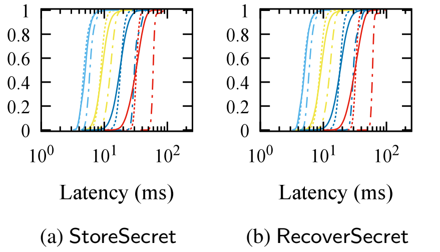
(a) StoreSecret (b) RecoverSecret

Figure 5: Request latency CDF for AWS Nitro, varying number of client threads, 10M users.



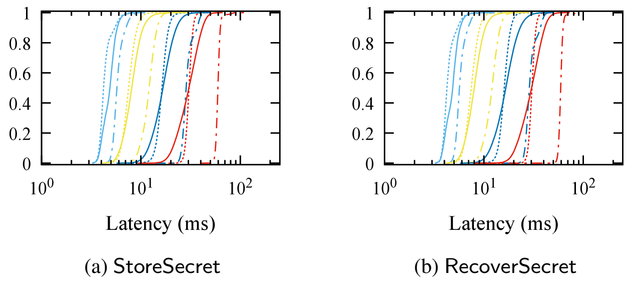
(a) StoreSecret (b) RecoverSecret

Figure 6: Request latency CDF for Intel SGX, 10M users.



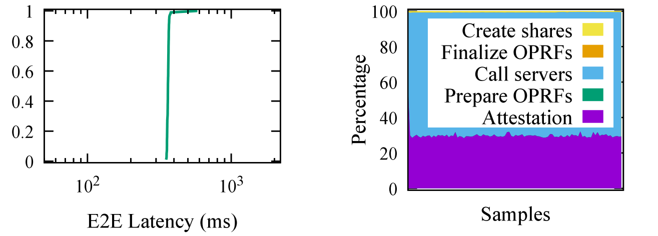
(a) StoreSecret (b) RecoverSecret

Figure 7: Request latency CDF for AMD SEV-SNP, 10M users.



(a) StoreSecret (b) RecoverSecret

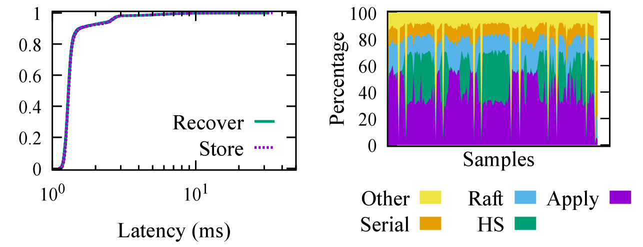
Figure 8: Request latency for AMD SEV-SNP, 100M users.



(a) Request (StoreSecret) latency CDF.

(b) Client request latency breakdown.

Figure 11: End-to-end performance.



(a) Request latency CDF.

(b) Request latency breakdown. HS = Noise handshake, Serial = serializing/deserializing protobufs, Apply = applying log entry (§5.3.2).

Figure 9: SVR3 performance without network latency from Raft$^{\circlearrowleft}$.

Enclave	**Network (B/user)**			
	StoreSecret		RecoverSecret	
	C ↔ S	S ↔ S	C ↔ S	S ↔ S
SGX	20,717	288–1,276	20,717	224–1,212
SEV-SNP	4,406	288–1,276	4,406	224–1,212
Nitro	4,593	288–1,276	4,593	224–1,212

Table 10: Network usage for a single client request to a 3-replica cluster. S=server, C=client. C ↔ S for SEV-SNP is an estimate.

- AMD SEV-SNP at GCP: 2 `n2d-standard-2` instances per enclave (one "confidential" and one for the untrusted host) with 2 cores and 8 GB RAM ($2 \cdot (\$70) =$ \$140/month/server).

In total, the staging cluster costs \$2,110/month to run (\$0.0025/user/year). For microbenchmarking, we evaluate on a testing cluster with the same machine types as our staging cluster but with 3 replicas per trust domain instead of 5 and a supermajority parameter of 0 instead of 1.

Our production infrastructure has more replicas (with more cores and RAM per replica) and is set up to handle over 500 million users (more details in §B of the full version [17]). We provision for 1 req/s/1M users and ~256B of RAM/user. Our experience operating this system gives us confidence that evaluating on the staging infrastructure is meaningful and that SVR3 scales gracefully. To validate this claim, we also evaluate on an AMD SEV-SNP cluster with 100 million users using `n2d-standard-4` instances (4 cores and 16 GB RAM).

8.1 Microbenchmarks

Throughput. We plot an average latency vs. throughput curve for write and recovery requests in Figure 4. We generate each point by varying the number of client threads and measuring the average latency and throughput of requests. Requests are spread out across all 3 servers. For the 10M-user deployments, the throughput of recovery requests levels off around 1,700 req/s for Nitro, 1,000 req/s for SGX, and 3,300 req/s for SEV-SNP (for both 10M-user and 100M-user deployments).

Latency. We plot CDFs of the latency of write and recovery requests in Figure 5, Figure 6, and Figure 7 for Nitro, SGX, and SEV-SNP, respectively. Within each figure, we plot the latency when requests are sent only to the leader, when requests are sent only to followers, and when requests are sent to all 3 servers. Requests sent to followers are forwarded to the leader, so the average latency of requests at followers is higher than at the leader. The latency distribution of requests when sending requests to all 3 servers improves compared to sending requests to only followers. The latency distribution is better than sending requests to only the leader for Nitro and SGX, and the tail latency is worse than sending requests to only the leader for SEV-SNP. At 100 client threads, the average latency for requests sent to all servers for key recovery is 56.9ms for Nitro, 98.3ms for SGX, and 32.3ms for SEV-SNP. We also plot the CDFs of recovery request latency for the 100M-user SEV-SNP deployment in Figure 8. The latency distribution of the 100M-user deployment is very similar to the 10M-user deployment and the average latency of the requests sent to all 3 servers for key recovery is 30.9ms.

We note that a majority of the latency is due to network latency when appending to the Raft$^{\circlearrowleft}$ log, which we validate in Figure 9. We run the same experiment as above, but with 1 client thread and 1 SGX node (effectively disabling the network requests of Raft$^{\circlearrowleft}$). We plot the CDF of request latencies under this regime in Figure 9a, and the average latency of these requests is 1.47ms. We also profile the server and plot the percentage of CPU ticks in Figure 9b. On average, the Noise handshake is about 35%, applying the log entry is about 21%, and 13% is encrypting peer messages for Raft$^{\circlearrowleft}$. The yellow spikes are due to periodic updating of environment statistics, which also contributes to the long tail request latencies in SGX (Figure 6).

Impact of supporting 16B-granularity. Informed by latency measurements, we can upper-bound the impact of latency from achieving page-level integrity from 16B-granularity using atomic regions (§5.3.2). Applying the log entry (which we will conservatively make an entire atomic region) takes $1.47 \cdot 0.21 = 0.3$ms. We could be interrupted by the APIC timer, the end of a thread scheduling quantum, or by a page fault from a memory access, of which there are $5 \cdot \log_2(100,000,000) = 120$ (from the Merkle tree accesses in Algorithm 1). In the worst case, we would repeat execution of the atomic region 122 times, resulting in a worst-case additional latency of 36.6ms. Note that this is a (very) loose upper bound and is still below user perceptibility.

Network usage. We measure the network usage of SVR3 running on each enclave type for a 3-replica cluster in Table 10. There is a range of network usage for Server ↔ Server because it depends on how many requests have been batched into a single Raft$^{\circlearrowleft}$ append request. The network usage between servers also depends on the number of servers in the cluster, growing proportionally to $m-1$ for m servers. From a deployment perspective, we are more concerned with the Client ↔ Server bandwidth, which is under 20KB for all enclave types. This is because exchanging more data between the client and the server can become a usability issue for users with limited data plans.

Memory usage. We measured the memory usage of SVR3 on SGX, varying the number of users in the system. Note that we expect the memory usage to be similar for all enclave types, since they are storing the same amount of data for each user. We find that memory usage grows by $\sim$450B/user until we start truncating the log at 100MB and then settles into a steady 170B/user added. At 100 million users, SVR3 uses 18.5GB of memory on each server, which is 185B/user/server.

8.2 End-to-end performance

We measure the end-to-end performance of SVR3 by running a client that stores its secret key by sending a (sequential) request to a server in each enclave cluster. For a more representative deployment, we geographically distribute the SGX cluster (`centralus`, `eastus`, `eastus2`, `southcentralus`, `westus`), the SEV-SNP cluster (`us-central1`, `europe-west3`, `asia-southeast1`, `europe-west4`, `europe-west3`), and the Nitro cluster (`us-east-1`, `us-east-2`, `us-west-1`, `us-west-2`, `eu-north-1`). The performance for recovering a key is almost identical to the performance for storing a key, so we only report the performance for storing a key. We plot the CDF of the latency of these requests in Figure 11a. The average end-to-end latency is 365ms, which is reasonable for a user to wait for a key recovery or key backup request. We plot the breakdown of the latency in Figure 11b. The majority of the latency is from waiting for servers to respond (69.3%), followed by remote attestation with the servers (29.9%).

9 Related work

Secret recovery systems. A number of companies have deployed secret recovery systems using secure hardware: Apple protects user iCloud data using hardware security modules (HSMs) [4, 43], Google protects Android backups using secure microcontrollers [96], and WhatsApp protects message histories using HSMs [98]. WhatsApp runs vanilla Raft [65] on a geographically distributed cluster of HSMs and uses OPAQUE [39] for key recovery. WhatsApp's consensus only requires one round trip between the leader and the replicas while SVR3 requires an extra round of communication (to guarantee safety in the face of rollbacks). Davies et al. analyzed the security of the WhatsApp encrypted backup protocol [24].

Like SVR3, all of these systems use secure hardware to allow a user to recover a cryptographic secret using a low-entropy secret (e.g., a 4-digit PIN). Unlike SVR3, they rely on a single type of secure hardware: the compromise of one secure hardware device can compromise many users' secrets.

Juicebox [88] is a key recovery protocol that distributes trust across one type of secure hardware and multiple trust domains in the traditional manner (across organizations). SVR3 has a simpler protocol that is not a multi-round PAKE as our servers never learn whether the PIN is guessed correctly or not (keys are deleted unconditionally when guesses run out). Secret shares are also stored directly on the servers in Juicebox. Thus, to prevent an attacker who compromises a threshold number of trust domains from reconstructing all the secrets without needing to mount a dictionary attack, they must mix the reconstructed secret with the PIN to create an encryption key that is then used to encrypt the target secret.

SafetyPin [20] is a PIN-based end-to-end encrypted backup system that defends against an attacker that can adaptively compromise some percent of HSMs. While SafetyPin protects against a more powerful attacker model, it requires a comparatively large number of HSMs.

Tutamen [78], Acsesor [11], and CanDID [52] split trust across multiple entities to allow users to recover their secrets (among other operations). Chen et al. [13] use cloud storage for secret recovery. These systems do not use secure hardware; the use of enclaves in SVR3 provides additional security and requires us to design for their limitations (e.g., rollback attacks). CALYPSO [42] also shards user secrets across different entities but, unlike SVR3, uses a blockchain. PreVeil [72] shards secret keys across other peers in a social or work graph, but requires manual setup from the user.

Another line of work has taken a more theoretical approach to the problem of secret key backups. Benhamouda et al. [7] use a proof-of-stake blockchain to allow users to store secrets while protecting against an attacker that can adaptively compromise a percent of the stake. Subsequent work improves efficiency in this model via batching [30].

Orisini et al. [68] also describe a scheme for end-to-end encrypted backups, but in their scheme, the user does not need to remember a PIN or something similar. Instead, users continuously monitor for illegitimate recovery attempts, allowing an honest user to thwart malicious recovery attempts but later recover their backup. While this approach is appealing in that it eliminates the PIN, it does not work for our setting where clients may go offline for extended periods of time.

Multi-party computation and secure hardware. Cryptocurrency wallets protect user secrets by distributing them across hardware enclaves or HSMs [27, 29, 41, 77, 81]. Cryptocurrency wallets are designed to avoid materializing the key in a single location rather than to enable users to recover secrets. Myst provides security by splitting trust across many hardware devices and operations like signing and decryption [54]. More broadly, prior work has examined composing multi-party computation and secure hardware for efficiency [6, 25, 44, 64]. Our use of secure hardware with multi-party computation is tailored to encrypted backups and, while this line of work uses secure hardware to reduce the costs of multi-party computation, we use it to augment the security of the system. In prior work [21], we observed that heterogeneous secure hardware hosted by different clouds can be useful for deploying systems that split user secrets, including encrypted backups, but we had not yet worked through and built out such a deployment.

Rollback prevention in enclaves. There has been a rich line of work on preventing rollback attacks in enclaves. Memoir [69] and Ariadne [86] store a small amount of state inside non-volatile memory (NVRAM) and use that to reconstruct application state during recovery. Both approaches are scoped to single machines, and do not provide availability in the event of a machine permanently failing. ROTE [53] uses a broadcast algorithm across enclaves to maintain a distributed counter, but requires NVRAM to update group membership, whereas we use our Raft$^{\circlearrowleft}$ log to update membership. Additionally, the abstraction that ROTE offers is one of a counter instead of generic log entries. Engraft [97] examines the safety issues of running off-the-shelf consensus inside enclaves. They use an underlying broadcast protocol similar to ROTE to maintain a distributed counter and introduce additional mechanisms to support node recoverability. However, in our setting, we can simply start a new node in the event of a node failure, so we do not need to support node recoverability.

Nimble [3] is a lightweight replication protocol that provides a freshness-guaranteed ledger. The ledger can be used to keep track of the state of untrusted storage, enabling applications that run on enclaves to persist their state to external (untrusted) storage and detect potential rollbacks on that storage. Note that our system is already protected against the class of rollback attacks on external storage described in §1 of [3] because *all data is stored and maintained in memory*. Nimble's threat model does not include physical rollback attacks on the enclave (both endorser and application). However, minimizing SVR3's trusted computing base (TCB) is an interesting and important future direction, and we discuss potential design decisions and open challenges in §10.

TrInc [49] shows that a secure log can be implemented with a secure counter. However, realizing a secure counter on enclaves is difficult. We cannot write PCRs to the TPM from inside an SGX enclave, and additionally, TPMs can limit the speed of counter updates (§6.1.1, [86]). CPU registers are written to the SSA, which can be rolled back. On SGX there is no CPU register where only an enclave can write to it. We are unaware of an (efficient) secure counter primitive on newer enclaves after consulting with Intel.

Consensus protocols. As Dinis et al. [26] point out, rollback behavior can be considered a subset of Byzantine behavior, so the Byzantine fault tolerant (BFT) model is stronger than necessary for our setting. Consequently, Raft$^{\circlearrowleft}$ is lighter weight than BFT flavors of Raft protocols like Tangoroa [19] which requires $O(m^2)$ communication scaling in the number of replicas. The supermajority parameter in Raft$^{\circlearrowleft}$, which increases the quorum size, is comparable to PBFT's [10] Byzantine nodes value. Engraft [97] and RR (TEEMS) [26] address data-sealing (software) rollback attacks. SVR3 not only defends against these data-sealing rollback attacks, but also defends against physical rollback attacks.

10 Discussion

Consensus in the enclave. Nimble [3] is able to maintain a secure log while removing the consensus mechanism from the TCB, and an important future direction for SVR3 would be to similarly minimize its TCB. However, it is not entirely straightforward, and there are interesting design and engineering challenges to address. First, Nimble will need to be hardened against physical rollback attacks, which seems straightforward to do. More significant is that since this log—which contains OPRF secrets—will be held in untrusted storage, it must be encrypted. This has important consequences for our system as we describe below, and addressing them may result in significant additional complexity (and thus increase the TCB).

First, we note that we will need enclaves similar to the ones we have today to handle client requests. These enclaves will now need to share a common encryption key to encrypt and decrypt these log messages. This shared key becomes a new single point of failure for the system. To maintain the forward secrecy we have today due to our use of Noise protocol [71] channels with rekeying between enclaves, it seems the enclaves will need to participate in some sort of continuous group key agreement (CGKA) [2] to rotate the key periodically and on membership changes.

Second, if this new system aims to keep the TCB small by maintaining the database state outside of the enclave, as with Juicebox [88] or WhatsApp [98], then the encryption key for the database becomes another single point of failure, but in this case it is not clear how we can achieve forward secrecy without periodically re-encrypting the entire database. If, on the other hand, we maintain the database in enclave memory, as we do now, then the use of CGKA to protect the encrypted log means that new members of a replica group will not be able to read old log messages to construct the database state. While we have a state transfer mechanism in our current system to handle truncated logs, we will need to refine it to ensure that new members are correctly initialized.

Taken together, we see removal of the consensus mechanism from the TCB as a project that requires careful design and analysis and significant engineering work that adds its own complexity. We note that the consensus protocol is a relatively small (1,541 LOC in C++) and well-understood part of our current codebase, so we need—and hope to find—clear rationale for its removal.

In-memory vs. disk-based storage. While disk-based storage solutions are cheaper than keeping the entire database of key recovery shares in memory, they are more susceptible to rollback attacks because the secrets are taken out of the enclave, and even enable rollback attacks that are software-based and can be performed without physical access.

Data privacy compliance. In general, a multi-cloud deployment may complicate compliance with data privacy laws. The design of SVR3, however, keeps compliance simple since by preventing any user data from being processed by our servers and blocking our administrators from accessing sensitive keys.

Malicious clients. SVR3 provides security guarantees for users using our clients, which we assume are well-behaved. Our client code is open source [55–57], and scrutinized by the community. If the user's client is compromised and malicious (e.g., the user has malware), it can affect the security of that user, but not the security or experience of other users with uncompromised clients.

Honest cloud providers? If we could assume that most cloud operators are honest, then that could change the parameterization of SVR3 (e.g., setting the number of trust domains that can be compromised t to 1), though this would also require assuming that the enclaves were not susceptible to any future vulnerabilities that could be exploited remotely. We would still use enclaves to prevent malicious system administrators from running arbitrary server code.

11 Conclusion

SVR3 demonstrates the potential of systems that provide security through a combination of cryptography and a diverse set of hardware enclaves, without putting trust in any single hardware component. Using different types of enclaves leads to an array of deployment challenges stemming from heterogeneous and shifting attacker models. SVR3 is a powerful defense against the evolving landscape of enclave security: by distributing trust across enclaves, even if a new threat arises in one type of enclave, user secrets are still secure. SVR3 costs \$0.0025/user/year and takes 365ms for a user to recover their key, which is a rare operation.

Acknowledgments. We are very grateful to our shepherd, Jay Lorch, for his detailed feedback, as well as to the collective OSDI reviewers for their suggestions, which greatly improved the presentation of this paper. We thank Trevor Perrin for the pointer to the PPSS construction [37]. Mark Johnson helped develop the early stages of this project. Ravi Khadiwala contributed significantly to SVR3's implementation. We thank Natacha Crooks, Christopher Fletcher, Jack Humphries, and Sky security students for their helpful feedback. The UC Berkeley authors are supported by NSF Graduate Research Fellowships, a Microsoft Ada Lovelace Fellowship, and gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, MBZUAI, Samsung SDS, SAP, Uber, and VMware.

References

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS*, 2016.

[2] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *TCC*, 2020.

[3] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback protection for confidential cloud services. In *OSDI*, 2023.

[4] Apple. iCloud Keychain security overview, 2021. https://support.apple.com/guide/security/icloud-keychain-security-overview-sec1c89c6f3b/.

[5] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS*, 2011.

[6] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In *FC*, 2017.

[7] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *TCC*, 2020.

[8] Brian N Bershad, David D Redell, and John R Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS*, 1992.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[10] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.

[11] Melissa Chase, Hannah Davis, Esha Ghosh, and Kim Laine. Acsesor: A new framework for auditable custodial secret storage and recovery. *Cryptology ePrint Archive 2022/1729*, 2022.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, 2019.

[13] Long Chen, Ya-Nan Li, Qiang Tang, and Moti Yung. End-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage. In *USENIX Security*, 2022.

[14] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security*, 2021.

[15] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[16] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. In *ISeCure*, 2011.

[17] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. Secret key recovery in a global-scale end-to-end encryption system. *Cryptology ePrint Archive 2024/887*, 2024.

[18] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *USENIX Security*, 2023.

[19] Christopher Copeland and Hongxia Zhong. Tangaroa: a Byzantine fault tolerant Raft, 2016. https://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf.

[20] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with human-memorable secrets. In *OSDI*, 2020.

[21] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *HotNets*, 2022.

[22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[23] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious pseudorandom functions (OPRFs) using prime-order groups. https://www.ietf.org/id/draft-irtf-cfrg-voprf-21.html.

[24] Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. *Cryptology ePrint Archive 2023/843*, 2023.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security*, 2014.

[26] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. RR: A fault model for efficient TEE replication. In *NDSS*, 2023.

[27] Fireblocks. https://www.fireblocks.com/platforms/mpc-wallet/.

[28] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[29] Gemini. Cold storage, keys & crypto: How Gemini keeps assets safe. https://www.gemini.com/blog/cold-storage-keys-crypto-how-gemini-keeps-assets-safe.

[30] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *PKC*, 2022.

[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE S&P*, 2018.

[32] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[33] Feng Hao and Paul C van Oorschot. SoK: Password-authenticated key exchange–theory, practice, standardization and real-world lessons. In *AsiaCCS*, 2022.

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[35] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via rowhammer attack. In *SysTEX*, 2017.

[36] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, 2014.

[37] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, 2016.

[38] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS*, 2017.

[39] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[40] Your Keybase account. https://book.keybase.io/account.

[41] Knox. Knox custody. https://www.knoxcustody.com/security.

[42] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. *Cryptology ePrint Archive 2018/209*, 2018.

[43] Ivan Krstic. Behind the scenes with iOS security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[44] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *IEEE S&P*, 2020.

[45] Leslie Lamport. Specifying systems: The TLA+ language and tools for hardware and software engineers. 2002.

[46] Ledger. How Ledger device generates 24-word recovery phrase. https://support.ledger.com/hc/en-us/articles/4415198323089-How-Ledger-device-generates-24-word-recovery-phrase, November 2023.

[47] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*, 2020.

[48] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[49] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[50] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. The deployment dilemma: Merits & challenges of deploying MPC, 2023. https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html.

[51] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/.

[52] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, Sybil-resistance, and accountability. In *IEEE S&P*, 2021.

[53] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *USENIX Security*, 2017.

[54] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. A touch of evil: High-assurance cryptographic hardware from untrusted components. In *CCS*, 2017.

[55] Signal Messenger. Signal Android client. https://github.com/signalapp/Signal-Android.

[56] Signal Messenger. Signal desktop client. https://github.com/signalapp/Signal-Desktop.

[57] Signal Messenger. Signal iOS client. https://github.com/signalapp/Signal-iOS.

[58] Meta. End-to-end encryption on Messenger explained, 2024. https://about.fb.com/news/2024/03/end-to-end-encryption-on-messenger-explained/.

[59] Microsoft. Bitlocker whitepaper Windows 10. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_firmware/pdf_3/Bitlocker_White_Paper_Windows_10.pdf, 2018.

[60] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, 2018.

[61] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[62] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE S&P*, 2020.

[63] Nitro secure module. https://github.com/aws/aws-nitro-enclaves-nsm-api/tree/v0.4.0.

[64] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[65] Diego Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014.

[66] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[67] Open Enclave SDK. https://github.com/openenclave/openenclave/tree/v0.19.0.

[68] Chris Orsini, Alessandra Scafuro, and Tanner Verber. How to recover a cryptographic secret from the cloud. *Cryptology ePrint Archive 2023/1308*, 2023.

[69] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *IEEE S&P*, 2011.

[70] Trevor Perrin. KEM-based hybrid forward secrecy for Noise. 2018. https://github.com/noiseprotocol/noise_hfs_spec/blob/master/output/noise_hfs.pdf.

[71] Trevor Perrin. The Noise protocol framework. 2018.

[72] PreVeil: Encrypted email and file sharing. https://www.preveil.com/.

[73] Protocol buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

[74] Proton Mail. https://proton.me/mail.

[75] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *IEEE S&P*, 2021.

[76] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *SOUPS*, 2019.

[77] Riddle&Code. Hardware security modules vs. secure multi-party computation in digital asset custody: The drawback of choosing just one and what happens when you combine them. https://www.riddleandcode.com/blog-posts/hardware-security-modules-vs-secure-multi-party-computation-in-digital-asset-custody.

[78] Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald. Tutamen: A next-generation secret-storage platform. In *SoCC*, 2016.

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[80] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[81] Sepior. https://sepior.com/products/advanced-mpc-wallet.

[82] Pavitra Shankdhar. Popular tools for brute-force attacks. https://resources.infosecinstitute.com/topics/hacking/popular-tools-for-brute-force-attacks/, 2020.

[83] Rob Shirley. Internet security glossary, version 2. https://datatracker.ietf.org/doc/html/rfc4949.

[84] Signal Messenger. Secure Value Recovery Service v2/3. https://github.com/signalapp/SecureValueRecovery2.

[85] Signal Messenger. https://signal.org/.

[86] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[87] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security*, 2017.

[88] Nora Trapp. Key to simplicity: Squeezing the hassle out of encryption key recovery, 2024. https://www.juicebox.xyz/blog/key-to-simplicity-squeezing-the-hassle-out-of-encryption-key-recovery.

[89] Anna Trikalinou and Dan Lake. Taking DMA attacks to the next level. 2017.

[90] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[91] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, 2020.

[92] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

[93] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[94] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[95] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. *arXiv preprint arXiv:2006.13353*, 2020.

[96] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[97] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Engraft: Enclave-guarded Raft on Byzantine faulty nodes. In *CCS*, 2022.

[98] WhatsApp. Security of end-to-end encrypted backups, 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[99] Kyle Wiggers. Apple launches end-to-end encryption for iCloud data. *TechCrunch*, 2022.

[100] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Flock: A Framework for *Deploying* On-Demand Distributed Trust

Darya Kaviani[1]* Sijun Tan[1]* Pravein Govindan Kannan[2] Raluca Ada Popa[1]

[1] *UC Berkeley* [2] *IBM Research*

Abstract

Recent years have exhibited an increase in applications that *distribute trust* across n servers to protect user data from a central point of attack. However, these deployments remain limited due to a core obstacle: establishing n distinct trust domains. An application provider, a *single* trust domain, cannot directly deploy *multiple* trust domains. As a result, application providers forge business relationships to enlist third-parties as trust domains, which is a manual, lengthy, and expensive process, inaccessible to many application developers.

We introduce the *on-demand distributed-trust architecture* that enables an application provider to deploy distributed trust automatically and immediately without controlling the other trust domains. The insight lies in *reversing* the deployment method such that each user's client drives deployment instead of the application provider. While at a first glance, this approach appears infeasible due to cost, performance, and resource abuse concerns, our system Flock resolves these challenges. We implement and evaluate Flock on 3 major cloud providers and 8 distributed-trust applications. On average, Flock achieves 1.05x the latency and 0.68-2.27x the cloud cost of a traditional distributed-trust deployment, without reliance on third-party relationships.

1 Introduction

Existing systems typically suffer from a central point of attack: an application provider holding many users' private data becomes the target of data breaches [117]. As a result, an increasing number of applications are using *distributed trust* [35, 38, 46, 47, 66, 67, 72, 77, 108, 161, 177–179, 195, 198]. This powerful paradigm avoids a central point of attack by distributing the users' sensitive data among n parties to protect its confidentiality or integrity. A typical requirement is that these n parties are in different *trust domains*, each of which corresponds to a distinct organization to ensure that they are controlled by different entities. Fig. 1 illustrates the stakeholders of this setting: the application provider, its users, and $n-1$ other trust domains. Even if $n-1$ out of n parties are compromised, the sensitive data remains secure: an attacker would have to breach all n parties to compromise the sensitive data. Various cryptographic tools rely on distributed trust, such as secure multi-party computation (MPC) [112, 128, 147, 163, 164, 202, 213, 219] and two-party private information retrieval (PIR) [105, 116, 124, 146].

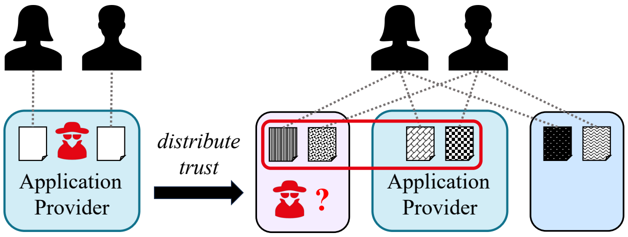


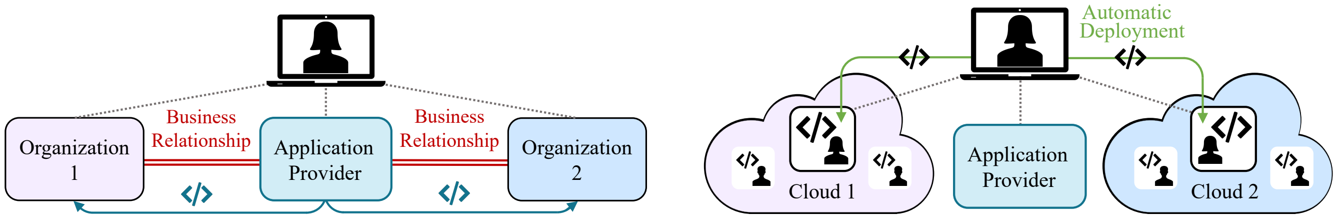
Figure 1: Secret-sharing data over 3 trust domains: breaching 2 trust domains reveals nothing.

Recent years have exhibited an increased adoption of distributed trust by application providers who aim to *protect their users' data* [161] (§6), including Signal [178, 179], Coinbase [35, 177], Fireblocks [46], Google [108], Apple [108], Meta [198], and J.P. Morgan [195]. For example, Signal's secure value recovery project aims to enable users to securely back up their private keys through distributed trust [178, 179]. Likewise, MPC wallets [35, 38, 42, 46, 47, 66, 72, 77, 97, 134, 177], including Coinbase [35, 177] and Fireblocks [46], secure billions of dollars by distributing their users' private keys and using MPC for signing [196].

The deployment challenge. Despite this interest, the adoption of distributed-trust applications remains limited. Recent works [130, 178, 198] discuss a core challenge: the *difficulty of deploying n distinct trust domains*. Indeed, the application provider must find $n-1$ organizations in different trust domains, who are willing to run the provider's workload while restricting access to anyone, including the provider itself. These organizations must offer sufficient availability, security, fault tolerance, logging, swift recovery, and must have a credible reputation in the user community—criteria that have empirically been challenging to satisfy [178]. Moreover, such business relationships are costly and require both time and manual effort to set up. While large corporations were able to forge such partnerships [108, 198], this is a barrier to entry for application developers [178] who lack the same resources.

For clarity, consider the **running example** of digital asset

*Equal contribution



(a) *Traditional Distributed Trust*: The application provider forms manual, costly relationships with third-parties, who deploy code.

(b) *On-Demand Distributed Trust*: The user's client automatically deploys code to $n-1$ clouds, which also have other user deployments.

Figure 2: Deployment workflow in traditional vs. on-demand distributed trust.

custody (although our work applies to a wide range of applications, as discussed in §4). Cryptocurrency [119, 189] users exchange digital assets by signing transactions with their private keys. A compromised private key can be used to steal any assets in the user's wallet [101, 118, 142, 184, 203]. This is why wallets like Coinbase [35, 177] and Fireblocks [46] secret-share the private key among n different parties. When Alice initiates a transaction, the n parties engage in MPC, inside which they reconstruct her key and sign the transaction. Although their desired design is to secret-share among multiple entities, many wallets only secret-share between the application provider and the client due to the difficulty of setting up other trust domains [178].

More generally, many academic papers on distributed trust simply assume the presence of n servers in different trust domains [126, 127, 131, 132, 138, 149, 165, 211], but do not offer guidance on how to establish such deployments in practice. In this paper, we address the following systems challenge with deploying distributed trust:

> How can an application provider, which is inherently a *single* trust domain, deploy a *multi-trust-domain* system?

To address this challenge, we propose the **on-demand distributed-trust** architecture, which enables an application provider to offer distributed-trust services to its users *automatically*, *immediately*, and *without* requiring third-parties. This is the first architecture that removes the burden of setting up cumbersome, manual business relationships with $n-1$ parties. We provide an intuition and overview in §1.1. Our second contribution is the design and implementation of *Flock*,[1] a system that realizes our on-demand distributed-trust architecture across major cloud providers. Flock enables an application provider to setup $n-1$ trust domains on $n-1$ cloud providers without the application provider being able to control the deployment. A straightforward instantiation of the on-demand architecture suffers from significant scalability and security issues. Flock overcomes these challenges with two additional technical contributions that are reusable beyond Flock: the *Flock Relay* and a *three-tier authentication protocol*, both overviewed in §1.2.

1.1 On-Demand Distributed Trust

To understand our approach and its challenges, first consider a natural strawman of using n reputable cloud providers as trust domains. Recent years have exhibited a spike in multi-cloud services [4, 11, 63, 94] and multi-cloud applications [12, 135, 153, 156, 193, 204, 218]. While *accounts* within a *single* cloud can be accessed by the cloud's administrators, *distinct* cloud providers have their own data centers, storage, compute resources, networking solutions, and, crucially, administrators. While the clouds are indeed distinct trust domains, this approach suffers from a central point of attack: the application provider that deploys VMs to each cloud controls them all, reducing the system to a single trust domain. As we discuss in §6, some proposals attempt to approximate trust domains with hardware enclaves [130, 179], but enclaves are vulnerable to side-channel attacks [115, 175, 188, 209] that allow the application provider to once again fully control the deployment. Hence, we seek an approach to deploying $n-1$ trust domains that the application provider cannot control, without depending on trusted hardware.

The primary insight of on-demand distributed trust is a paradigm shift in the deployment approach: Instead of the application provider deploying the n parties to distinct clouds, *each user* drives the deployment for *their own data*. At a first glance, this approach appears infeasible with respect to ease-of-use and cost because every user has a *separate* deployment. From a security standpoint, though, it is fitting: a user Alice is the trusted owner of her *own* sensitive data. Hence, she can deploy VMs across $n-1$ major clouds, with the application provider as the n-th party. Alice can secret-share her sensitive data—for example, her private key for digital asset custody—across these n VMs because no other party can control all of them, not even the application provider. Fig. 2 compares the deployment workflow of traditional and on-demand distributed trust.

The on-demand architecture enables a wide range of distributed-trust applications, but not all of them:

> Flock can support all applications where every distributed-trust computation takes as input only data that is owned by *exactly one user* or is public.

[1] Multi-species bird flocks may not always trust each other, but flock together among the clouds to increase the likelihood of detecting predators [62].

We show that Flock can support 8 types of distributed-trust applications, including secret key recovery for end-to-end encrypted systems (as in Signal [179]), password managers, digital asset custody (as in Coinbase [35], Fireblocks [46] and other MPC wallets [38, 42, 47, 66, 72, 77, 97, 134, 177]), certificate authority signing, code signing, two-server private information retrieval, and data rollback protection. These applications demonstrate different aspects of Flock's expressivity: they enable 5 major cryptographic modules, which showcase protection for data confidentiality, data integrity, data-sharing, or query privacy on a public database. An example of an application that Flock does not support is privately training a machine learning model over *all* users' private data because the input to the training process is the combined data of all users that no *individual* user owns and is not public.

1.2 Summary of Techniques

We now summarize Flock's techniques. A careful reader may be concerned about the cost of the on-demand approach. Continuously running one VM *per* user *per* cloud would be prohibitively expensive and shutting them off intermittently would introduce significant startup time.

We identify *serverless computing* [18, 24, 52, 102] as the most fitting paradigm (§3.1) for Flock. Their "pay-as-you-go" model means that we can invoke a serverless instance exclusively when the user runs an operation, and incur no cost when the user is idle. For example, in the digital asset custody application, Alice's serverless instances only run when Alice wants to perform a transaction. Further, executing a serverless instance on-demand is fast, unlike typical VM boot times. Thus, Flock offers a **cross-cloud serverless system for secure computation (§3.1)**, which runs sophisticated cryptographic libraries in serverless across clouds. A challenge is that serverless offerings cannot innately form peer-to-peer connections [150, 214] because they are publicly inaccessible and ephemeral. To enable end-to-end secure cross-cloud serverless networking, we design a relay that leverages the application provider to connect the serverless instances without trusting the provider with the contents of the communication. To achieve this, we introduce a two-phase TLS establishment protocol that only requires a *single* TCP connection per serverless instance, which is used for both authentication and end-to-end TLS. By reducing secure message-forwarding to copying bytes across sockets, the relay achieves 27x higher throughput (Table 4) than the current best potential approach. We expect the relay to have independent utility in any application that requires end-to-end secure cross-domain (e.g. cross-cloud, cross-region) serverless communication.

Flock's **automatic deployment mechanism (§3.4)** empowers *regular* users to *automatically* setup cross-cloud accounts and serverless deployments without being exposed to underlying cloud-level intricacies. First, each user's Flock client automates multi-cloud account creation by filling in the corresponding forms for the user through the webpage automation framework Playwright [65]. Second, the Flock client conducts programmatic deployment through cloud-provided APIs. The user experience of a Flock application is comparable to that of a regular application. Users will not have to conduct manual cloud registration or serverless deployment per cloud, and the only difference is that users may need to complete $n-1$ authentication steps for cloud registration (e.g. SMS, email).

To secure this deployment, Flock contributes a **three-tier authentication protocol (§3.2)**, which safeguards against the impersonation of a user Alice, her deployments, or the application provider. Our new setting of user-driven distributed-trust deployment introduced new attack vectors, requiring a novel design for authentication: *How can an application provider secure a deployment they do not control?* We first identified the required security "checkpoints" across three tiers—cloud, network, and application—leading us to design a unified protocol spanning these layers. At the cloud level, fine-grained access keys prevent unauthorized users from invoking Alice's serverless instances. At the network level, a secure deployment protocol guards the communication amongst Alice and her "flock." At the application level, Alice's "flock" must authenticate her before conducting operations on her sensitive data. However, it is onerous for each user to authenticate n times, once *per* party. To avoid this, we identify MPCAuth [206] as particularly well-suited in this scenario. MPCAuth enables a user to perform the usual work of authenticating to a *single* "logical" server—which is an MPC of the n servers—with the same security as authenticating to n servers independently.

User-centered deployment introduces a new axis of challenges: Flock should allow the provider to manage **billing without controlling users' cloud instances (§3.3)** or exposing the provider to resource abuse. We use cloud billing infrastructure to prevent malicious users from draining application provider funds and cloud access keys to prevent an attacker from wasting serverless compute resources before they are detected by application-level authentication.

1.3 Evaluation Summary

We implement and evaluate Flock (§5) across three major cloud providers: Amazon Web Services, Azure, and Google Cloud Platform. We have also successfully deployed Flock to IBM Code Engine. When compared to the traditional distributed-trust setup (Fig. 2a), Flock has 1.05x latency and 0.68-2.27x the cloud cost, averaged over all 5 modules. This value does not account for the traditional method's additional cost of business relationships with the $n-1$ third-party organizations (e.g. employee salaries, operational costs)—expenses that do not exist in Flock. Moreover, Flock achieves this without the manual and time-consuming process of identifying and setting up other organizations as trust domains. By removing this deployment barrier, we believe that Flock can foster a new wave of adoption for distributed trust.

2 Threat Model & Security Guarantees

System model. An application provider seeking to offer distributed-trust security to its users invokes the Flock API in its client and server code. Users install an *application client* on their device. The application provider runs the *application server*, which we consider to be a logical server (even if it comprises of multiple physical servers). We refer to the n distinct trust domains that execute the distributed-trust modules as *parties*. The application provider constitutes one party, and the $n-1$ clouds constitute the other $n-1$ parties.

Security guarantees. Flock is not a specific cryptographic scheme or application, but a *system for deploying* distributed trust across the clouds for a variety of applications with different threat models. The guarantee of the on-demand distributed-trust architecture is that each of the n parties are deployed independently of each other, without any one party being able to control the others. Hence, none of these parties are a central point of attack, and crucially, the application provider cannot control the deployments in the $n-1$ clouds. To provide this guarantee, Flock relies on the security mechanisms of each cloud in a black-box manner. As long as cloud i upholds its guarantees, party i stands as an uncompromised trust domain in the Flock deployment.

When running a distributed-trust application App using Flock, the resulting security guarantees are a *combination* of the guarantees of App and Flock, and often, the weaker of the two. The Flock system and modules provide the strong guarantee of malicious security against $n-1$ out of n compromised parties. In particular, an attacker cannot see any secret data distributed across the parties or tamper with the integrity of sensitive operations. Hence, if App *also* provides malicious security, so does the overall App-Flock deployment. If, on the other hand, App provides the weaker semi-honest security, so does the overall App-Flock deployment.[2]

Availability. While the application provider is not trusted with confidentiality and integrity, it is trusted for availability because it is the entity that wishes to provide this service. We also assume that the clouds are available given their service-level agreements [20,51,78]. In each cloud, Flock uses cloud services that are fault-tolerant. If, despite this, the provider or a cloud are not available, Flock does not offer availability.

Application code. Like prominent end-to-end encrypted applications [69,79,86,95,98] and blockchains [111,119,189], Flock assumes that the application client code is not compromised and that it is open source and community-scrutinized. Likewise, Flock is open source (§5.1). Flock's focus is to protect against attacks to the application servers. Application servers are a prolific target of attack because they aggregate data across all users. They can also read user data and alter server execution unchecked, which is more difficult to perform with openly scrutinized client code.

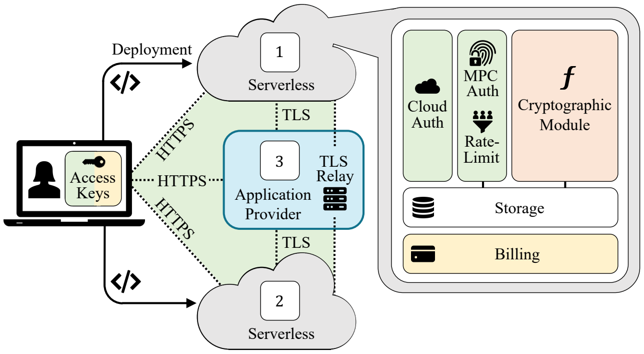


Figure 3: The Flock system architecture. *Client:* Programmatic deployment. *Authentication (green):* Cloud-level via access keys, network-level via TLS, and application-level via MPCAuth. *Resource Protection (yellow):* Unauthorized client abuse is prevented by access keys, while user abuse is prevented through billing. *Cryptographic modules (orange).*

Compromised client devices of a user do not affect the security of *another* user in Flock. For the same user, If Alice's device is compromised *during an active Flock session*, Flock does not provide security guarantees for Alice. This is not specific to Flock, and is the case for many distributed-trust applications. For example, Signal's SVR [79,179] saves Alice's private keys on her device and distributes them among parties for the purpose of backup. However, if Alice is logged out (and thus not in an active session) during the compromise of her device, Flock's security guarantees remain. Many apps, like password managers and digital wallets, log users out after sessions to bolster security upon a device compromise. Flock implements sessions and removes each user's sensitive key material from the client when the user is logged out.

Authentication (§3.2). Users in Flock authenticate through multiple factors, e.g. with email and SMS on cloud accounts, or PIN and U2F for MPCAuth. Naturally, Flock only protects the data of users with uncompromised authentication factors.

Resource protection (§3.3) in Flock prevents malicious users from draining compute resources and cloud funds from the application provider or denying its service.

3 Flock's System Design

Architecting an on-demand distributed-trust system poses several challenges along the dimensions of cost-efficiency, networking, authentication, resource protection, deployment, and registration. In this section, we describe how we address each challenge in building Flock. Fig. 4 illustrates the Flock system architecture.

[2] The on-demand distributed-trust architecture also supports applications with t out of n security for a threshold $t < n$, but our current implementation only supports $t = n$.

3.1 Serverless Architecture

As discussed in §1.1, the straightforward method of implementing on-demand distributed trust is having *each user* deploy one always-on VM on each of $n-1$ clouds. Despite the infrequency of operations like secret recovery due to device loss, the clouds will charge constantly for each VM. A natural strawman is to have the user turn off each VM upon operation completion and reactivate them as needed. Unfortunately, the user would incur *minutes* of additional latency for VMs to boot *per* operation. This delay is prohibitive for applications like password managers, where users frequently invoke Flock.

We observe that the *serverless* computing paradigm [102, 176, 180] alleviates this problem by charging only for active use with a "pay-as-you-go" model [176]. Developers upload code that exclusively consumes resources at execution [102, 180]. Serverless instances are *event-driven*, so they can be triggered with minimal start-up time through a programmable HTTPS interface [176]. The most common serverless compute offerings are serverless functions [18, 24, 50], where a client triggers an API query, it is validated by the cloud provider, and a previously deployed function is invoked in an isolated environment [73–75, 176, 182]. However, basic serverless functions [24, 50] do not naturally support multi-language codebases or runtimes for programming languages that are commonly used to implement cryptographic tools, such as C++. This makes them inconvenient for porting existing cryptographic frameworks and codebases in Flock. Instead, we turn to serverless *containers*, which are light-weight, standalone executable software packages that include the code, runtime, and system libraries. Containers can support multi-language codebases and any runtime, and are offered by AWS Lambda [18] and Google Cloud Run [52].

MPC requires high interactivity, often with one party awaiting another's response. Unfortunately, serverless instances commonly communicate via services like cloud storage, which is prohibitively slower and pricier than direct networking [107, 150]. The challenge is that serverless instances possess private IPs under unique network address translations (NATs), so they cannot accept incoming network connections. Serverless offerings that expose public IP addresses [16, 22] are intended for long-running workloads, and therefore suffer significant coldstart delays [3], and even charge for a minute-long minimum runtime for AWS Fargate [16].

Several works employ NAT traversal and hole-punching to facilitate serverless communication [137, 139, 186, 214], but this method is not robust since it relies on a cloud provider's NAT configurations, which are prone to change. Both Lambda and Google Cloud Run only support cross-cloud hole-punching through NAT gateways and virtual private clouds [29] with impractical per-user cost. Instead, prior systems have facilitated serverless communication through a central relay [140, 141, 210], but do not consider security. A secure relay-based approach for connecting publicly inaccessible endpoints is Wireguard [96] over Tailscale DERP relay servers [37]. As we show in §5.4, this setup incurs significant overhead since Wireguard conducts per-packet encryption and must redundantly layer TLS over Wireguard to supplement it with mutual authentication. Also, Wireguard does not use a federally approved encryption protocol [5], unlike TLS.

3.1.1 Secure Cross-Cloud Serverless Networking

To resolve this challenge, we architect a NAT-independent Flock Relay protocol for secure serverless networking at the transport layer (L4). To deploy the relay, we employ the help of the application provider for availability without trusting it otherwise. The application provider runs a multi-user relay that connects serverless instances by accepting their *incoming* connections, then securely routes messages between them with authentication and end-to-end encryption. The relay observes only message lengths, which do not reveal the private user inputs because of the oblivious nature of MPC. Hence, the provider, though capable of barring the *availability* of the relay, cannot compromise data confidentiality or integrity.

To facilitate secure serverless-to-serverless communication, the Flock Relay must authenticate each serverless instance to ensure that the correct endpoints connect to one another, and facilitate their communication without serving as a central point of attack. While standard TLS connections are established between two endpoints directly, the Flock Relay needs to facilitate end-to-end TLS establishment between two authenticated endpoints.

An insecure strawman for establishing serverless-to-serverless TLS is the following: (1) After a serverless instance initiates a serverless-to-relay connection, the relay verifies the serverless instance and provides it with an authentication token. In future connections, the token will inform the relay that it has previously authorized the serverless endpoint. (2) The serverless instance sends the token to the relay to authorize the end-to-end serverless establishment. Crucially, however, the second step would require the token to be sent in *plaintext*. A passive network eavesdropper could use the token to impersonate valid connections in the future.

Two-phase connection establishment. To ameliorate this issue, we architect our relay to connect serverless instances with the *same* sockets that were already authorized. The Flock Relay executes two phases of TLS establishment to form a *single* end-to-end TLS session, as we show in Fig. 4.

(1) *Serverless-to-relay* (S2R): Every pair of serverless instances s_1 and s_2 each initiate an independent TLS connection with the relay. The relay needs to maintain access control to ensure that only the instances within a single user's "flock" can connect to one other. Thus, the TLS handshake in this phase authenticates the serverless instances. Over the S2R TLS session, each instance notifies the relay of the ID of the serverless instances it wishes to connect to, which hides these IDs from the public Internet. Next, the relay downgrades its

TLS connection with s_1 and s_2 to TCP, and begins forwarding messages between the two TCP sockets.
(2) *Serverless-to-serverless* (S2S): Every pair of serverless instances in a user's "flock" performs a TLS handshake over the TCP connection obtained after the S2R phase, setting up the S2S TLS session between them. To send a message to s_2, s_1 sends the relay a TLS-protected message under the S2S session, which the relay forwards to s_2.

For both S2R and S2S, we use mTLS (Mutual TLS), which enables *mutual* authentication. We also use the relay to connect serverless instances to the application provider (as if it were another serverless instance), for ease of implementation. We assume the user's application client obtains the relay's destination from the application provider.

3.1.2 Flock Relay Certificate Issuance

We now discuss how Flock sets up TLS certificates to ensure all communication occurs with intended parties. The client maintains hardcoded public keys for the application server and relay. For the serverless instances to verify each other in S2S sessions, an observation is that a user Alice is trusted within *her* "flock" and can therefore serve as its certificate authority. Alice's client creates a public-private keypair for each party in her deployment (serverless instances and application provider) and signs a certificate for each of their public keys. Each party stores its certificate and Alice's public key, allowing parties to mutually verify one another.

S2R sessions enlist the application provider as a certificate authority. Indeed, this phase of TLS only exchanges information about message recipients, which must be hidden from the public Internet, but visible to the relay. Overall, the Flock Relay is responsible for managing a public-private keypair for signing, engaging in a setup protocol for certificate issuance for each user, verifying these certificates per-invocation, and facilitating message-forwarding between serverless instances.

For client-to-serverless connections, upon invocation, Alice's client contacts each Flock instance and the application provider via HTTPS. Alice provides parameters including the IP address and port of the relay, as well as query-specific input, so the provider can redeploy or load-balance the relay without requiring Alice to redeploy her instances. Google Cloud Run [52] and AWS Lambda [18] URLs offer HTTPS with trusted CAs, so Alice knows that she is contacting the intended instances.

Reissuance. To prevent an attacker that steals Alice's device from issuing certificates, Alice deletes her certificate issuance secret key post-deployment. Because reissuance is uncommon, she can reauthenticate to each cloud, regenerate keypairs (including her own), and reissue. If the relay updates its public-private keypair, it must reissue a certificate for each user. The application client will be updated with this new relay public key and certificate.

The convenient aspect is that Alice can update her parties' certificates and public keys (both her own and the relay's) without redeploying the entire codebase because they are stored in cloud-provided secret managers [34, 91]. This process is more lightweight than a full-fledged application software update, which necessitates serverless redeployment. For a consistent certificate keypair, one can use Flock's secret recovery (§4.1) or signing (§4.2) to store Alice's secret key.

3.1.3 Flock Relay Protocol

We now detail the Flock Relay protocol, shown in Fig. 4. For simplicity, we only list parameters in certificates or message tuples that are specific to our protocol, but a deployment must contain all the other standard parameters and defenses.

Per-Relay Setup:

1: The application provider generates a keypair $(\mathsf{PK}_r, \mathsf{SK}_r)$, which is used to self-sign a certificate for the relay $\mathrm{RelayCert}_r = \mathrm{GenerateCert}(\mathsf{SK}_r; r, \mathsf{PK}_r)$.
2: The application provider deploys the relay with $(\mathsf{PK}_r, \mathsf{SK}_r, \mathrm{RelayCert}_r)$.
3: Relay listens for new users at RelayUserTarget and for serverless connections at RelayTarget.

Per-User Setup:

1: Alice generates $(\mathsf{PK}_u, \mathsf{SK}_u)$.
2: For each party $i \in \{1, \ldots, n\}$, Alice generates $(\mathsf{PK}_i, \mathsf{SK}_i)$ and certificate $\mathrm{E2ECert}_i = \mathrm{GenerateCert}(\mathsf{SK}_u; i, \mathsf{PK}_i)$.
3: Alice locally deletes SK_u.
4: Alice sends each PK_i to the relay at RelayUserTarget.
5: Relay generates the relay-specific user certificate $\mathrm{RelayCert}_i = \mathrm{GenerateCert}(\mathsf{SK}_r; \text{"Alice"}.i, \mathsf{PK}_i)$.
6: Relay sends $(\mathsf{PK}_r, \mathrm{RelayCert}_i)$ to Alice for each party i.
7: In each serverless deployment for party i, Alice embeds: $(\mathsf{PK}_u, \mathsf{PK}_r, \mathrm{E2ECert}_i, \mathrm{RelayCert}_i, \mathsf{PK}_i, \mathsf{SK}_i, i)$.

Per-Invocation Protocol (Fig. 4)**:**

1: Alice invokes each s_i using HTTPS, with the parameters $\{\mathrm{PartyID} : i, \mathrm{RelayTarget} : (\mathrm{IP}, \mathrm{Port})\}$.
2: Each s_i loads $(\mathsf{PK}_i, \mathsf{SK}_i, \mathsf{PK}_r, \mathrm{RelayCert}_i, \mathsf{PK}_u, \mathrm{E2ECert}_i)$.
3: To establish an S2R session, each s_i performs an mTLS handshake with the relay, in which PK_r is used to verify $\mathrm{RelayCert}_r$ is from the intended relay and $\mathrm{RelayCert}_i$ is from Alice's i-th party.
4: Over S2R, s_1 sends the relay "2" as its intended destination and s_2 sends the relay "1".
5: Over S2R, the relay arbitrarily assigns s_1 as "TLS Server" and s_2 as "TLS Client," declaring the assignment to both.
6: Over S2R, the relay sends s_1 (TLS Server) an SSL_shutdown OpenSSL message to close the SSL connection. s_1 confirms completion and starts listening on the same socket for a future TLS connection request.

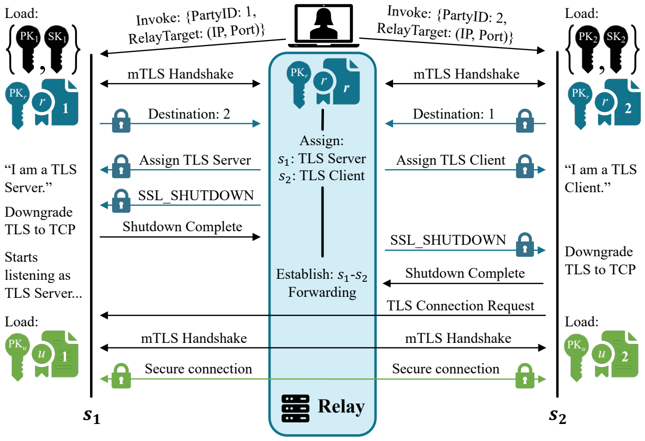


Figure 4: Flock Relay Per-Invocation Protocol. Certificates denote the signing entity in the circle. Teal denotes serverless-to-relay (S2R); green denotes serverless-to-serverless (S2S).

7: Over S2R, the relay sends s_2 (TLS Client) an SSL_shutdown message. When s_2 confirms, the relay establishes s_1-s_2 forwarding over the pair of sockets.

8: To establish S2S, s_2 sends s_1 a TLS connection request through the relay. s_1 and s_2 engage in an mTLS handshake, in which they verify the other's E2ECert$_i$ with PK_u. s_1 and s_2 now share an end-to-end TLS connection.

Hence, the Flock Relay only requires a *single* TCP connection per serverless instance, which is used for *both* authentication (S2R) and end-to-end TLS (S2S). By reducing secure message-forwarding to merely copying bytes across sockets, the relay achieves high throughput, as we show in §5.

3.2 Three-Tier Authentication

Authentication in Flock differs from a traditional system because of the three-layered nature of Flock's design: Alice must authenticate to her cloud deployments, their network sessions, and their running application to be able to execute sensitive operations. We have already described Flock's network-level authentication via certificates in §3.1.2. We now present the application-level and cloud-level authentication in Flock.

3.2.1 MPCAuth for Application-level Authentication

To perform a sensitive operation, Alice needs to authenticate to her serverless instances and the application provider. The natural approach is for her to run multi-factor authentication with each one of these parties. In an application with m authentication factors, Alice must authenticate m times for each party, which is burdensome. For example, Alice must input her password $n = 3$ times, perform 3 U2F authentications or lookup 3 emails with security codes, amounting to $n \times m$ total authentications. For applications like cryptocurrency transaction signing (§4.2) or password retrieval during web browsing (§4.1), such repetitive tasks are on the critical path.

Instead, we employ a recent cryptographic protocol, MPCAuth [206], as a black box. MPCAuth enables users to authenticate once, achieving the security of n distinct authentications. At a high level, MPCAuth performs an MPC computation between the n parties to simulate a "trusted server inside MPC" to which the user authenticates. This imaginary server, an amalgamation of the n parties, ensures that as long as one server remains honest, authentication proceeds correctly. A user seeking to trigger an operation will authenticate to the n serverless containers using their m pre-configured authentication factors. If the user successfully authenticates, the sensitive task is executed. The user's experience is unchanged: Alice authenticates to one logical server, when in fact, she is authenticating to all n servers via MPCAuth.

Any factors supported by MPCAuth can be integrated into Flock, including PIN, passcode, U2F, email, SMS, server-side biometrics, and security questions. Distributed-trust applications like Signal's SVR employ PIN because it does not rely on an outside provider like email or SMS. U2F keys are also a common choice because the authentication secret resides on separate hardware. Flock currently integrates U2F, PIN, and passcode. MPCAuth does *not* require cloud support because it is embedded directly in the Flock deployment.

Supplementing MPCAuth. We remark that using MPCAuth alone is insufficient: it does not offer protections for the cloud tier or our more complex network tier because in MPCAuth, the n servers are fixed and known. Supporting n ephemeral, cross-cloud, user-owned instances introduces attack vectors at the cloud and network layers. To secure these layers, Flock ensures serverless instances are invoked by authorized users (§3.2.2) and have authorized network connections (§3.1.2) per invocation.

Rate-limiting is essential for thwarting brute force attacks in distributed-trust applications, particularly those using low-entropy PINs. Flock supplements MPCAuth with a rate-limiting protocol that tracks two parameters at each party: a counter for remaining attempts and a timestamp for the last failed attempt. Upon a failed authentication, the counter decreases. If it hits zero and the time since the last attempt is less than a set lockout period, further attempts are halted. Successful authentication resets this counter. Even if $n - 1$ instances are malicious, a single honest instance preserves the rate-limit's integrity by locking out malicious users eventually. While this framework provides a solid rate-limiting foundation, it is also flexible, allowing integration of sophisticated mechanisms tailored to application needs, including serverless product offerings [17, 70, 71].

3.2.2 Access Keys for Cloud-level Authentication.

An attacker might continually invoke other users' serverless instances, depleting application provider funds. Upon invocation, the serverless instances run MPCAuth, preventing the attacker from authenticating and executing a sensitive operation. However, executing MPCAuth incurred a charge. Further, if rate-limiting is in place, the attacker can even lock the legitimate user out of their account.

To address this, Flock ensures that only pre-approved users can activate their serverless containers. In each cloud, we leverage specialized cloud access keys for local device storage, configured with fine-grained IAM permissions [9, 25, 32]. Without these, an attacker cannot invoke a user's instances. The owner and authorized users of the instances (see data-sharing applications in §4) store access keys locally. Even if an attacker compromises a user's device, the keys do not grant access to the user's secrets, but only to the invocation of the serverless instances. In the unlikely scenario that a user loses their device with the serverless URLs and cloud access keys, the user can manually authenticate with the cloud providers to retrieve them for a new client installation.

3.3 Resource Protection & Billing

Billing is challenging because the provider must pay for user storage and compute *without* being able to access them. At the same time, even though the provider relinquishes its deployment control to the users, malicious users should not be able to deplete the provider's funds. As we outline in §3.2.2, *invocation access keys* prevent an attacker from invoking other users' parties and draining the provider's funds. Now, we must ensure that a user cannot abuse provider resources through its own deployment, especially since attackers can also create user accounts. Hence, in Flock, application providers set a maximum spending limit per user.

To enforce this spending limit, we first discuss what cloud providers offer in this direction, and then describe a solution based on virtual cards. Prominent cloud tools like AWS Organizations [19, 215] and GCP Projects [53] allow a billing account to pay for other accounts' resources without having access to them. AWS Budgets [14] and GCP Budgets & Alerts [48] grant providers policies which trigger alerts and halt spending [30, 45] if a user's spending surpasses a limit. While these services provide what we need, they are not foolproof because cloud providers have not previously operated in the model of strictly preventing a billing account from accessing the accounts it funds. Hence, it is likely that the billing account can gain access in a case-by-case basis to the paid-for account, e.g. by calling a cloud admin for support. However, the fact that these mechanisms already exist suggest that it would be a small change for these clouds to turn this into a strict enforcement, which we advocate for.

Meanwhile, Flock can use virtual cards, which render Flock fully functional today with existing tools. *Virtual card services* such as Karta offer credit cards with set monthly limits for Azure, AWS, and GCP [159], and AWS also accepts pre-paid cards [93]. These providers can issue capped digital cards to users, replenishing funds as needed. Payment platforms like Stripe offer APIs to issue standard cards with spending limits [84], which only incur a few cents per user.

`create_acc(user_info, module, auth_policy)`
`(cloud_auth)`
Create user account: programmatic registration & deployment.

`deploy(user_info, module, auth_policy)` Called in `create_acc` & for application & Flock updates.
`setup_module(module, auth_policy, new_state)` Authenticates the client & sets sensitive data for the parties.
`execute(module_inputs, auth_inputs)` Authenticates the client & executes a sensitive module.

Table 1: Flock API

3.4 Programmatic Registration & Deployment

Table 1 presents the Flock API used by application developers. A key feature of Flock is that we automate the user experience for clients, so that it is similar to a regular application. The main difference from the standard experience is that the user's application client may surface $n-1$ interactive authentication steps, one for each cloud. However, the Flock API ensures that the user does not bear the burden of manually registering for $n-1$ cloud accounts or interfacing directly with the clouds to deploy the serverless instances.

3.4.1 Registration

When registering for an application, the user's client needs to create $n-1$ cloud accounts. To ensure that the *user* does not manually perform this work, Flock utilizes Playwright [65], a webpage automation framework [65, 76]. Users input their details (name, password, email) in the UI of the application only once at account setup, as they typically do. Flock then automatically populates these details across the $n-1$ cloud registration forms, deriving unique passcodes from the user's provided passcode. Flock inputs application-provided data without user intervention.

Cloud registration requires multi-step user interactions like SMS, email, or CAPTCHAs. To handle these, the `create_acc` function (Table 1) takes the initial `user_info` input that can be automatically populated. Next, the function returns the interactive `cloud_auth` object, which surfaces the steps that necessitate user intervention during registration. For example, AWS registration [109] (1) sends an email with a code to input, (2) asks for contact information, (3) requests billing data, and (4) sends an SMS with a code to input. Thus, Flock anticipates `user_info` to initially collect an email, contact information, billing data, and phone number. This data is held in-memory and fed programmatically into each cloud form as registration proceeds, while `cloud_auth` surfaces mid-registration interactive user input requests (e.g.

SMS codes or CAPTCHAs) to the application UI, transferring the resulting input to the cloud frontends.

3.4.2 Deployment

Once all cloud accounts are instantiated, `create_acc` calls `deploy` (Table 1), which deploys the serverless containers corresponding to the `module`. This deletes any previous deployments, and performs the setup for the cryptographic module and `auth_policy` authentication factors by calling `setup_module` (Table 2). Upon deployment, the client device locally stores cloud access keys (§3.2.2).

4 Applications & Cryptographic Modules

Flock enables a diversity of applications where every distributed-trust computation only takes input data from a single owner or from public sources. Some applications naturally meet this criteria because sensitive data often equates to user-owned secret values. Other applications might initially appear as if they do not fit the on-demand distributed-trust model (e.g. data-sharing, private information retrieval). By *reframing* these applications to the Flock setting, we demonstrate how they, too, can benefit from on-demand deployment. We show how Flock can enable 8 types of distributed-trust applications, based on 5 fundamental cryptographic modules. Across these applications, Flock enables data confidentiality and integrity, private queries on public data, and data-sharing. Table 2 summarizes how each module is encapsulated by the Flock API functions `setup_module` and `execute`.

4.1 Secret Recovery

Secret recovery applications allows a user to back up her secret key k in the form of n secret-shares $\{k_1,\dots,k_n\}$ [202], each one stored at each party. Even if an attacker compromises $n-1$ parties, it cannot reconstruct the key k without the n-th share. The user retrieves all shares to reconstruct k. For integrity, the client initially stores a salted hash of k at each party and confirms the hash of the reconstructed key matches.

Secure key recovery for end-to-end encryption (E2EE) applications [56, 69, 79, 86, 95, 98] has been a long-standing issue. If a user loses their private key k (e.g. by losing their device), they lose access to their data. However, backing up the key on the application server breaks the guarantees of E2EE by introducing a central point of attack. To avoid this, application providers like Signal secret-share user keys [179]. The user can authenticate to the n servers using a PIN [200] to retrieve the shares of k. One honest party prevents brute force attacks through PIN rate-limiting and allows the client to detect if the recovered key is incorrect.

Password managers [6, 7, 26, 59, 60, 83, 205] store encrypted or hashed versions of user passwords in the cloud. When a single cloud or cloud account is compromised as in the recent high-profile LastPass hack [133], an attacker can brute-force passwords [190]. Flock secret-shares passwords across n parties, and allows the user to reconstruct the passwords locally upon use. If up to $n-1$ parties are compromised, the attacker cannot brute-force the passwords.

4.2 Signing

Signing applications secret-share a signing key k among n parties. Later, the client authenticates to the parties, who run MPC to sign the client's message m with the secret key k. The secret key is never materialized, and the MPC produces the signature. Flock uses a maliciously-secure multi-party signature generation protocol [143].[3]

Digital asset custody is offered by MPC wallets [35, 38, 42, 46, 47, 66, 72, 77, 97, 134, 177], who secure billions [101, 196]. While these wallets typically secret-share between the the client and the application server, Flock enables them to achieve their roadmapped objective of increasing the number of trust domains to $n > 2$ [178]. To send assets to Bob, Alice's client formulates a message tx and invokes the parties, who reconstruct her key within MPC to sign tx.

Certificate authorities [39, 43, 49, 61] routinely sign certificates that bind digital identities to cryptographic keys. Breached signing keys have led to fraudulent certificates that green-light malware and impersonate trusted websites [216]. Flock enables certificate issuance without ever materializing the signing key on one server, providing a more cost-effective alternative to hardware security modules [15, 23].

Code signing services [2, 33, 39, 49, 57, 68, 87, 92] allow organizations who provide critical software to sign code updates. With Flock, the attacker cannot endorse malicious software unless all n parties are breached.

4.3 Decryption

Using Flock, Alice can secret-share an encryption key k_{Alice} among her n parties. An authorized user, say Bob, can provide a ciphertext c to the parties, who will reconstruct k_{Alice} within MPC and decrypt c for Bob. Unlike secret recovery, Bob cannot obtain k_{Alice}, but can use it to perform decryptions that satisfy a certain access policy, as exemplified below. Flock uses a maliciously-secure AES-in-MPC protocol [41, 129], guaranteeing the decryption's integrity.

4.4 Data-sharing in Hierarchical File Systems

End-to-end encryption systems [21, 28, 36, 56, 67, 81, 85, 88] typically operate hierarchically: a user or group key encrypts a directory key, which then encrypts a group of file keys, each of which encrypts a file. If Alice, for instance, is unavailable, and Bob urgently needs a file F encrypted with k_F under

[3] While an attack for the GG18 [143] and GG20 [144] protocols was recently discovered [181], the patch was integrated into the library we use [89].

	`setup_module`	`execute`
Secret Recovery	Shard k as $k_1 \dots k_n$, $\texttt{cloud}_i$ stores k_i.	$\texttt{cloud}_i$ sends k_i to client, who reconstructs k from $k_1 \dots k_n$.
Signing	Clouds gen. $k_1 \dots k_n$ in MPC, each store k_i.	Client sends m. Clouds retrieve k_i, sign m in MPC.
Decryption	Shard AES key as $k_1 \dots k_n$, $\texttt{cloud}_i$ stores k_i.	Clouds retrieve k_i, AES decrypt ciphertext in MPC.
PIR	No client setup.	Send clouds DPF requests, reconstruct $d[i]$ via responses.
Freshness	Store file k in $\texttt{cloud}_1$, $h = H(k)$ in $\texttt{cloud}_2$.	Client retrieves k and h, and checks $h = H(k)$.

Table 2: Setup & execution specification for Flock modules.

Alice's key k_{Alice}, he should be able to access the file based on an access policy (e.g. a period of inactivity, signatures from users with authority), but without learning the key k_{Alice}, which would grant him excessive access. During setup, Alice shares her invocation access keys with Bob and configures her parties with the access policy and the authentication factors to verify from Bob (e.g. Bob's U2F) (§3.2.2). Bob can then authenticate, supply the encrypted k_F to Alice's n parties, which, after policy verification, allow Bob to decrypt k_F.

4.5 Private Information Retrieval

Private information retrieval (PIR) [113, 116, 124, 125, 146, 152, 169] enables users to query a public database at index i, without the servers learning i, and has many use cases [27, 105, 106, 110, 148, 149, 157, 158, 166, 174, 191, 197, 201, 211, 212].

The integration of two-party PIR in Flock showcases a different type of sensitive data access compared to aforementioned modules. Instead of storing a user-specific secret at the parties, we have a public dataset accessed by all the users and the sensitive data of each user is their *query*. In a traditional deployment, each PIR server stores the database. In Flock, we observe that the user's parties can serve as PIR parties since the data is public. However, the cost of storing the entire database in *each* user's cloud would compound. Instead, the database owner can place a public database copy in each of the n cloud providers, accessible by any Flock user deployment. For instance, if Alice queries index i from her Flock deployment in AWS, she would only need to access the AWS database copy, eliminating cross-cloud latency and egress. We port an existing PIR implementation [10] to Flock.

To introduce malicious security, a trusted database owner can store each entry with a signature of the entry, which the client can verify upon reconstruction. If one party tampers with the signature and the other is honest, all queries will fail. In contrast, adding malicious security to single-server PIR exhibits significant costs [125]. Public databases are community-scrutinized to prevent the database owner from tampering with the database. However, a known approach of encoding MAC key into DPF keys can remove this trust assumption from the database owner [125, 132].

4.6 Data Freshness

Data freshness applications often power rollback protection and file integrity, which are long-standing obstacles in systems where the application provider has control over stored user data [104, 155, 160, 167, 183]. Flock utilizes a hash-based freshness module based on Verena [160]. This application demonstrates that Flock's sensitive data does not need to be *secret* data; rather, the sensitive data is the *integrity* of the file system. In Flock, the application provider acts as the file storage server, while a hash server is deployed by the user via Flock. Users safeguard against tampered or outdated file versions by storing a hash of their latest file in the hash server, allowing users to guarantee *their own* file integrity. When a file is stored, its latest hash is saved and signed by the hash server for client verification. During retrieval, the hash server sends the client a signed hash, confirming the file's latest version. Flock's freshness module also incorporates access control. The deploying user retains ownership, granting `read` and `write` permissions so other users can view or update the latest file hash. Signatures from the hash server guarantee the integrity of the file.

5 Evaluation

In this section, we answer: *How do the performance and cost of Flock compare to traditional distributed trust?*

5.1 Implementation

We implemented Flock using ∼2,000 lines of Go (signature protocol, relay), ∼2,000 lines of Python (freshness protocol, client, deployment, storage, server-side "frontend"), and ∼2,000 lines of C++ (passcode, decryption, PIR, relay client). We used cloud SDKs and black-boxed foundational cryptographic libraries: `tss-lib` [89] for the Multi-Party Threshold Signature Scheme [143], `emp-agmpc` [41] for Global-Scale Secure Multi-Party Computation [213], and Google's implementation [10] of incremental distributed point functions [113]. For the relay, we used OpenSSL [64] and its Go bindings[4] for the SSL_shutdown [82] procedure (§3.1).

[4] We fork and adapt github.com/spacemonkeygo/openssl.

Module	System	Breakdown (ms)		End-to-End (ms)	
		Client	*Server*	Mean (μ)	SD (σ)
Secret Recovery:	*Baseline*	80.50	252.74	356.05	20.03
`setup_module`	*Flock*	84.51	260.21	409.09	35.73
Secret Recovery:	*Baseline*	77.45	201.91	302.33	18.13
`execute`	*Flock*	77.52	208.82	340.48	19.75
Signing:	*Baseline*	2.49	4,537.43	4,574.12	19.55
`setup_module`	*Flock*	2.75	4,718.79	4,776.36	21.84
Signing:	*Baseline*	2.60	1,031.31	1,053.44	8.55
`execute`	*Flock*	2.81	1,322.75	1,360.35	23.72
Decryption:	*Baseline*	1.33	200.11	213.82	7.74
`setup_module`	*Flock*	3.05	171.67	231.65	7.76
Decryption:	*Baseline*	3.36	21,767.18	21,786.63	489.57
`execute`	*Flock*	3.33	21,926.81	21,974.29	626.51
PIR:	*Baseline*	12.25	202.54	227.20	6.19
`execute`	*In-memory*	12.25	10.62	35.28	3.29
	Flock	12.24	131.02	170.22	8.85
Freshness:	*Baseline*	14.87	218.55	252.47	23.65
`setup_module`	*Flock*	5.05	209.79	244.63	10.80
Freshness:	*Baseline*	4.77	189.27	205.90	14.04
`execute`	*Flock*	4.99	188.48	222.76	8.10
Authentication Factor					
U2F	*Baseline*	7.12	527.28	595.88	32.85
	Flock	8.74	515.16	646.85	18.77
PIN	*Baseline*	13.40	909.48	988.60	28.44
	Flock	12.96	1,268.09	1,341.47	35.45

Table 3: Latency of modules & authentication factors. We fix 2^{10} as the input size for brevity.

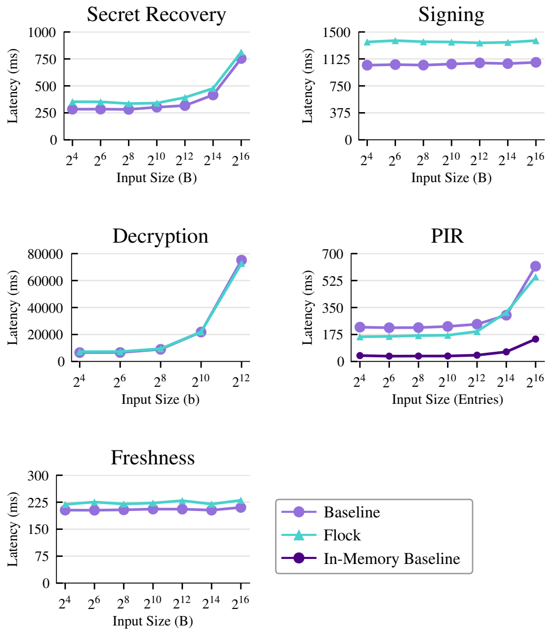


Figure 5: End-to-end latency of cryptographic modules across varying input sizes: number of database entries (Entries) for PIR, bits (b) for decryption, bytes (B) for all other modules.

Our implementation is open-sourced at github.com/flock-org/flock.

5.2 Experiment Setup

The **baseline** is a traditional distributed-trust setup consisting of three VMs in the three major clouds. We use 2 vCPU, 8 GB memory servers in California for AWS (m5.large, \$80.64/month), Azure (Standard_D2as_v4, \$80.64/month), and GCP (n2-standard-2, \$85.16/month). Selecting servers in close proximity minimizes network delays, in-line with typical MPC deployments. The client is an AWS m5.large. This setup is comparable to that of traditional distributed-trust systems in prior literature (§6).

In **Flock**, application providers typically run one party and users deploy $n-1$ parties (§1.1). It is more cost-efficient for an application provider to run their party on a single VM because they can amortize costs among many users without halting the instance [154]. The application server is in Azure and has the same configuration as the baseline's application server in Azure. The client and the regions for the parties are also like in the baseline. We use an Azure Standard_B2s VM (2 vCPU, 4 GB, \$36/month) for the Flock Relay. For the serverless containers of signing, decryption, and PIR, we used 2 vCPU AWS Lambda [18] (3,538 MB memory, \$0.0000575/s) and Google Cloud Run [52] (512 MB, \$0.00006895/s) instances. For secret recovery and freshness which primarily conduct cloud storage operations rather than server-side compute, we use smaller serverless instances: 0.5 vCPU AWS Lambda (895 MB memory, \$0.0000144/s) and 0.75 vCPU Google Cloud Run (512 MB, \$0.00002695/s). We selected the smallest available memory size for serverless instances.

5.3 Latency

We evaluate the latency of the baseline and Flock for each cryptographic module (described in §4), scaling with their respective input sizes: the secret size for secret recovery, message size for signing, plaintext size for decryption, number of database entries for PIR, and file size for freshness checks. We chose sizes reflective of typical workloads (e.g. size of a cryptocurrency transaction, certificate, or encryption key). For PIR, input size is the number of *database entries* of size 128 B. For decryption, input size is the number of *bits* since our suggested application only requires efficient file key encryption. For all others, input size is the *byte count*. Latency benchmarks were averaged over 10 runs. For the two-party modules (PIR and freshness), we use the application provider server and the Lambda.

Fig. 5 depicts the latency-input size relationship for both baseline and Flock. Table 3 breaks down latency results for an input size of 2^{10} into client, server, and end-to-end times, with the latter also accounting for client-server network time. We also break down the latency of MPCAuth authentication factors PIN (standard 4-digit format) and U2F. The PIN implementation also supports long passcodes. The function-independent phase in the decryption and PIN circuits can be executed offline to further reduce latency. Utilizing Flock for PIR necessitates streaming and deserializing the database from cloud storage for each request; in a traditional two-server PIR, databases can be held in-memory so we also evaluate a version of our baseline with an in-memory database.

As evidenced in Table 3 and Fig. 5, Flock does not significantly impact the latency of the 5 major cryptographic modules and their MPCAuth factors. More modules exhibit slightly higher standard deviation in Flock, which is expected due to the burstiness of serverless computing. The relay is used in decryption, signing, and PIN: While decryption is 1.01x the latency of the baseline since it is more compute-heavy, signing is 1.28x since it is communication-heavy and the relay slightly impacts performance, as we will concretize in §5.4. As anticipated, Flock exhibits considerably higher latency that the PIR in-memory baseline variant. In exchange, Flock PIR deployments reap the benefits of automatic trust domains and practical malicious security (§4.5). For consistency between Flock and the baseline, as well as between the modules, we use the S3 storage PIR baseline for the remainder of experiments and calculations. Flock-enabled PIR can be enhanced by employing latency-optimized cloud storage services at an extra cost [8] or by parallelizing computation while streaming the remainder of the database.

Averaged over all modules, Flock has a 1.05x latency overhead compared to the S3 baselines. As expected, most cryptographic modules exhibit higher latency with greater input size. Signing remains constant since the protocol [143] signs a *hash* of the message. Freshness is also constant since it is bottlenecked by reads and writes to cloud storage, which are fast at this file size.

Serverless coldstart & deployment latency are factors in Flock, unlike traditional distributed trust. We opted for serverless containers with low coldstart [18, 52, 58] over those with high coldstart [3, 16, 22] and designed lean Docker containers (634 MB pre-compression, 225 MB post-compression). Containers are stored in the application provider's Elastic Container Registry on each cloud, so that users need not build containers. Deployment latency averaged 16.13s for AWS Lambda and 15.88s for Google Cloud Run across 10 tests. For coldstart measurements, we used AWS CloudWatch's X-Ray and invoked Google Cloud Run after idle periods, resulting in 1.02s (AWS Lambda) and 2.10s (Google Cloud Run), averaged over 10 runs. Providers can minimize (or eliminate) coldstart times by keeping containers warm through periodic polling [1]. Applications can also deploy smaller containers with only necessary module dependencies, not all 5.

	Per-Conn.	Setup Latency (ms)		Concurrent Users	
	Gb/s	*S2R*	*Total*	*Sign*	*Decrypt*
Baseline	1.94	–	24.48	–	–
Flock	1.72	20.66	49.72	11,700	1,900
Wireguard	0.063	25.5	78.43	–	–

Table 4: Single connection throughput & establishment latency, and number of concurrent users supported by the relay. S2R includes steps 1-7 of the per-invocation protocol (§3.1.3).

5.4 Relay Evaluation

Per-connection latency & throughput. Table 4 compares the setup latency of the Per-Invocation protocol (§3.1.3) and the throughput[5] of a Flock Relay TLS connection to the baseline's direct TLS connection. We use our Azure and GCP VMs as endpoints, with the relay hosted in the AWS VM. Averaged over 50 runs, a Flock connection's throughput is 0.89x that of a direct TLS connection since all traffic is forwarded through the relay. The setup latency is 2x that of a direct connection due to the additional S2R handshake.

We also benchmark the method used by Tailscale DERP [37], which connects Wireguard [96] endpoints with a relay that re-encrypts Wireguard packets into TLS messages. The Wireguard setup is significantly less performant than the Flock Relay at 0.03x the per-connection throughput of the baseline. As we explain in §3.1, the Wireguard setup incurs significant overhead from encrypting packets at the more granular IP layer, decrypting and re-encrypting all traffic using TLS at the relay, and redundantly TLS-encrypting the Wireguard packets at the endpoints. Setting up Wireguard interfaces and iptable routes introduces 3.2x the setup latency of the baseline. Therefore, Flock Relay connections outperform prior work and nearly match the efficiency of direct TLS.

Cross-user throughput. To evaluate the maximum capacity of the relay, we measure its concurrent user throughput, independent of external factors like application server performance and the compute of cryptographic modules. By emulating traffic patterns for the signing and decryption modules from multiple threads, we saturated the CPU utilization of the relay VM. Results in Table 4 show it can handle 11,700 concurrent signing or 1,900 decryption requests, using 1.9 GB memory. The application provider can further scale the relay deployment based on user demand. We remark that these values represent a worst-case scenario where *all* users invoke Flock, generating traffic patterns in a burst without compute-induced delay. Typically, the Flock Relay can support additional users when they spend intermittent time on compute tasks.

[5] We use github.com/udhos/goben/ for throughput measurements.

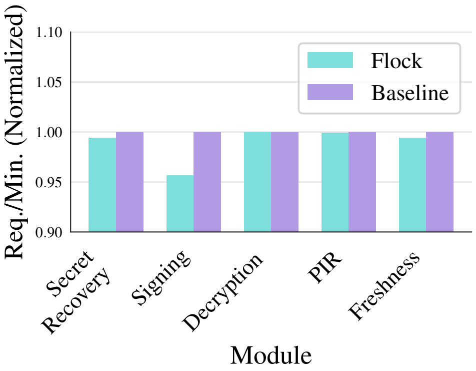


Figure 6: Maximum requests/min. of Flock (F), normalized over the baseline (B). (Secret Recovery: F-1376, B-1384; Signing: F-66, B-69; Decryption: F-5, B-5; PIR: F-1195, B-1196; Freshness: F-1162, B-1169)

5.5 Throughput & Cost

System throughput. Flock achieves throughput comparable to the baseline, as illustrated in Fig. 6. We demonstrate this by fixing an input size of 2^{10} and invoking as many Flock requests as possible.We verify that the CPU utilization of the VM(s) in both Flock and the baseline is 100% at the maximum load, which is the threshold at which additional threads cannot further increase the number of successfully completed requests per minute.

Cost. We use this experiment to calculate the cloud cost estimates for the baseline and Flock in the worst case (Table 5) and across varying server utilization rates (Fig. 7). For each module, we use the baseline's request-per-minute from Fig. 6 to calculate the cost-per-request by dividing the monthly server cost by the number of monthly requests. We then measure Flock's cost-per-request using the per-second vCPU and memory costs, and the same method as the baseline to calculate the application provider's server cost. Finally, we aggregate all cloud expenses to get the total computational cost per operation. We measure the bytes of network traffic transferred by each protocol. AWS, GCP, and Azure charge a network egress fee of \$0.09/GB, \$0.085/GB, and \$0.087/GB, respectively, which we use to calculate the cross-cloud and cloud-to-client data transfer fees. Each module also includes a persistently stored state. For one month, AWS, GCP, and Azure charge \$0.026/GB, \$0.023/GB, and \$0.021/GB, respectively. We include the resulting bandwidth, storage, network, and compute cost per invocation in Table 5.

Averaging across modules, Flock is 2.27x the worst-case cost of the baseline. Table 5 assumes server load is saturated, yielding the lowest possible cost-per-user. However, application providers rarely operate at full server capacity and often provision excessive resources to handle spikes in usage. Fig. 7 shows Flock and the baseline's module average of the per-invocation cost, varying the server utilization from 5-100% of the maximum requests-per-minute. For operating at 50% utilization, the cost of Flock is only 23% more. Flock is actually less expensive on average than the baseline when the monthly requests completed are up to 20% of the baseline's maximum capacity, because of the serverless compute model.

Finally, we emphasize that the baseline has *additional costs* beyond the cloud cost, which do not exist in Flock (§1.1). First, the application provider must compensate its business partners (who have employees or seek profit). Second, the hidden price of the traditional setup is the manual, time-consuming, and difficult challenges of finding suitable business relationships to setup distributed trust.

6 Related Work

Traditional distributed-trust deployments have exhibited a host of obstacles [178] for industry-leading teams, including Signal [79], ISRG [54], and Coinbase [177]. Prio has been employed for private analytics in COVID-19 exposure notifications and Firefox telemetry [100, 108, 126, 136], but ISRG encountered difficulties with cross-organizational inconsistencies in testing and debugging [99, 145, 178]. Signal struggled to deploy traditional distributed trust for its secret recovery application [179], citing reliance on third-parties for security, constant up-time, and user trust [79, 178]. Meta's Private Lift leverages MPC for private advertising, yet advertiser onboarding is time-consuming [187, 198]. Astran [12] has attempted secret-sharing user data across clouds, but their servers see the plaintext data and are therefore a central point of attack [13]. Thus, while the cryptographic guarantees of distributed trust have been instrumental in securing several impactful applications, *deployment* has been a central challenge.

MPC [112, 128, 147, 162–164, 213, 219] **and PIR** [40, 105, 113, 116, 125, 146, 152, 169] **applications** are growing in relevance. MPC applications include private analytics [114, 126] and MPC wallets [35,38,42,46,47,66,72,77,97,101,134,177]. PIR applications include private contact discovery [158], credential reporting [174, 191, 207, 212], blocklist lookups [166], and media delivery [149]. Both primitives have been used for private search [131, 132, 151, 194, 197, 211], private advertising [110, 148, 157, 187, 201], and anonymous messaging [105, 106, 123, 127, 138, 170, 171, 217]. **Data freshness** is important for preventing rollback attacks, e.g. in trusted execution environments [104, 167, 183]. Prior work introduces hash servers for file integrity [155, 160]. Flock's contribution is orthogonal and focuses on *deploying* such systems. Many of the systems that Flock supports can be mapped to our baseline setup in §5, and use an underlying cryptographic module that we benchmark. Another line of work [172] aids in the deployment of non-cryptographic distributed trust by offering different privileges to each trust domain; our work instead focuses on offering a deployment mechanism for distributed trust based on strong *cryptographic* guarantees.

Module	*Bandwidth (KB)*	*Storage*	*Network*	*Compute*	***Total***	*Compute*	***Total***
Secret Recovery	25.83	0.0000	0.0002	0.0004	0.0006	0.0016	0.0018
Signing	67.38	0.0000	0.0006	0.0083	0.0089	0.0213	0.0219
Decryption	59,763	0.0000	0.5219	0.1141	0.6360	0.3340	0.8559
PIR	1.38	0.0009	0.0000	0.0005	0.0014	0.0024	0.0033
Freshness	2.89	0.0000	0.0000	0.0005	0.0005	0.0011	0.0011

Table 5: Worst-case cost per one user invocation (USD *cents*). We show the maximum number of requests per minute that the baseline can handle, bandwidth (KB), storage cost, networking cost, and the compute cost for each a single invocation in the baseline and Flock.

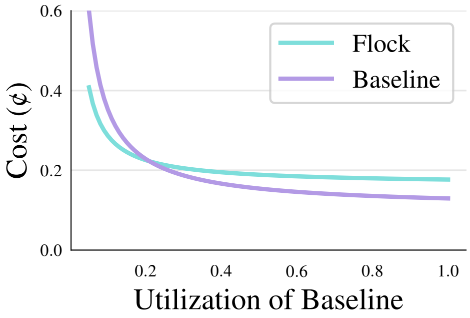


Figure 7: Average per-invocation cost (¢) across all modules for Flock and the baseline, between 5-100% utilization of the maximum baseline capacity.

Serverless networking is a longstanding limitation [107, 150] of serverless computing. While service meshes [31, 55] and proxies [44] can connect services by abstracting network connections, they do not handle *private* endpoints. As we discuss in §3.1, a line of work employs NAT traversal and hole-punching for serverless communication [137, 139, 186, 214], but requires costly per-user services like private clouds or NAT gateways. Recent systems use a relay to enable serverless networking [140, 141, 210], but do not consider security. Some works conduct TLS over multi-step network connections [80, 208], but cannot handle publicly inaccessible endpoints. Wireguard [96] and Tailscale DERP relays [37] securely connect private endpoints, but are unsuitable for serverless as we explained in §3.1. We build upon the the relay-based technique in the literature to architect the first *end-to-end encrypted* relay which has negligible detriment to performance.

Hardware enclaves have been proposed [130, 179] as a replacement to deploying $n-1$ trust domains to safeguard secrets and execution from the application provider. However, enclaves are vulnerable to side-channel attacks that compromise remote attestation, including leaks through SGAxe [209], Plundervolt [188], AEPic Leak [115], and CIPHERLEAKS [175]. With root access to deployment servers, application providers can exploit such side-channels to access secrets. Hence, while enclaves are often utilized as a *supplementary* defense alongside cryptography, applications often opt for cryptography as the primary security measure [178]. In contrast, Flock sets up distributed trust on n major clouds without relying on trusted hardware.

User-centric deployment has been validated in traditional systems work [90, 103, 120–122, 168, 173, 185, 192, 199] in which users deploy components of the applications to retain privacy from a provider. Users sandbox and isolate components of their application to enforce user control. Unlike Flock, these methods do not utilize distributed trust, and thus position a cloud, device, or server as a central point of attack. Flock draws from the underlying principles of user-centric deployment by applying this framework to distributed trust.

7 Conclusion

This work introduces the on-demand distributed-trust architecture, which enables application providers to automatically deploy distributed-trust applications, thus surpassing the cumbersome, manual, and time-consuming process of setting up business relationships. To reverse the deployment from provider to users, our platform Flock consists of a cost-effective cross-cloud serverless framework supporting a variety of distributed-trust applications. We hope that Flock catalyzes an increase in the deployment of distributed trust.

Acknowledgements. We thank the anonymous reviewers, our shepherd Luis Rodrigues, as well as Sam Kumar, Emma Dauterman, Sebastian Angel, Henry Corrian-Gibbs, and the students in the Sky security group for their feedback. This work is supported by NSF CAREER 1943347 and gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, Uber, and VMware.

References

[1] 3 solutions to mitigate the cold-starts on Cloud Run. https://medium.com/google-cloud/3-solutions-to-mitigate-the-cold-starts-on-cloud-run-8c60f0ae7894, Nov 2020.

[2] Authenticode Digital Signatures. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/authenticode, Dec 2021.

[3] AWS Fargate vs. Lambda: Performance. https://www.simform.com/blog/aws-fargate-vs-lambda/, June 2022.

[4] Connect multicloud microservices using Oracle API Gateway. https://docs.oracle.com/en/solutions/oci-multicloud-api-gateway/, Dec 2022.

[5] Should businesses consider WireGuard? https://www.twingate.com/blog/what-is-wireguard-vpn, Aug 2022.

[6] 1Password. https://1password.com/, April 2023.

[7] About the 1Password security model. https://support.1password.com/1password-security, April 2023.

[8] Amazon Elastic Block Store (Amazon EBS). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS, Oct 2023.

[9] AmazonECSTaskExecutionRolePolicy. https://docs.aws.amazon.com/aws-managed-policy/latest/reference/AmazonECSTaskExecutionRolePolicy.html, October 2023.

[10] An Implementation of Incremental Distributed Point Functions in C++. https://github.com/google/distributed_point_functions, September 2023.

[11] Anthos Multi-Cloud API. https://cloud.google.com/anthos/clusters/docs/multi-cloud/reference/rest, Oct 2023.

[12] Astran. https://astran.io/, Nov 2023.

[13] Astran API Tutorial. https://docs.astran.io/docs/api/getting-started/, Nov 2023.

[14] AWS Budgets. https://aws.amazon.com/aws-cost-management/aws-budgets/, October 2023.

[15] AWS CloudHSM Pricing . https://aws.amazon.com/cloudhsm/pricing/, Oct 2023.

[16] AWS Fargate. https://aws.amazon.com/fargate/, September 2023.

[17] AWS Fargate throttling quotas. https://docs.aws.amazon.com/AmazonECS/latest/userguide/throttling.html, Oct 2023.

[18] AWS Lambda. https://aws.amazon.com/lambda/, April 2023.

[19] AWS Organizations. https://aws.amazon.com/organizations/, October 2023.

[20] AWS Service Level Agreements (SLAs). https://aws.amazon.com/legal/service-level-agreements/, April 2023.

[21] AxCrypt. https://axcrypt.net/, April 2023.

[22] Azure Container Instances. https://azure.microsoft.com/en-us/products/container-instances, September 2023.

[23] Azure Dedicated HSM pricing. https://azure.microsoft.com/en-gb/pricing/details/azure-dedicated-hsm, Oct 2023.

[24] Azure Functions. https://azure.microsoft.com/en-us/products/functions/, April 2023.

[25] Azure resource provider operations. https://learn.microsoft.com/en-us/azure/role-based-access-control/resource-provider-operations#microsoftcontainerinstance, October 2023.

[26] Bitwarden. https://bitwarden.com/, April 2023.

[27] Blyss. https://blyss.dev/, October 2023.

[28] Boxcryptor. https://www.boxcryptor.com/, April 2023.

[29] Building a secure webhook forwarder using an AWS Lambda extension and Tailscale. https://aws.amazon.com/blogs/compute/building-a-secure-webhook-forwarder-using-an-aws-lambda-extension-and-tailscale/, Sep 2023.

[30] Capping API usage. https://cloud.google.com/apis/docs/capping-api-usage, October 2023.

[31] Cilium. https://cilium.io, Nov 2023.

[32] Cloud Run IAM roles. https://cloud.google.com/run/docs/reference/iam/roles, October 2023.

[33] Comodo. https://www.comodo.com/business-security/code-signing-certificates/code-signing.php, April 2023.

[34] Configure secrets. https://cloud.google.com/run/docs/configuring/services/secrets, Nov 2023.

[35] Cryptography and MPC in Coinbase Wallet as a Service (WaaS). https://coinbase.bynder.com/m/687ea39fd77aa80e/original/CB-MPC-Whitepaper.pdf, June 2023.

[36] Cryptomator. https://cryptomator.org/, April 2023.

[37] DERP Servers. https://tailscale.com/kb/1232/derp-servers/, Oct 2023.

[38] Dfns. https://www.dfns.co/, September 2023.

[39] DigiCert. https://www.digicert.com/, April 2023.

[40] DoTS: Berkeley Distributed Trust Stack. https://sky.cs.berkeley.edu/dots/, May 2023.

[41] emp-agmpc. https://github.com/emp-toolkit/emp-agmpc, April 2023.

[42] Entropy. https://entropy.xyz/, April 2023.

[43] Entrust. https://www.entrust.com/, April 2023.

[44] Envoy. https://www.envoyproxy.io, Nov 2023.

[45] Examples of automated cost control responses. https://cloud.google.com/billing/docs/how-to/notify#cap_disable_billing_to_stop_usage, October 2023.

[46] Fireblocks. https://www.fireblocks.com/, April 2023.

[47] Fordefi. https://www.fordefi.com/, September 2023.

[48] GCP Budgets. https://cloud.google.com/billing/docs/how-to/budgets, October 2023.

[49] GlobalSign. https://www.globalsign.com/, April 2023.

[50] Google Cloud Functions. https://cloud.google.com/functions, April 2023.

[51] Google Cloud Platform Service Level Agreements. https://cloud.google.com/terms/sla, April 2023.

[52] Google Cloud Run. https://cloud.google.com/run, September 2023.

[53] IAM roles for billing-related job functions. https://cloud.google.com/iam/docs/job-functions/billing, April 2023.

[54] Internet Security Research Group. https://www.abetterinternet.org/, September 2023.

[55] Istio. https://istio.io, Nov 2023.

[56] Keybase. https://keybase.io/, April 2023.

[57] Keyfactor. https://www.keyfactor.com/platform/enterprise-code-signing/, April 2023.

[58] Lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments, Oct 2023.

[59] LastPass. https://www.lastpass.com/, April 2023.

[60] LastPass Technical Whitepaper. https://support.lastpass.com/help/lastpass-technical-whitepaper, April 2023.

[61] Let's Encrypt. https://letsencrypt.org/, April 2023.

[62] Mixed-Species Flocking. https://web.stanford.edu/group/stanfordbirds/text/essays/Mixed-Species_Flocking, Oct 2023.

[63] Multi-cloud provisioning. https://www.terraform.io/use-cases/multi-cloud-deployment, Oct 2023.

[64] OpenSSL. https://www.openssl.org/, Oct 2023.

[65] Playwright. https://playwright.dev/, September 2023.

[66] Portal. https://www.portalhq.io/, September 2023.

[67] PreVeil. https://www.preveil.com/, April 2023.

[68] PrimeKey. https://www.primekey.com/solutions/code-signing/, April 2023.

[69] ProtonMail. https://proton.me/, April 2023.

[70] Rate limiting overview. https://cloud.google.com/armor/docs/rate-limiting-overview, Oct 2023.

[71] Resource availability & quota limits for ACI. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-resource-and-quota-limits, Oct 2023.

[72] Safeheron. https://www.safeheron.com/, September 2023.

[73] Securing Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/security-concepts, April 2023.

[74] Securing Cloud Functions. https://cloud.google.com/functions/docs/securing, April 2023.

[75] Security in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html, April 2023.

[76] Selenium. https://www.selenium.dev/, April 2023.

[77] Sepior. https://sepior.com/, April 2023.

[78] Service Level Agreements (SLA) for Online Services. https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services, April 2023.

[79] Signal. https://signal.org/, April 2023.

[80] Signal TLS Proxy. https://github.com/signalapp/Signal-TLS-Proxy, Oct 2023.

[81] SpiderOak. https://spideroak.com/, April 2023.

[82] SSL_shutdown. https://www.openssl.org/docs/manmaster/man3/SSL_shutdown.html, Nov 2023.

[83] Storage: Bitwarden Help Center. https://bitwarden.com/help/data-storage/, April 2023.

[84] Stripe Card Issuance. https://stripe.com/issuing, October 2023.

[85] Sync. https://sync.com/, April 2023.

[86] Telegram. https://telegram.org/, April 2023.

[87] Thawte. https://www.thawte.com/code-signing/, April 2023.

[88] Tresorit. https://tresorit.com/, April 2023.

[89] tss-lib. https://github.com/bnb-chain/tss-lib, April 2023.

[90] Use Advanced Data Protection for your iCloud data . https://support.apple.com/en-gb/guide/iphone/iph584ea27f5/ios, Oct 2023.

[91] Use AWS Secrets Manager secrets in AWS Lambda functions. https://docs.aws.amazon.com/secretsmanager/latest/userguide/retrieving-secrets_lambda.html, Nov 2023.

[92] Venafi. https://venafi.com/codesign-protect/, April 2023.

[93] Visa Prepaid Cards. https://usa.visa.com/pay-with-visa/cards/prepaid-cards.html, October 2023.

[94] VMware Cross-Cloud Services. https://www.vmware.com/cross-cloud-services, Oct 2023.

[95] WhatsApp. https://www.whatsapp.com/, April 2023.

[96] Wireguard. https://www.wireguard.com/, Nov 2023.

[97] ZenGo. https://zengo.com/, April 2023.

[98] Zoom. https://zoom.us/, April 2023.

[99] John Aas. Project Update and New Name for ISRG Prio Services: Introducing Divvi Up. https://divviup.org/blog/prio-services-update/, Dec 2021.

[100] Josh Aas and Tim Geoghegan. Introducing ISRG Prio Services for Privacy Respecting Metrics. https://www.abetterinternet.org/post/introducing-prio-services/, Nov 2020.

[101] Svetlana Abramova and Rainer Böhme. Anatomy of a High-Profile Data Breach: Dissecting the Aftermath of a Crypto-Wallet Case. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 715–732. USENIX Association, August 2023.

[102] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434. USENIX Association, February 2020.

[103] Muneeb Ali, Ryan Shea, Jude Nelson, and Michael J Freedman. Blockstack Technical Whitepaper. *Blockstack PBC, October*, 12, 2017.

[104] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 193–208, 2023.

[105] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.

[106] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569. USENIX Association, November 2016.

[107] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[108] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA) White Paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf, Apr 2021.

[109] AWS. Creating an AWS account. https://docs.aws.amazon.com/accounts/latest/reference/manage-acct-creating.html, Apr 2023.

[110] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *2012 IEEE Symposium on Security and Privacy*, pages 257–271. IEEE, 2012.

[111] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State Machine Replication in the Libra Blockchain. 2019.

[112] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, page 1–10. Association for Computing Machinery, 1988.

[113] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight Techniques for Private Heavy Hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776, 2021.

[114] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight Techniques for Private Heavy Hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776, 2021.

[115] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[116] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 1292–1303. Association for Computing Machinery, 2016.

[117] Chuck Brooks. Alarming Cyber Statistics For Mid-Year 2022 That You Need To Know, 2022.

[118] Jon Buck. Coincheck: Stolen $534 Mln NEM Were Stored On Low Security Hot Wallet. https://cointelegraph.com/news/coincheck-stolen-534-mln-nem-were-stored-on-low-security-hot-wallet, Jan 2018.

[119] Vitalik Buterin. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. 2013.

[120] Tej Chajed, Jon Gjengset, M Frans Kaashoek, James Mickens, Robert Morris, and Nickolai Zeldovich. Oort: User-Centric Cloud Storage with Global Queries. 2016.

[121] Tej Chajed, Jon Gjengset, Jelle Van Den Hooff, M Frans Kaashoek, James Mickens, Robert Morris, and Nickolai Zeldovich. Amber: Decoupling User Data from Web Applications. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[122] Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich. Separating Web Applications from User Data Storage with BSTORE. In *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[123] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptology*, 1:65–75, 1988.

[124] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50, 1995.

[125] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J Wu, and Bryan Ford. Authenticated private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3835–3851, 2023.

[126] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282. USENIX Association, March 2017.

[127] Henry Corrigan-Gibbs, Dan Boneh, and David Mazieres. Riposte: An Anonymous Messaging System Handling Millions of Users. *2015 IEEE Symposium on Security and Privacy*, page 321–338, 2015.

[128] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for Dishonest Majority. Cryptology ePrint Archive, Paper 2018/482, 2018. https://eprint.iacr.org/2018/482.

[129] Ivan Damgård and Marcel Keller. Secure Multiparty AES. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, page 367–374. Springer-Verlag, 2010.

[130] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on Trusting Distributed Trust. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, page 38–45. Association for Computing Machinery, 2022.

[131] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An Encrypted Search System with Distributed Trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119. USENIX Association, November 2020.

[132] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A Private Time-Series Database from Function Secret Sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2450–2468, 2022.

[133] Ben Demers. Struggling LastPass Suffers New Data Breach. Is Your Account at Risk? `https://www.kiplinger.com/personal-finance/lastpass-hack`, Jan 2023.

[134] Juergen Eckel. Hardware Security Modules vs. Secure Multi-Party Computation in Digital Asset Custody: The Drawback of Choosing Just One and What Happens When You Combine Them. `https://blog.riddleandcode.com/hardware-security-modules-vs-107729d6d3ea`, Oct 2022.

[135] Yehia Elkhatib. Mapping Cross-Cloud Systems: Challenges and Opportunities. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[136] Steven Englehardt. Next steps in privacy-preserving Telemetry with Prio. `https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/`, Jun 2019.

[137] Jeffrey Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. 01 2005.

[138] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792. USENIX Association, August 2021.

[139] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, April 2005.

[140] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488. USENIX Association, July 2019.

[141] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376. USENIX Association, March 2017.

[142] Jake Frankenfield. Private key: What it is, how it works, best ways to store. `https://www.investopedia.com/terms/p/private-key.asp`, Feb 2023.

[143] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. Cryptology ePrint Archive, Paper 2019/114, 2019. `https://eprint.iacr.org/2019/114`.

[144] Rosario Gennaro and Steven Goldfeder. One Round Threshold ECDSA with Identifiable Abort. Cryptology ePrint Archive, Paper 2020/540, 2020. `https://eprint.iacr.org/2020/540`.

[145] Tim Geoghegan. Exposure Notifications Private Analytics: Lessons Learned From Running Secure MPC at Scale. `https://divviup.org/blog/lessons-from-running-mpc-at-scale/`, May 2022.

[146] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658. Springer Berlin Heidelberg, 2014.

[147] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, page 218–229. Association for Computing Machinery, 1987.

[148] Matthew Green, Watson Ladd, and Ian Miers. A Protocol for Privately Reporting Ad Impressions at Scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1601, 2016.

[149] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107. USENIX Association, March 2016.

[150] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. *CoRR*, abs/1812.03651, 2018.

[151] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private Web Search with Tiptoe. *Cryptology ePrint Archive*, 2023.

[152] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905. USENIX Association, August 2023.

[153] Jiangshui Hong, Thomas Dreibholz, Joseph Adam Schenkel, and Jiaxi Alessia Hu. An Overview of Multi-cloud Computing. In *Web, Artificial Intelligence and Network Applications: Proceedings of the Workshops of the 33rd International Conference on Advanced Information Networking and Applications (WAINA-2019) 33*, pages 1055–1068. Springer, 2019.

[154] Shay Horovitz, Roei Amos, Ohad Baruch, Tomer Cohen, Tal Oyar, and Afik Deri. FaaStest - Machine Learning based Cost and Performance FaaS Optimization. In *Economics of Grids, Clouds, Systems, and Services: 15th International Conference, GECON 2018, Pisa, Italy, September 18–20, 2018, Proceedings 15*, pages 171–186. Springer, 2019.

[155] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 851–877. USENIX Association, February 2020.

[156] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G Patil, Joseph E Gonzalez, and Ion Stoica. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1375–1389, 2023.

[157] Ari Juels. Targeted Advertising... And Privacy Too. In *Cryptographers' Track at the RSA Conference*, pages 408–424. Springer, 2001.

[158] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile Private Contact Discovery at Scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1447–1464, 2019.

[159] Stacy Kanevskaia. 4 ways Azure, AWS, and Google Cloud virtual cards simplify subscription management. `https://karta.io/blog/13-4-ways-azure-aws-and-google-cloud-virtual-cards-simplify-subscription-management`, Dec 2022.

[160] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913, 2016.

[161] Darya Kaviani and Raluca Ada Popa. `https://mpc.cs.berkeley.edu`.

[162] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 1575–1590. Association for Computing Machinery, 2020.

[163] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct 2016.

[164] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. *Advances in Cryptology – EUROCRYPT 2018*, page 158–189, Mar 2018.

[165] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, August 2021.

[166] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892, 2021.

[167] Kari Kostiainen, N Asokan, and Jan-Erik Ekberg. Credential Disabling from Trusted Execution Environments. In *Information Security Technology for Applications: 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers 15*, pages 171–186. Springer, 2012.

[168] Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, Michael Walfish, et al. World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, 2007.

[169] E. Kushilevitz and R. Ostrovsky. Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[170] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally Scaling Strong Anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, page 406–422. Association for Computing Machinery, 2017.

[171] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An Efficient Communication System With Strong Anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016, 08 2015.

[172] Brent Lagesse, Mohan Kumar, Justin Mazzola Paluska, and Matthew Wright. DTT: A Distributed Trust Toolkit for pervasive systems. In *2009 IEEE International Conference on Pervasive Computing and Communications*, pages 1–8. IEEE, 2009.

[173] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πBox: A Platform for Privacy-Preserving Apps. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 501–514. USENIX Association, April 2013.

[174] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for Checking Compromised Credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1387–1403, 2019.

[175] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 717–732. USENIX Association, August 2021.

[176] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.*, 54(10s), Sep 2022.

[177] Yehuda Lindell. How Smart Cryptography Makes Coinbase More Secure. https://www.coinbase.com/blog/how-smart-cryptography-makes-coinbase-more-secure, Oct 2022.

[178] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. The Deployment Dilemma: Merits and Challenges of Deploying MPC. https://mpc.cs.berkeley.edu/blog/deployment-dilemma, Sep 2023.

[179] Joshua Lund. Technology Preview for Secure Value Recovery. https://signal.org/blog/secure-value-recovery/, 2019.

[180] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169, 2017.

[181] Nikolaos Makriyannis and Oren Yomtov. Practical Key-Extraction Attacks in Leading MPC Wallets. *Cryptology ePrint Archive*, 2023.

[182] Eduard Marin, Diego Perino, and Roberto Di Pietro. Serverless Computing: A Security Perspective - Journal of Cloud Computing, Oct 2022.

[183] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, page 1289–1306. USENIX Association, 2017.

[184] David Z. Morris. 4 Unanswered Questions About the Bitfinex Hack. https://www.coindesk.com/layer2/2022/02/09/4-unanswered-questions-about-the-bitfinex-hack/, Feb 2022.

[185] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, et al. Personal Data Management with the Databox: What's Inside the Box? In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 49–54, 2016.

[186] Daniel William Moyer. Punching Holes in the Cloud: Direct Communication between Serverless Functions Using NAT Traversal. Master's thesis, Virginia Tech, Jun 2021.

[187] Graham Mudd. Privacy-Enhancing Technologies and Building for the Future. https://www.facebook.com/business/news/building-for-the-future, Aug 2021.

[188] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against

Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[189] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, Dec 2008.

[190] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[191] Bijeeta Pal, Mazharul Islam, Thomas Ristenpart, and Rahul Chatterjee. Might I Get Pwned: A Second Generation Compromised Credential Checking Service. In *USENIX Security*, 2022.

[192] Shoumik Palkar and Matei Zaharia. DIY Hosting for Online Privacy. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2017.

[193] Dana Petcu. Multi-Cloud: Expectations and Current Approaches. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 1–6, 2013.

[194] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146. USENIX Association, August 2021.

[195] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime Match: A Privacy-Preserving Inventory Matching System. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6417–6434. USENIX Association, August 2023.

[196] PRNewsfoto/Fireblocks. Fireblocks Surpasses $600 Billion in Digital Assets Transferred. https://www.prnewswire.com/news-releases/fireblocks-surpasses-600-billion-in-digital-assets-transferred-301293822.html, 2021.

[197] Joel Reardon, Jeffrey Pound, and Ian Goldberg. Relational-Complete Private Information Retrieval. *University of Waterloo, Tech. Rep. CACR*, 34(2007), 2007.

[198] James Reyes. Building the next generation of digital advertising with MPC. *Real World Crypto*, 2022.

[199] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. Solid: A Platform for Decentralized Social Applications Based on Linked Data. *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.*, 2016.

[200] Randall Sarafa. Introducing Signal PINs. https://signal.org/blog/signal-pins/, May 2020.

[201] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. AdVeil: A Private Targeted Advertising Ecosystem. *Cryptology ePrint Archive*, 2021.

[202] Adi Shamir. How to Share a Secret. In *Programming Techniques R. Rivest Editor*, 1979.

[203] Jimmy Song. Mt. Gox hack technical explanation. https://jimmysong.medium.com/mt-gox-hack-technical-explanation-37ea5549f715, Aug 2017.

[204] Ion Stoica and Scott Shenker. From Cloud Computing to Sky Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 26–32. Association for Computing Machinery, 2021.

[205] Nick Summers. Why you can trust 1Password's cloud-based storage and syncing. https://www.kiplinger.com/personal-finance/lastpass-hack, February 2023.

[206] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multi-Factor Authentication for Distributed-Trust Systems. In *IEEE Symposium on Security and Privacy*, 2023.

[207] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, et al. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, 2019.

[208] Kfir Toledo, Pravein Govindan Kannan, Michal Malka, Etai Lev-Ran, Katherine Barabash, and Vita Bortnikov. ClusterLink: A Multi-Cluster Application Interconnect. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, page 138. Association for Computing Machinery, 2023.

[209] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX Fails in Practice. https://sgaxeattack.com/, 2020.

[210] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov,

Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281. USENIX Association, February 2020.

[211] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical Private Queries on Public Data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313. USENIX Association, March 2017.

[212] Ke Coby Wang and Michael K Reiter. Detecting Stuffing of a User's Credentials at Her Own Accounts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2201–2218, 2020.

[213] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-Scale Secure Multiparty Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 39–56. Association for Computing Machinery, 2017.

[214] Michal Wawrzoniak, Ingo Muller, Rodrigo Bruno, and Gustavo Alonso. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*, January 2021.

[215] Iam Hazel Virginia Whitehouse-Grant-Christ. IAM tutorial: Delegate access to the billing console. https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial_billing.html, 2011.

[216] Josephine Wolff. How a 2011 Hack You've Never Heard of Changed the Internet's Infrastructure. https://slate.com/technology/2016/12/how-the-2011-hack-of-diginotar-changed-the-internets-infrastructure.html, Dec 2016.

[217] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182. USENIX Association, October 2012.

[218] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, et al. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, 2023.

[219] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces

Sara McAllister Yucong "Sherry" Wang Benjamin Berg* Daniel S. Berger†
George Amvrosiadis Nathan Beckmann Gregory R. Ganger

Carnegie Mellon University **UNC Chapel Hill* *†Microsoft Azure and University of Washington*

Abstract

Datacenters need to reduce embodied carbon emissions, particularly for flash, which accounts for 40% of embodied carbon in servers. However, decreasing flash's embodied emissions is challenging due to flash's limited write endurance, which more than halves with each generation of denser flash. Reducing embodied emissions requires extending flash lifetime, stressing its limited write endurance even further. The legacy Logical Block-Addressable Device (LBAD) interface exacerbates the problem by forcing devices to perform garbage collection, leading to even more writes.

Flash-based caches in particular write frequently, limiting the lifetimes and densities of the devices they use. These flash caches illustrate the need to break away from LBAD and switch to the new Write-Read-Erase iNterfaces (WREN) now coming to market. WREN affords applications control over data placement and garbage collection. We present FairyWREN[1], a flash cache designed for WREN. FairyWREN reduces writes by co-designing caching policies and flash garbage collection. FairyWREN provides a 12.5× write reduction over state-of-the-art LBAD caches. This decrease in writes allows flash devices to last longer, decreasing flash cost by 35% and flash carbon emissions by 33%.

1 Introduction

DATACENTER CARBON EMISSIONS are a topic of growing concern. At current emission rates, datacenters' share of global emissions are projected to rise to 20% by 2038 [48] and 33% by 2050 [53]. In the next few decades, many companies — including Amazon [1], Google [2], Meta [11], Microsoft [71] — are looking to achieve Net Zero, i.e., greenhouse gas emissions close to zero. To achieve this goal, many datacenters are adopting renewable energy sources such as solar and wind [11, 39, 64, 71]. Google, AWS, and Microsoft are expected to complete their transition to renewable energy by 2030 [30, 49, 59]. However, this switch in energy source does not reduce datacenters' *embodied emissions*, the emissions produced by the manufacture, transport, and disposal of datacenter components. Embodied emissions will account for more than 80% of datacenter emissions once datacenters move to renewable energy [39].

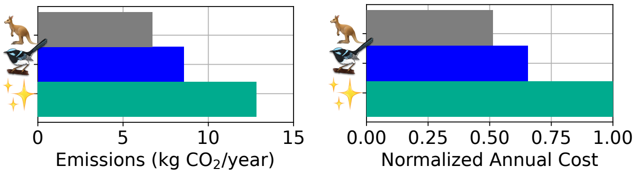


Fig. 1: Carbon emissions and cost for flash in Kangaroo (🦘), FairyWREN (🐦), and "minimum writes" (✨)—an idealized cache with no extra writes—over a 6-year lifetime for a production Twitter trace and a target 30% miss ratio. Compared to Kangaroo, FairyWREN reduces carbon emissions by 33% and cost by 35%.

Embodied emissions are produced by one-time lifecycle events. Datacenters can reduce these emissions by: *(i)* replacing hardware with less carbon-intensive alternatives, and *(ii)* extending the lifetime of components to amortize embodied emissions over a longer period. Recent work has studied embodied emissions in processor design [24, 38, 39, 85], but considerably less attention has been paid to memory and storage, even though they constitute 46% and 40% of server emissions, respectively [64]. It is therefore crucial to both move from carbon-intensive technologies like DRAM to flash, which has 12× less embodied carbon per bit [38], *and* to extend flash lifetimes to amortize flash's embodied carbon.

However, flash introduces a new challenge: *limited write endurance*. A flash device can only be written a limited number of times before it wears out. Each new generation of flash has lower write endurance as a result of manufacturers packing more bits into each cell. This packing, however, does improve sustainability by storing more capacity in the same silicon (i.e., less carbon per bit). To realize the benefits of denser flash, applications must write to flash much less frequently. The write-rate budgets that applications must operate under to achieve longer lifetimes are tiny: to achieve a six-year lifetime on a 2 TB QLC drive, the application can write only 14 MB/s,

[1] Fairywrens (🐦) are vibrant birds native to Australia. Common varieties include Superb Fairywrens, Splendid Fairywrens, and Lovely Fairywrens.

or 0.09% of available write bandwidth (Sec. 2).

Reducing carbon from caching. Hence, write-intensive flash applications present a major challenge in reducing overall datacenter emissions. This paper focuses on reducing carbon from flash caching, an increasingly popular use of flash in the datacenter [3,16,21,22,35,36,83]. We aim to demonstrate, through caching, how to *leverage emerging flash interfaces* to reduce writes, in particular by *re-purposing garbage collection to do useful work.*

Caching is fundamentally write-intensive, as new objects must be frequently admitted to maintain hit rates [15,18]. Datacenter caches also store many small objects [16,67], which is particularly problematic because flash can only be written at a coarse granularity. Because of this mismatch, admitting small objects to the cache can lead to significant *write amplification*: i.e., more bytes are written to the underlying flash device than requested by the application.

Most current flash devices are *Logical Block-Addressable Devices (LBAD)* that present the same block device abstraction used by hard disks. This abstraction hides significant details about how SSDs work. In particular, while the interface allows reading and writing 4KB blocks, the underlying flash device can only erase large (MB to GB) regions. To implement the LBAD interface, the flash firmware performs garbage collection, copying blocks of valid data and erasing entire regions to make room for new writes. Current flash caches have a limited ability to optimize these internal writes, which can amplify the total bytes written by 2× to 10× [67].

Opportunity: WREN. New flash SSD interfaces, such as ZNS [19] and FDP [66], allow closer integration of host-level software and flash management. The key difference between these interfaces and LBAD is that these interfaces include `Erase` as a first-order operation, allowing the cache to control garbage collection. We use the name *Write-Read-Erase iNterfaces (WREN)* to collectively refer to such interfaces, and we describe the necessary and sufficient operations for flash caches to minimize write rate. However, we also show that merely porting existing flash caches to WREN does not reduce flash writes. *Flash caches must be re-designed to leverage the additional control provided by WREN.*

Our solution: FairyWren. We design and implement FairyWren, a flash cache that harnesses WREN to reduce writes. The main insight in FairyWren is that every flash write, whether from the application *or from garbage collection*, is an opportunity to admit objects to the cache. When flash is written during garbage collection, FairyWren can admit objects "for free". This idea cannot be realized on LBAD, since these devices offer no control over garbage collection. FairyWren uses the features of WREN to perform a "nest packing" algorithm on *every* write, *unifying cache admission and garbage collection in a single algorithm.* FairyWren also leverages WREN to enable large-small object separation and hot-cold set-partitioning, further reducing writes.

Summary of results. We find that, without major changes to flash interfaces and cache designs, deploying denser flash will not reduce the carbon emissions of flash caches. *For current caching systems, the reduced write endurance of denser flash outweighs the gains in density.* Only by changing the flash interface and optimizing the cache to this new interface can we realize the significant emissions savings of denser flash.

To illustrate this point, we implement FairyWren as a flash cache module within CacheLib [16]. We evaluate FairyWren on production traces from Meta and Twitter using both simulation and a real ZNS SSD. *FairyWren reduces flash writes by 12.5×* vs. the research state-of-the-art. By enabling caching on denser flash, *FairyWren reduces flash's carbon emissions by 33%* vs. the research state-of-the-art (Fig. 1). FairyWren performs close to an idealized, minimum-write cache on both carbon emissions and cost.

Contributions. This paper contributes the following:

- *Flash trends (Sec. 2):* By studying flash trends, we identify opportunities for more sustainable flash caching as well as challenges that prevent current flash caches from realizing these benefits (Sec. 3).
- *Critical elements of flash interfaces (Sec. 4):* We identify the `Erase` operation and control over garbage collection as the essential features of emerging flash interfaces. We describe tradeoffs and fundamental constraints of flash interfaces, showing that some features are, contrary to prior work, unhelpful for caching.
- *FairyWren (Sec. 5):* FairyWren's key insight is to leverage emerging flash interfaces to unify garbage collection and cache admission as one operation, greatly reducing overall flash writes. FairyWren further reduces writes by partitioning objects by size and popularity (hot vs. cold).
- *Analysis of flash emissions (Sec. 6):* We develop a model to analyze carbon emissions from flash. We show that FairyWren's write reduction allows flash caches to improve sustainability using denser flash for longer lifetimes.

2 Opportunities in flash caching

Flash is an increasingly attractive option for caching [16, 21,22,35,57,67,68,83]. In this section, we discuss how trends in the design of flash devices present growing opportunities to reduce the cost and carbon emissions of caching.

Opportunity 1: *Flash is less carbon-intensive than DRAM, so caches are more sustainable with less DRAM.*

DRAM often makes up 40% to 50% of server cost [58, 79,82] and is no longer scaling (Fig. 2). DRAM also has a large embodied carbon footprint and has large operational emissions due to requiring up to half of system power [38].

Flash is cheaper per-bit, embodies 12× less carbon, and requires less power per-bit than DRAM [38]. Thus, datacenters should use flash over DRAM whenever possible [37], even for traditionally DRAM workloads, such as caching [16,35, 67,68] or machine learning [95].

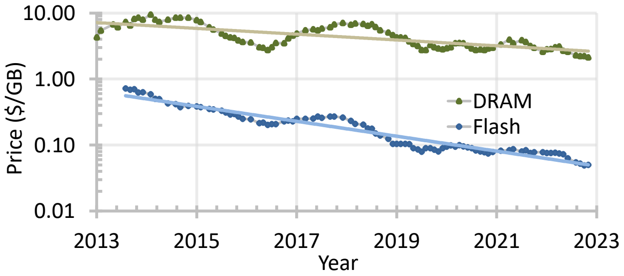


Fig. 2: Cost for flash and DRAM over the last 10 years [4, 6]. Flash prices have decreased over 14×, while DRAM prices have only decreased by ≈2×.

Opportunity 2: *Flash caches should use denser flash where possible to reduce emissions.*

Flash is becoming denser, moving from *single-level cells (SLC)*, which store 1 bit/cell, to *tri-level cells (TLC)*, which store 3 bits/cell. Flash SSDs will soon use *quad-level cells (QLC)* and *penta-level cells (PLC)* [73]. Denser flash is cheaper; e.g., PLC is forecast to be 40% cheaper per-bit than TLC [9]. Denser flash also reduces carbon emissions, since more bits are packed onto roughly the same silicon.

Opportunity 3: *Lengthening device lifetime is an effective way to improve datacenter sustainability.*

Traditionally, datacenter hardware replacement cycles have been around three years [64] due to the rate of improvement in hardware performance and energy efficiency. Today, datacenters deploy devices for longer. Longer replacement cycles have become common due to their cost advantages and the slowing of Moore's Law. For example, Microsoft Azure increased the depreciable lifetime of servers from four to six years [42, 65], and Meta recently started planning for servers to last 5.5 years [12]. Additionally, hyperscalers are finding that servers do not fail quickly: failure rates at Azure have little evidence of increasing before eight years [17, 64].

Moving to longer lifetimes amortizes both cost and embodied carbon. As datacenters shift to renewable energy, they are rapidly reducing operational carbon. As a result, embodied carbon now dominates datacenter carbon emissions [12, 38, 39, 84]. The major challenge, though, is how to extend *flash* lifetime, given its limited write endurance.

3 Challenges in flash caching

Flash SSDs have limited write endurance and are warrantied only for a stated write budget [10]. Exceeding this write budget can cause the device to fail. Hence, while flash caching presents carbon-saving opportunities (Sec. 2), caches must severely limit the amount they write. Here, we discuss the challenges of flash caching in detail and describe how current systems fail to address these challenges.

3.1 Wherefore device write amplification?

Flash devices cannot write new values without first erasing a large region of the device. To support random writes, devices must read all live data in a region, erase the region, and then write the live data back to the drive along with any new data. As a result, flash SSDs perform more writes than requested by the application. The *device-level write amplification* (DLWA) [23, 35, 41, 54, 57, 62, 83] captures this relative increase in bytes actually written to flash vs. bytes written by an application. (If an SSD writes 3GB to serve 1GB of application writes, then DLWA is 3×.) DLWA can be large: a factor of 2× to 10× is common [67]. DLWA causes write-intensive applications to quickly wear out flash devices, increasing their replacement frequency and embodied emissions over time.

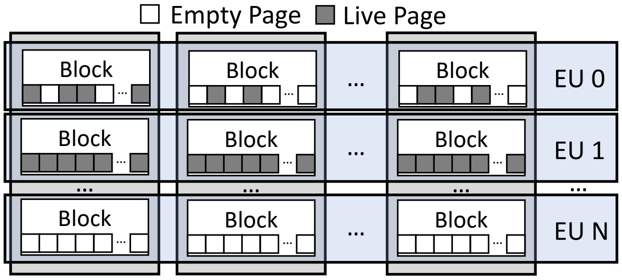


Fig. 3: The internal arrangement of flash devices into planes, blocks, pages, and EUs. Each EU has blocks in multiple pages. EU 0 is a partially full, EU 1 is entirely full, and EU N has just been erased.

DLWA is primarily caused by the physical limitations of flash storage. Flash devices are organized in a physical hierarchy (Fig. 3). The smallest unit is the *page*, usually 4 KB. Flash can be written at page granularity, but a page must be erased before it can be rewritten. To avoid electrical interference during erasure, pages are grouped into *flash blocks* [13, 19, 20, 41, 63]. A flash block is the minimum erase size. In practice, however, flash drives stripe writes across blocks to improve bandwidth and error correction. Striping increases the effective *erase unit (EU)* size to gigabytes [19].

The mismatch between the granularity of writes and erases is the root cause of DLWA. To maintain the 4 KB read/write block interface, flash devices garbage collect (GC), moving live pages from partially empty EUs (such as EU 0 in Fig. 3) to a writable EU (such as EU N) before erasing the EU and freeing dead pages. The less the available capacity on the device, the more frequently it has to GC, introducing a tradeoff between flash utilization and flash writes.

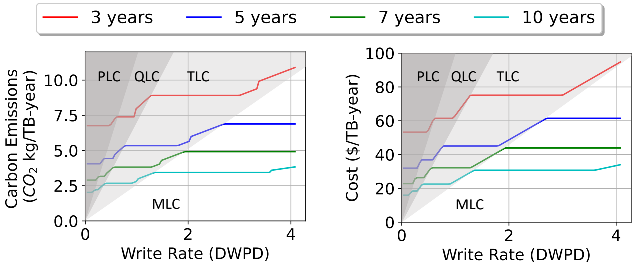
One might hope that technological advances would decrease EU sizes, closing the gap between write and erase granularities. However, *flash EU sizes have gotten larger as flash has gotten denser*. Effective block sizes on an SLC flash device were 128 KB [86], MLC and TLC flash devices are around 20 MB [81], and QLC devices will be 48 MB [80]. Striping these blocks with hundreds of 3D-stacked layers [80] results EUs in the gigabyte range [19, 69].

Lesson for flash caches: Write amplification is caused by the size mismatch between writes and erases in flash. This mismatch will keep increasing.

3.2 Denser flash has lower write endurance

As flash becomes denser, its write endurance drops significantly. For example, while PLC flash is up to 40% denser than TLC, PLC is forecast to have only *16%* of TLC's writes [9]. Additionally, because denser flash has to differentiate between more voltage levels, even small voltage changes can make data unreadable. TLC uses two-phase writes and more frequent refresh to prevent data loss [70]. Two-phase writes require the device to have enough RAM and capacitance to remember all in-flight writes, limiting the number of EUs that can be "active" (i.e., writable) at any point in time, often to less than ten. Writing to more EUs than this requires closing an active EU, incurring more internal device writes.

Fig. 4 models how write rate affects both emissions and cost when varying lifetimes and flash density. Each line shows a device of a different lifetime, and shaded regions show which flash density is best for a given write rate. The model calculates how much capacity must be provisioned for each technology to achieve the desired lifetime at a given write rate. For example, a device lasting 7 years (green) has lower annualized carbon emissions than one lasting 3 or 5 years, and it should use dense flash (e.g., TLC) only at write rates below two device-writes-per-day.



(a) Carbon emissions.

(b) Cost.

Fig. 4: The annual carbon emissions and cost of flash depending on the required average write rate and desired lifetime.

Lesson for flash caches: Device lifetime is the most important factor in reducing carbon emissions. Moreover, denser flash can improve sustainability, but only if flash write rate is very small — much less than one device-write per day.

3.3 Shortcomings of existing solutions

To limit embodied emissions, sustainable flash caches must minimize *(i)* idle flash space — which incurs emissions for no benefit; *(ii)* DRAM usage for object metadata — which can add up to tens of GBs [35, 67]; and *(iii)* flash write rates — which wear out the device, reducing lifetime. No prior flash-cache design meets these criteria (Table 1). In particular, although caches must admit new objects to maintain hit rates, flash caches must be designed to minimize application- and device-level write amplification to extend device lifetime.

Flash caches ≠ DRAM caches. Both flash caches and DRAM caches try to reduce misses, but flash caches must also contend with flash's limited write endurance, leading to much different designs. Flash caches are designed to achieve low end-to-end write amplification, i.e., the product of *application-level write amplification* (ALWA) (e.g., from having to write 4KB to flash to admit a 100B object) and DLWA.

	Flash caches should minimize ...			
	Unused flash	**DRAM**	**ALWA**	**DLWA**
Key-value stores	✗	✓	✓	✓
Log-structured caches	✓	✗	✓	✓
Set-associative caches	✗	✓	✗	✗
Kangaroo [67]	✓	✓	✓	✗
FairyWren	✓	✓	✓	✓

Table 1: Comparison of FairyWren vs. prior cache designs. FairyWren is the only design to minimize all important overheads.

Flash caches ≠ key-value stores. KV stores [5, 7, 33, 55, 60, 75, 90] support a similar read-write interface as caches and likewise minimize flash writes and DRAM overhead. However, flash caches have significantly different design goals.

The main difference is that `delete` operations are uncommon in KV stores, but very frequent in caches. Caches frequently evict objects and must reclaim space immediately to admit new objects [67]. Most KV stores do not support deleting objects quickly enough to implement cache eviction policies. Specifically, standard KV store data structures like LSM trees [5, 7, 31, 32, 60, 75, 90] will not work well for caching unless the KV store is massively overprovisioned, often by more than 2× the cache capacity [21, 22, 83].

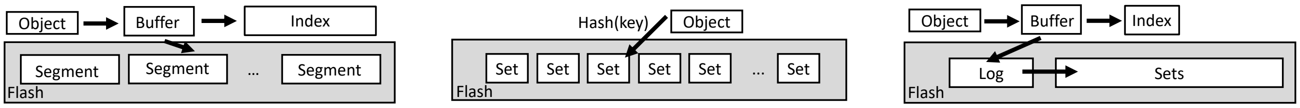
Moreover, KV stores do not exploit a cache's biggest advantage: caches are free to evict objects whenever it is convenient. Evicting objects opportunistically can greatly reduce writes and maximize space utilization, but KV stores are not built to exploit this cache-specific optimization.

Existing flash caches do not address DLWA. Because of the unique challenges of flash caching, there is a growing body of work devoted to improving flash cache designs. Prior flash caches generally fall under three categories (Fig. 5): log-structured, set-associative, and hierarchical.

Log-structured caches. To minimize writes, many flash caches are log-structured [16, 27, 35, 83]. These caches append objects to an on-flash log (Fig. 5a), locating objects through a DRAM index and evicting objects in large groups. The log allows large sequential writes to flash and thus achieves nearly ideal write amplification.

While log-structured caches work well for larger objects, the DRAM index becomes prohibitively large for small objects, even if it is highly optimized [67], significantly increasing overall emissions and cost (see Fig. 11). Flash caches are thus often partitioned, using a log-structured cache for large objects and a different design for small objects [16].

Set-associative caches. Set-associative caches, such as the Small Object Cache in Meta's CacheLib [16], replace the DRAM index with a hash function that maps each object to a unique set (usually a 4 KB page) on flash (Fig. 5b).



(a) Log-structured *(b) Set-associative* *(c) Kangaroo*

Fig. 5: Designs of prior flash caches: (a) Log-structured caches write objects segments to flash sequentially, (b) Set-associative cache write objects to a set based on the key's hash, and (c) Kangaroo is a hierarchical design that combines a log-structured and a set-associative cache.

The downside of these caches is that they cause significantly more writes. When a set-associative cache admits a small object (say, 100 B), it must write at least one flash page (4 KB), resulting in large ALWA (40×). Even worse, these caches perform random writes, leading to DLWA of 2× to 10× [67]. Since write amplification (WA) is the product of ALWA and DLWA, *a set-associative cache's WA easily exceeds 100×*. To mitigate this, Meta's flash caches use only 50% of the drive [16], increasing miss ratio and carbon emissions.

Hierarchical. FairyWREN builds on Kangaroo [67, 68], a hierarchical flash cache for small objects that combines a small log-structured cache (KLog) and a large set-associative cache (KSet) (Fig. 5c). Kangaroo uses KLog to reduce ALWA and KSet to reduce DRAM. Objects are first buffered in KLog and then admitted in batches to KSet, amortizing its ALWA across several admitted objects. KSet comprises more than 90% of the cache capacity, limiting the DRAM needed to index KLog. Kangaroo also includes a selective admission policy to reduce flash writes and a partitioned index data structure to reduce DRAM. Due to its low DRAM overhead, Kangaroo achieves large emission reductions over a memory-optimized log-structured cache, Flashield [35], for workloads with many small objects (Fig. 11 in Sec. 6.2).

While Kangaroo optimizes both DRAM and ALWA, it still has too many writes because *Kangaroo does not address device-level write amplification.* KSet performs random 4 KB writes, the worst case for DLWA. As a result, Kangaroo's emissions do not reduce with denser flash. Fig. 4 shows that, for a 10-year lifetime, QLC requires fewer than 0.37 device-writes per day (DWPD) and PLC requires fewer than 0.16 DWPD, whereas Kangaroo performs 1.46 DWPD in our evaluation.

4 Write-Read-Erase iNterfaces (WREN)

Prior flash caches incur excessive DLWA (Sec. 3). The root causes are the mismatch between write and erase granularities and a legacy LBAD interface that hides this mismatch from software. This section discusses recent *Write-Read-Erase iNterfaces (WREN)*, such as ZNS [19] and FDP [66], that include `Erase` as a first-order operation. We show that WREN is necessary but insufficient: a new flash interface does not reduce writes by itself, changes to the cache design are required.

4.1 Today's interface is LBAD

Most flash SSDs today are *logical block addressable devices (LBAD)*, sharing the same interface as disks. LBAD presents the flash device as a linear address space of fixed-size blocks[2] that can be independently read or written.

LBAD eased the transition from HDDs to SSDs, but does not expose the erase granularity of flash (Sec. 3). As a result, the LBAD device firmware must perform garbage collection (GC) that can cause high DLWA and tail latency. Although there has been work to decrease DLWA [40, 41, 44, 56, 89, 91], LBAD devices still hide erase units and GC from applications, preventing co-optimization to minimize overall flash writes.

4.2 Challenges of new interface design

While a variety of flash interfaces have been proposed [20, 44, 51, 52, 72, 78, 88, 96], none have gained widespread adoption. Two proposals, Multi-streamed SSDs and Open-Channel SSDs, illustrate the pitfalls of designing a new flash interface.

Multi-streamed SSDs [51, 52] allow users to direct writes to different *streams*. Streams provide isolation between workloads: different streams write to different EUs. When objects with similar lifetimes are grouped into the same stream, GC is more efficient. However, because the application does not control GC directly, DLWA remains a significant issue.

Open-Channel SSDs [20] remove all flash-device logic and force applications to handle *all* of flash's complexities, even beyond those described in Sec. 3. While the hope was to develop layers of abstraction in software to hide some of this complexity, this software was never widely deployed.

Lesson for flash caches: An ideal flash interface for caching would allow the cache to control *all* writes, including GC, but still present a simple abstraction to application developers.

4.3 What makes an interface WREN?

We call interfaces that delegate `Erase` commands and garbage collection to the host *Write-Read-Erase iNterfaces (WREN)*. WREN is defined by three main features:

1) WREN operations. WREN devices must let applications control which EU their data is placed in and when that EU is erased. Specifically, WREN devices must, at least, have `Write`, `Read`, and `Erase` operations.

These operations can be implemented differently. For example, *Zoned Namespaces* (ZNS) [19] and *Flexible Data Placement* (FDP) [66] are both WREN. Both interfaces are NVMe standards with strong support from industry and provide an abstraction for writing to an EU[3]. However, they have different philosophies, which can be seen, for instance, in their `Write` operations. ZNS provides either sequential writes to an EU or nameless writes through Zone Append [96]. FDP

[2] These fixed-size blocks correspond to pages, not flash blocks (Sec. 3)

[3] This abstraction is called a *zone* in ZNS and a *reclaim unit* in FDP.

provides random writes within an EU as long as the application tracks that the number of pages written is less than the EU size. Despite these differences, both provide the control over data placement into EUs required by WREN.

Moreover, the aforementioned Open-Channel interface is also WREN. But Open-Channel SSDs expose the full complexity of the device to the host, which is additional complexity *not* required to reduce a cache's DLWA.

2) The `Erase` requirement. Unlike LBAD, WREN devices do not move live data from an EU before erasing it. Applications are responsible for implementing GC to track and move live data before calling `Erase`. `Erase` is different from a traditional `trim` because `Erase` targets an entire EU rather than individual pages. Failure to perform correct and timely GC is subject to implementation-specific error handling by the device. A major difference between FDP and ZNS is how they treat violations of `Erase` semantics, but this error behavior is inessential to reducing DLWA and thus beyond WREN.

3) Multiple, but limited, active EUs. An *active EU* is one that can be written to without being erased. WREN devices support a few active EUs at one time. Since an active EU typically requires a device buffer for the EU's data, the maximum number of active EUs is implementation-specific. FairyWREN requires four simultaneously active EUs, which we expect will be supported in the vast majority of WREN devices.

4.4 WREN alone is not a cure for WA

WREN devices make it easy to perform large, sequential writes with no WA. When writing sequentially, the user can maintain a single active EU and fill the EU completely before activating the next EU. Furthermore, if all writes are large and sequential, it is generally easy to find an EU consisting of invalid data when GC is required, resulting in low WA.

Set-associative flash caches also want low WA for small, random writes, which incur high DLWA on LBAD devices. One might hope that WREN devices can achieve lower WA. A reasonable first attempt at implementing a set-associative cache on WREN is to treat each set as an object in a log-structured store, allowing the cache to write updates sequentially to a single active EU. We find that this naive approach does not reduce WA because it just moves the GC from the device to the cache.

The impact of smaller EUs. One idea for mitigating WA under small, random writes is to reduce the EU size, e.g., from a GB to tens of MB, by removing error correction between flash blocks, which caches can tolerate. Prior systems literature uses smaller EUs to minimize GC [14, 69] because, intuitively, lowering the number of sets per EU creates more EUs that are either mostly invalid (good candidates for GC) or mostly valid (bad candidates for GC that are skipped). However, other prior work that analyzes the WA of FIFO GC policies [34, 46] has largely ignored the effect of EU size. In fact, this modeling work assumes that changing the EU size will not change the WA from GC. To remedy this discrepancy in prior work, we model the WA of a FIFO GC policy for a set-associative cache, capturing the effect of EU size.

Following the approach of [46], we approximate the distribution of the number of live pages in the EU at the tail of a log-structured store (see Appendix A for details). Our approximation shows that when EU sizes are small, FIFO is more likely to find EUs that are mostly invalid or completely valid. To quantify this effect, we approximate the long-run average WA under FIFO. Our approximation (Fig. 6) matches simulation results, with a R^2 value of 0.9996.

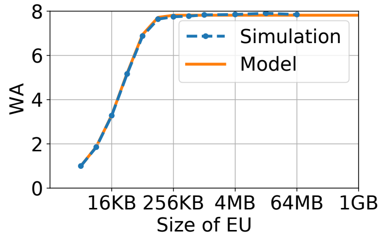


Fig. 6: The DLWA for a set-associative cache running on WREN with 7% overprovisioning. EUs have to be less than 128 KB to significantly reduce DLWA.

Lesson for flash caches: We find that *reducing EU size only improves WA for very small EU sizes*. To realize a significant reduction in WA, the EU size must be tens of KBs, but that is unachievable in current devices (Sec. 3). Hence, we conclude that WREN alone does not reduce WA for caches. To reduce WA, we must also re-design the cache.

5 FairyWREN Overview and Design

FairyWREN uses WREN to substantially reduce WA by unifying cache admission with garbage collection. The resulting reduction in overall writes lets FairyWREN use denser flash while extending device lifetime to improve sustainability.

5.1 Overview

How FairyWREN reduces writes. FairyWREN uses WREN's control over data placement and garbage collection to reduce writes in two main ways. First, FairyWREN introduces *nest packing* to combine garbage collection with cache admission and eviction. When live data is rewritten during GC, FairyWREN has an opportunity to evict unpopular objects and admit new objects in their place. In LBAD, by contrast, these objects would have to be rewritten separately for GC and admission/eviction.

Second, FairyWREN groups data with similar lifetimes into the same EU, separating data that in prior caching systems would have been in the same page. If all of the data in each EU has roughly the same lifetime, EUs will either consist mostly of live data or mostly of dead data. FairyWREN can then GC the mostly dead EUs with few additional writes. FairyWREN leverages two main techniques to enable this grouping: large-small object separation and hot-cold set partitioning.

Architecture of FairyWREN. FairyWREN partitions its capacity into a large-object cache (LOC) and a small-object cache (SOC), as seen in Fig. 7. Incoming requests first check the LOC and then check the SOC.

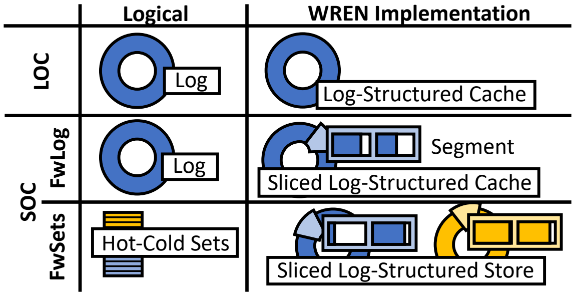


Fig. 7: The components of FairyWREN.

The *large-object cache* (Sec. 5.2) stores objects larger than 2 KB and uses a simple log-structured design, since it can tolerate higher per-object DRAM overhead.

The *small-object cache* (Sec. 5.3) uses a hierarchical design based on Kangaroo [67]. The SOC contains two levels: FwLog and FwSets. FwLog is a log-structured cache with a relatively high per-object DRAM overhead. The main function of FwLog is to buffer objects so they can be written efficiently to FwSets. Therefore, FwLog can have a fairly low capacity ($\approx 5\%$), keeping its DRAM overhead low. FwSets is a set-associative cache, but, since WREN does not support random writes, the sets are kept in a log-structured store. FwSets stores sets, not individual objects, in the log to minimize DRAM. When this log-structured store is garbage collected, objects are opportunistically moved from FwLog into FwSets. Finally, each set in FwSets is further partitioned into hot (frequently accessed, long-lived) objects and cold (recently admitted, short-lived) objects (Sec. 5.4).

5.2 The LOC

The LOC is a log-structured cache. Adapting log-structured caches to WREN is straightforward, since they only perform large, sequential writes. The LOC is broken into large segments, each the size of an EU. Segments can then be evicted in LRU or FIFO order with minimal WA. The LOC uses DRAM in two ways: *(i)* an in-memory, EU-sized buffer for log insertions, and *(ii)* an in-memory index tracking object locations on flash. Because the LOC stores large objects, it contains relatively few objects and needs little DRAM. Besides the segment buffer, all LOC objects are stored on flash.

Insertions. Objects are first inserted into an in-memory segment buffer and added to the in-memory log index. Once the segment buffer is full, it is written to an empty EU in the log.

Lookup. Reads look up the object's key in the log index. If found, the cache reads the object from the indicated EU.

Eviction. Eventually, the log will fill up and LOC will evict a log segment based on the eviction policy. Since log segments are aligned to EUs, eviction simply `Erase`s an EU, evicting those objects from the cache. This design does not rewrite any objects, incurring minimum WA of 1×.

5.3 The SOC

The focus of FairyWREN is the SOC. Log-structured caches are impractical for caching small objects because a large flash cache can fit billions of small objects, requiring a large DRAM index to track them all (Sec. 3.3). FairyWREN's SOC is based on Kangaroo [67], a recent flash cache designed for small objects with low DRAM overhead and low ALWA. The SOC is a hierarchy of two levels: FwLog, a small log-structured cache, and FwSets, a large set-associative cache. FwLog contains about 5% of the SOC's capacity, with the remaining 95% for FwSets. We describe FwLog and FwSets individually, and then how they work together.

FwLog design. FwLog's goal is to buffer new small objects for insertion into FwSets. Like the LOC, FwLog is a log-structured cache that uses an in-memory segment buffer and an in-memory index to track objects in the FwLog. All other objects in the FwLog are stored on flash.

FwSets design. FwSets is a set-associative cache that maps each object to a unique set by hashing its key. When admitting an object, FwSets evicts old objects from the object's set then overwrites it. However, overwriting is impossible in WREN, so FwSets stores *the sets themselves* as objects in a log-structured store. FwSets uses an in-memory index to track the location of each set on flash, but, unlike prior work [56,61,78], it does not track individual objects, since this would incur too much DRAM overhead. The index's DRAM overhead is low because a set is at least 4 KB, whereas objects can be just 10s of bytes. (Larger sets reduce the size of FwSets's DRAM index, but increase average read latency.)

When FwSets's log-structured store is close to full, it must garbage collect in order to admit new objects to the cache. The simplest scheme would be to erase the EU at the tail of the log, evicting all sets — and thus their objects — mapped to this segment[4]. However, since each set contains a mixture of popular and unpopular objects, throwing away entire sets would significantly increase miss ratio. Instead, FwSets rewrites live sets during GC before erasing the EU.

SOC operation. FwLog and FwSets operate as a hierarchy:

Lookup. Lookups first check FwLog for the object. If not found, FwSets hashes the object's id and looks up the *set's* location. The set is read and scanned for the object.

Insertion. FairyWREN first inserts objects into FwLog. When FwLog is full, objects are evicted from FwLog and inserted into FwSets, as described next. Similarly, inserting into FwSets can cause cascading eviction from FwSets.

Eviction (nest packing). If either FwLog or FwSets is running out of space, FairyWREN needs to perform *nest packing* (Fig. 8). FairyWREN's SOC chooses an EU for eviction from FwLog or FwSets, depending on which is full. If both logs are full, FwSets is chosen because FwSets must have space

[4] In this scenario, FwSets would be on a log-structured cache.

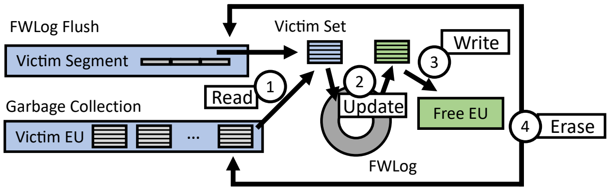


Fig. 8: Nest packing in FairyWREN's small-object cache.

to receive objects evicted from FwLog.

The victim EU is first read into memory. If evicting from FwLog, each object in the EU hashes to a *victim set*. Otherwise, when evicting from FwSets, each set in the EU *is* a victim set. Then, ① FairyWren rewrites each victim set by: ② finding all objects in FwLog that map to a given set, forming a new set containing these objects (evicting objects as necessary), and ③ rewriting the set by appending it to FwSets's log. Finally, ④ FairyWren erases the victim EU.

SOC design rationale. Prior flash caches relied on LBAD GC to reclaim flash space from evicted sets, causing DLWA. *The key difference of FairyWren from prior flash caches is its coordination of cache insertion and eviction with flash GC.*

FairyWren's nest packing algorithm combines previously distinct processes. LBAD caches pay for eviction as ALWA and for garbage collection as DLWA. In the worst case, a set is copied by garbage collection and then is immediately rewritten to admit objects from FwLog. It is impossible to merge these flash writes in LBAD. FairyWren leverages WREN to eliminate unnecessary writes by aligning the eviction and garbage collection cadences of FwLog and FwSets.

5.4 Optimizing the SOC

The SOC is the main source of DRAM overhead and WA in FairyWren. We employ a variety of optimizations to improve the memory and write efficiency of the SOC.

Reducing flash writes by separating hot and cold objects. Even after using nesting to decrease writes, FwSets is still the primary source of flash writes in FairyWren. FairyWren further reduces writes by separating objects by popularity, as determined by a modified RRIP algorithm [45, 67]. Instead of a set being *one unit* that is written every insertion, each set in FwSets is split in twain, into a subset for popular objects and a subset for unpopular objects, each backed by its own log-structured store. Each subset is at least a page. Paradoxically, since the unpopular objects are most likely to be evicted, the subsets with unpopular objects correspond to hot (i.e., frequently written) pages on flash. Hence, we refer to the subsets with unpopular objects as *hot subsets* and we refer to the subsets with popular objects as *cold subsets*.

With hot and cold subsets enabled, objects evicted from FwLog are inserted into the hot subset. The cold subset is not typically written during insertion. Every *n* nest packing operations on a subset, both the hot and cold subsets are read. In memory, these subsets are merged and redivided by object popularity, as seen in Fig. 9. Any popular objects found in the hot subset are moved into the cold subset, since these objects are likely to remain in the cache for longer and do not need to be rewritten as frequently. The least popular objects found in the cold subset are moved into the hot subset so that FwSets can evict them if they remain sufficiently unpopular.

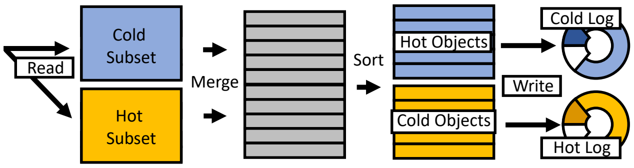


Fig. 9: FwSets is split in two: hot subsets with cold objects and cold subsets with hot objects. Most of the time objects are inserted into the hot subset. However, every n subset updates, both subsets are read, merged, split by object popularity, and then both rewritten.

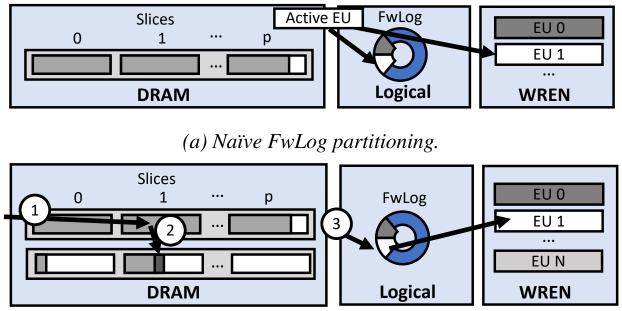
Hot-cold object separation can nearly halve FwSets's write amplification. If *n* is 5 and sets are 8 KB (two 4 KB subsets), FairyWren without hot-cold object separation would have to write all 8 KB on each insertion to a set. With hot-cold object separation, FairyWren writes 4 KB for the hot subset on every insertion, but only has to write 4 KB for cold subset on every fifth insertion. Thus, FwSets writes only 24 KB instead of 40 KB every five inserts to a set, a 40% reduction.

Theoretically, FairyWren could further reduce writes by further dividing sets. However, there are some practical limitations to this, namely that WREN devices only support a limited number of active EUs, often less than 10. FairyWren currently needs 4 active EUs: 1 for LOC, 1 for FwLog, and 2 for FwSets. Using only 4 active EUs allows FairyWren to run concurrently with other programs on the flash without interference and ensures compatibility with a wide range of WREN devices while still achieving low write rates.

Moreover, separating objects by popularity yields diminishing returns since it increases miss ratio, which will then require more cache capacity to reduce the miss ratio. Wrong object-popularity predictions, which are frequent since very few bits of metadata are used to track each object's popularity, can lead to increases in both writes and miss ratio. The miss ratio will increase if popular objects are placed in hot subsets and evicted prematurely. This type of error becomes more frequent as one tries to separate objects by popularity at finer ganularity. In fact, even our single layer of hot-cold separations causes a modest increase in miss ratio (Sec. 6.5).

Minimizing DRAM in FwLog by slicing. Like Kangaroo [67, 68], FwLog is implemented as 64 *slices*, i.e., 64 independent log-structured caches that operate in parallel over subsets of the keyspace. This is done to save $\log_2 64 = 6$ bits per flash pointer in the DRAM index.

A naïve implementation of slicing on WREN would require one active EU for each slice. Many WREN devices do not permit 64 simultaneously active EUs due to the prohibitively large DRAM overhead this would impose on the flash device. Instead, FwLog uses a single active EU and shares segments among all 64 slices, giving each slice an equal static share of



(a) Naïve FwLog partitioning.

(b) Inserting to FwLog with overflow memory buffering.

Fig. 10: FwLog *uses partitioning to minimize memory and overflow buffers to ensure the log segments are full.*

each segment (Fig. 10a). The downside of sharing FwLog segments is that one slice could fill up its share of the segment before the others. In the worst case, one slice fills before the others contain any objects, causing internal fragmentation in FwLog. This fragmentation reduces FwLog's ability to minimize WA in FwSets. Via simulation and stochastic models, we found that fragmentation could exceed 20%.

FwLog reduces fragmentation via double buffering (Fig. 10b). On insertion, FwLog ① attempts to insert an object into its slice in the "primary" segment buffer. If the primary is full, ② the object is inserted into its slice in the secondary, "overflow" segment buffer. ③ When any slice in the overflow buffer becomes more than half full, FwLog writes the primary buffer to flash. The overflow buffer then becomes the new primary buffer and vice versa. Double buffering increases the number of objects seen before a buffer is written, reducing the variance in the number of objects in each slice. Using balls-and-bins [74] to approximate the maximum objects in a slice, we find that this optimization limits the capacity loss from fragmentation to <1%, even for small (16 MB) buffers.

Minimizing DRAM in FwSets by slicing. Like FwLog, FwSets also slices the log-structured store to reduce DRAM overhead, sharing segments to minimize active EUs and segment buffers. However, since sets are much larger than individual objects, FwSets is more susceptible to internal fragmentation than FwLog. FwSets therefore uses only 8 slices.

Reducing DRAM in FwSets by using larger sets. Finally, FwSets further reduces DRAM by using sets larger than 4 KB, reducing the number of sets that need to be tracked proportionally. Naïvely, one might expect that increasing set size would increase flash writes. In a pure set-associative cache, this would be true. However, FwLog buffers objects, and the number of objects that hash to a set also increases proportionally with set size, so FwSets's writes are roughly independent of set size. We see only a 5% increase in WA when going from 8 KB to 16 KB sets with a 4 KB hot subset and a 12 KB cold subset.

DRAM overhead breakdown. Compared to a LBAD set-associative cache, FwSets requires additional DRAM to track sets. Hot-cold object separation compounds this effect, doubling the number of (sub)sets to track.

Component	Kangaroo	Naïve SOC	SOC
Log total	*48 bits/obj*	*48 bits/obj*	*48 bits/obj*
Set index	–	≈ ***3.1 b***	≈ ***1.4 b***
Sets (other)	4 b	4 b	4 b
Sets total	*4 bits/obj*	*7.1 bits/obj*	*5.4 bits/obj*
Log metadata	≈ 0.8 b	≈ 0.8 b	≈ 0.8 b
Log size	5% = 2.4 b	5% = 2.4 b	5% = 2.4 b
Set size	95% = 3.8 b	95% = 6.7 b	95% = 5.1 b
Total	**7.0 bits/obj**	**9.9 bits/obj**	**8.3 bits/obj**

Table 2: Kangaroo and FairyWren's SOC's DRAM overhead for a 2 TB small-object cache with a 5% log. Despite tracking sets, FairyWren's SOC still needs fewer than 10 bits per object.

Table 2 shows the per-object DRAM overhead for Kangaroo and FairyWren's SOC. Due to partitioning and double buffering, FairyWren achieves the same log overhead as Kangaroo. FairyWren's added overhead shows up in FwSets. Naïvely, when FairyWren has 4 KB subsets and 200 B objects, each set would need 8 bytes, for 3.1 bits/obj. However, since FairyWren uses 8 KB subsets and slices FwSets in eighths, FwSets needs just 1.4 bits/obj to track sets.

FairyWren uses 19% more DRAM than Kangaroo, a 1.5 GB DRAM overhead increase for a 2 TB cache. However, FairyWren's DRAM overhead is still much lower than a log-structured cache, and this modest DRAM increase allows FairyWren to greatly decrease flash writes (by 12.5×), netting large savings in carbon emissions and cost.

6 Evaluation

We compare FairyWren to prior flash caches and find that: (1) FairyWren reduces flash writes by 92% over the research state-of-the-art Kangaroo, leading to a 33% carbon reduction and a 35% cost reduction, (2) FairyWren is within 11% of the minimum write rate, and (3) FairyWren is the first cache design to actually benefit from QLC.

6.1 Experimental setup and model

Implementation. We implement FairyWren in C++ as a module in CacheLib [16]. All experiments were run on two 16-core Intel Xeon CPU E5-2698 servers running Ubuntu 18.04 with 64 GB of DRAM, using Linux kernel 5.15. For WREN experiments, we use a Western Digital Ultrastar DC ZNS540 1 TB ZNS SSD, using the LOC and ZNS library

Parameter	FairyWren	Kangaroo
Interface	WREN (ZNS)	LBAD
Flash capacity	400 GB	400 GB
Usable flash capacity	383 GB	376 GB
LOC size	10% of flash	10% of flash
SOC log size	5% of SOC	5% of SOC
SOC set size	4 KB hot, 4 KB cold	4 KB
Hot-set write frequency	every 5 cold set writes	
Set over-provisioning	5%	

Table 3: FairyWren and Kangaroo experiment parameters.

	SLC	MLC	TLC	QLC	PLC
Write endurance	4.4×	4×	1×	0.32×	0.16×
Capacity discount	3×	1.5×	1×	0.75×	0.6×

Table 4: Scaling factors for different flash densities. We optimistically assume that increasing the bits per cell does not affect emissions or cost.

written by Western Digital [50]. The ZNS SSD has a zone (EU) capacity of 1077 MiB. The devices support 3.5 device writes per day for an expected 5-year lifetime.

We compare to Kangaroo [67] over the first ≈2.5 days of a production trace from Meta. FairyWren uses a ZNS SSD and Kangaroo uses an equivalent LBAD SSD with similar parameters (Table 3). Both caches use 400 GB of flash capacity and achieve similar miss ratios as Kangaroo's production experiments [67]. We overprovision FwSets by 5% to ensure forward progress during nest packing, giving several free EUs to the FwSets log-structured store. Thus, FairyWren effectively uses 383 GB. This idle capacity should decrease in larger flash devices. Kangaroo only uses 376 GB of capacity due to device-level overprovisioning. We approximate Kangaroo's DLWA based on results in the Kangaroo paper [67].

Simulation. In addition to flash experiments, we implemented a simulator to compare a much wider range of possible configurations for FairyWren. The simulator replays a scaled-down trace to measure writes and misses from each level of the cache, including the LOC, FwLog, and FwSets.

We evaluate our cache in simulation on a 21-day trace from Meta [16] and a 7-day trace from Twitter [92]. The Meta trace accesses 6 TB of unique bytes with a 13.8% compulsory miss ratio and an average object size of 395 bytes. Small objects (<2 KB) are 95.2% of requests, and these requests account for 60.2% of bytes requested. The Twitter trace accesses 3.5 TB of unique bytes, has a 17.2% compulsory miss ratio, and an average object size of 265 bytes. Small objects are >99% of requests, and these requests account for >99% of bytes requested. Both of these traces are higher fidelity than the open-source traces [16, 92]. We present results for the last 2 days of the trace.

Carbon emissions and cost model. We evaluate carbon emissions and cost while varying cache configuration, flash density, and device lifetime. We assume that a flash device will have the same caching workload for its entire lifetime and that flash write endurance is the main lifetime constraint. We normalize all results based on device lifetime and we assume that all required flash is purchased at the beginning of deployment.

We estimate emissions and cost from the total flash needed to cover both the cache's capacity and its writes over the desired lifetime. For example, a 2 TB cache with a 6-year lifetime will require at least 2 TBs of flash, but it may require 2.5 TB of flash to accommodate the cache's write rate over 6 years. LBAD devices use 7% overprovisioning, the standard on datacenter drives [8].

We base our write endurance and cost projections on Micron 7300 NVMe U.2 TLC SSDs. For other densities, we multiply the TLC write endurance by the write-endurance factors in Table 4, based on [9]. For cost, we interpolate linearly between flash capacities and include power as the operational expense. Cost is normalized to Kangaroo with a 30% miss ratio for the Twitter trace and 20% for Meta. We optimistically assume that different flash densities will have the same cost and emissions per cell; e.g., 1 TB of PLC has the same emissions as 600 GB of TLC (5:3 ratio). Our model can incorporate more data on denser flash if it becomes available.

We use the ACT model [38] to estimate operational and embodied emissions from CPUs, DDR4 DRAM, and flash. We assume the grid is a 50/50 mix of wind and solar, a common renewable-energy mix [12].

6.2 Carbon emissions of flash caches

We first examine the carbon emissions of different flash caches for a 6-year deployment. Fig. 11 compares FairyWren to three systems: Minimum Writes, Kangaroo, and a Flashield-like log-structured cache [35]. Minimum Writes is an unachievable, idealized cache with WA of 1× and no DRAM overhead. Flashield also assumes a WA of 1×, but requires a DRAM:SSD capacity ratio of 1:10, as originally proposed. Since we cannot faithfully replicate Flashield's ML eviction policy (and no working implementation is available), we assume that Flashield achieves FairyWren's miss ratios.

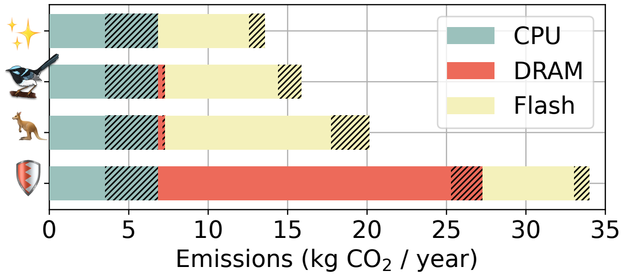
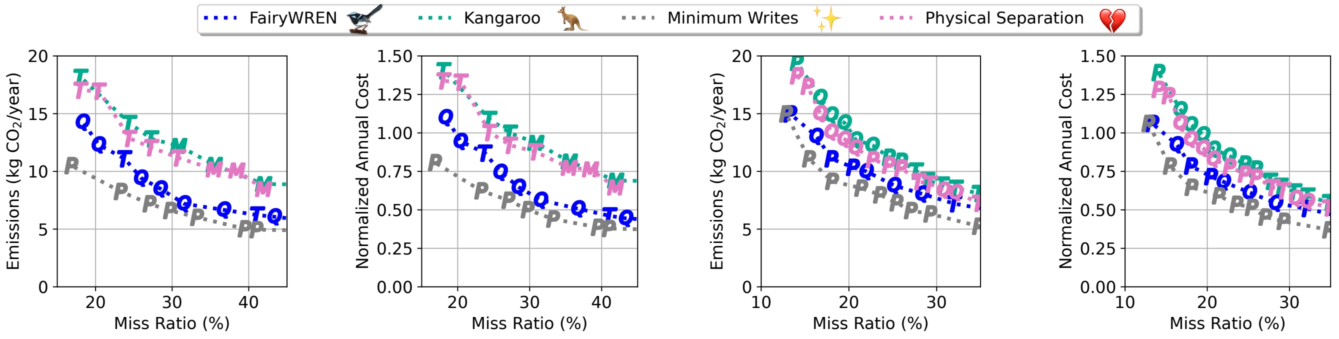


Fig. 11: Yearly carbon emissions for 4 caching systems: minimum writes (✨) with a write amplification of 1 with no additional DRAM, FairyWren (🐦), Kangaroo (🦘), and a Flashield-like log-structured cache (🛡). Our results include the embodied and operational (hatched) emissions from CPU, DRAM, and flash.

Takeaway 0: *Sustainable flash caches must use much less DRAM than log-structured cache designs.*

Although we optimistically assumed that Flashield incurs no write amplification, Flashield's overall carbon emissions are 1.7× higher than Kangaroo's. These emissions are due to its high DRAM overhead, despite several optimizations in Flashield designed to save DRAM. High DRAM overhead is unfortunately inherent in the design of a log-structured cache.

Kangaroo reduces DRAM overhead through its hierarchical design. Unfortunately, Kangaroo also incurs a far higher write rate than a log-structured cache. Kangaroo accounts for its



(a) Twitter Carbon Emissions *(b) Twitter Cost* *(c) Meta Carbon Emissions* *(d) Meta Cost*

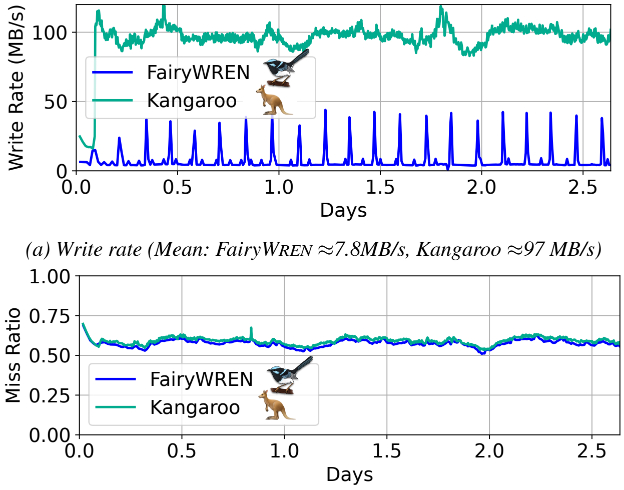
Fig. 13: The emissions and cost over six years for Kangaroo (🦘), FairyWREN (🐦), Min. Writes (✨), and Physical Sep. (💔).

increased writes by overprovisioning flash capacity, increasing embodied carbon emissions. While Kangaroo is far more sustainable than Flashield, it leaves room for improvement.

FairyWREN maintains Kangaroo's low memory overhead while greatly reducing the flash write rate. Consequently, FairyWREN reduces overall carbon emissions by 21.2% compared to Kangaroo. As this improvement comes from reducing flash emissions, we focus on flash emissions for the remainder of the evaluation.

6.3 On-flash experiments

To study how FairyWREN reduces flash writes, we evaluate FairyWREN on real flash drives using the setup in Sec. 6.1.



(a) Write rate (Mean: FairyWREN ≈7.8MB/s, Kangaroo ≈97 MB/s)

(b) Miss ratio (Mean: FairyWREN ≈ 0.575, Kangaroo ≈ 0.594)

Fig. 12: The miss ratio and write rate for Kangaroo and FairyWREN.

Takeaway 1: *FairyWREN greatly reduces flash writes while maintaining a slightly better miss ratio than Kangaroo.*

Fig. 12 plots the flash write rate and miss ratio over time for Kangaroo and FairyWREN. The figure shows small write rate spikes in FairyWREN. This is because FairyWREN performs nest packing at the granularity of an EU, ≈1 GB. Kangaroo's write rate appears smooth as it flushes more frequently, at 256 KB granularity.

The main goal of FairyWREN is to reduce writes, enabling the use of denser flash. In Fig. 12a, FairyWREN reduces writes by 12.5× over Kangaroo, from 97 MB/s to 7.8 MB/s. To achieve this, FairyWREN leverages WREN to combine cache logic and GC and to separate writes of different lifetimes.

However, reducing writes must not increase misses. Fig. 12b shows that, in fact, FairyWREN and Kangaroo have very similar miss ratios: on average, 0.575 for FairyWREN vs 0.594 for Kangaroo. FairyWREN's small advantage comes from reducing idle capacity due to overprovisioning.

We see very similar results for write amplification: a 12.2× reduction, from 23× in Kangaroo to 1.89× in FairyWREN. The slight difference between the write rate and WA comes from FairyWREN's slightly better miss ratio.

Takeaway 2: *FairyWREN outperforms Kangaroo for both throughput and read latency at peak load.*

While the primary performance metric for caches is miss ratio, FairyWREN must provide enough throughput that it does not require more servers — and thus more carbon emissions — to handle the same load. In our experiments, FairyWREN's throughput is 104 KOps/s whereas Kangaroo's is 40.5 KOps/s. FairyWREN's significant throughput increase is mostly due to lower write rate, but also due to better engineering that moved work off the critical path for lookups and inserts.

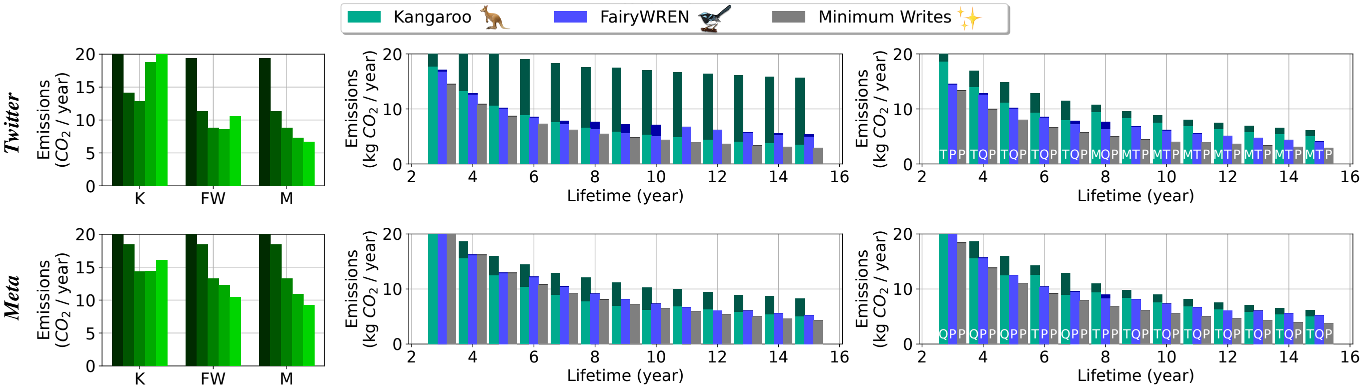
Similarly, we find that FairyWREN's and Kangaroo's 99th-percentile latencies are 170 µs and 1,370 µs, respectively. But note that, in practice, the overall tail latency is set by the backing store, not the flash cache.

6.4 FairyWREN reduces carbon emissions

We now evaluate carbon emissions and cost via simulation, comparing FairyWREN (🐦), Kangaroo (🦘), Minimum Writes (✨), and Physical Separation (💔). Physical Separation represents Kangaroo on WREN, where each cache component (e.g., LOC, KLog, KSet) is placed in its own EU to separate traffic and thereby allow LOC and KLog to have WA of 1×.

Takeaway 3: *FairyWREN's reduced writes translate into reduced carbon emissions and reduced cost across miss ratios.*

Fig. 13 plots emissions and cost for a 6-year lifetime vs. miss ratio over a wide range of cache configurations. Each point is labeled with the flash density used (e.g., TLC).



(a) Impact of density (6 years)

(b) Impact of lifetime (on QLC)

(c) Impact of lifetime (on best flash density)

Fig. 14: The carbon emissions to achieve a 30% miss ratio on Twitter trace or 20% miss ratio on Meta trace for (a) 6 years on SLC (darkest) to PLC (lightest), (b) different lifetimes with QLC flash, and (c) different lifetimes with any flash density. For (b) and (c), the darker part of each bar represents emissions due to overprovisioning.

For the Twitter traces (Fig. 13a, Fig. 13b), Kangaroo is limited to either MLC or TLC due to its high write rate, and likewise for Physical Separation because it does not reduce writes by much (Sec. 6.5). Meanwhile, FairyWREN leverages its low WA to use mostly QLC across miss ratios, giving it large carbon and cost reductions vs. Kangaroo. However, FairyWREN still has too many writes to use PLC. While the gap between Minimum Writes and FairyWREN grows at low miss ratios, there is only a 10.1% difference in their emissions at 20% miss ratio and a 7.7% difference in cost.

The Meta traces (Fig. 13c, Fig. 13d) are less write-intensive. However, even here we see that FairyWREN reduces cache emissions and cost compared to both Kangaroo and Physical Separation. In this case, FairyWREN is able to lower the write rate sufficiently to use QLC and PLC. As a result, FairyWREN performs close to Minimum Writes, even at low miss ratios.

Takeaway 4: *FairyWREN benefits from using denser flash when Kangaroo cannot.*

Flash devices are becoming denser over time (Sec. 2). Fig. 14a shows the carbon-optimal cache configurations over a 6-year lifetime at a target miss ratio of 30% for Twitter and 20% for Meta, varying flash density from SLC (left) to PLC (right). Kangaroo performs best when using TLC on the Twitter trace and QLC on the Meta trace. Using PLC increases Kangaroo's emissions due to the excessive overprovisioning needed to compensate for PLC's lower write endurance. FairyWREN's lower write rate enables it to use QLC for Twitter and PLC for Meta, reducing emissions and cost. Since Twitter's trace is more write-intensive, using PLC increases carbon emissions by 24% due to overprovisioning.

For Minimum Writes on Twitter, emissions decrease by 17% going from TLC to QLC and by 8% from QLC to PLC. On Meta, emissions reduce by 18% and 15%. While these numbers show that denser flash reduces emissions, they suggest diminishing returns even for an optimal cache.

Takeaway 5: *FairyWREN's low WA allows it to avoid massive overprovisioning on dense flash as lifetime is increased.*

To explore the trend of increasing device lifetime (Sec. 2), Fig. 14b considers the emissions for caches on QLC devices, showing emissions from overprovisioning in a darker shade.

For a 6-year lifetime, Kangaroo requires 2.2× the emissions of FairyWREN on Twitter and 1.17× on Meta. At 12 years, the gap increases to 2.6× and 1.54×. Due to the DLWA in LBAD devices, Kangaroo's emissions are lowest when it has some amount of overprovisioning. FairyWREN does not need this overprovisioning due to its lower WA.

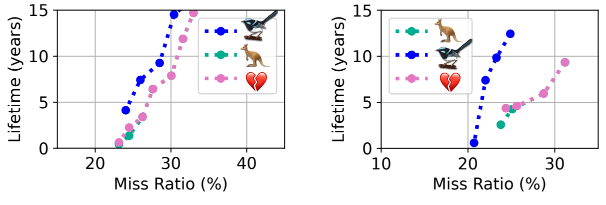
Takeaway 6: *Increasing flash density does not necessarily improve sustainability, as lifetime matters more than density.*

To minimize emissions, we need to optimize both lifetime and flash density. Fig. 14c shows each system's emissions for all lifetimes, with the best density displayed on each bar. Kangaroo usually prefers MLC and TLC because, to provide enough write endurance, QLC and PLC require too much overprovisioning. FairyWREN has fewer emissions than Kangaroo at all lifetimes and stays within 30% of Minimum Writes.

The best flash density decreases for longer lifetimes. FairyWREN prefers PLC on Twitter over 3 years, but TLC over 9 years. At these long lifetimes, the reduced write endurance of denser flash outweighs its sustainability benefits, and extending lifetime is more important than using denser flash.

Takeaway 7: *For a given flash device, FairyWREN extends lifetime by at least a couple of years.*

So far, we have evaluated emissions when deploying the optimal drive for a given lifetime and flash density. However, flash deployments are often constrained to specific devices with a pre-determined capacity and density. In these situations, emissions reductions come from extending lifetime. Fig. 15 evaluates device lifetime for a 3.6 TB drive at different miss ratios. Compared to Kangaroo, FairyWREN is able to extend the device's lifetime by at least 2 years and by over 5 years on the Meta trace. By contrast, Physical Separation barely improves lifetime vs. Kangaroo.



(a) Twitter *(b) Meta*

Fig. 15: The lifetimes for a 3.6 TB cache for Kangaroo (🦘), FairyWren (🐦), and Physical Separation (💔).

6.5 Where are benefits coming from?

We next explore how FairyWren's optimizations contribute to its write rate reduction. Fig. 16 shows the write rate on the Twitter trace starting with Kangaroo on LBAD (Log + Sets). We then add the optimizations of FairyWren incrementally. First, we port Kangaroo naively to WREN (+WREN), then we physically separate the large and small objects into different erase units (+Physical Sep.). Then we add nest packing (+Nest Packing), and, finally, hot-cold object separation (+Hot-Cold) to realize FairyWren. We first present the write rates for the different systems across different capacities and miss ratios, showing the emissions-optimal flash density for one capacity. We then show how the lifetimes of each design would vary if deployed on a QLC drive.

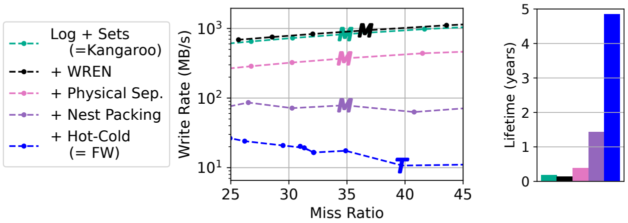


Fig. 16: Write rate (log-scale) and lifetime breakdown on the Twitter trace, incrementally adding optimizations to go from Kangaroo to FairyWren.

Takeaway 8: *Caches on optimal LBAD devices cannot achieve the same write rate as FairyWren.*

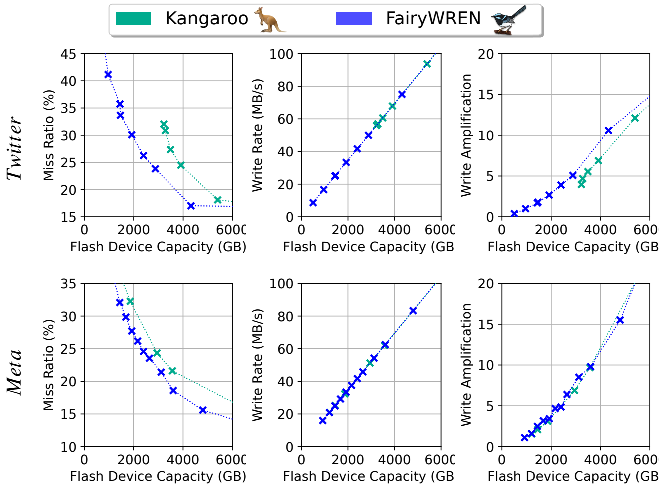
Three of the lines in Fig. 16 are achievable with LBAD devices: Log + Sets, +WREN, and +Physical Sep. Log + Sets represents the current Kangroo implementation on LBAD. +WREN is a naive port of Kangaroo to WREN devices that redirects all cache writes to a single log-structured store using FIFO garbage collection. This naïve port does not attempt any separation of objects by expected lifetime, and we assume it has the same WA as Kangaroo. However, current LBAD devices do try to separate objects belonging to different, concurrent streams, so one would expect an LBAD device to perform, in practice, somewhere between +WREN and +Physical Sep. But even in the best case, Physical Sep. still incurs far too many writes, limiting the lifetime of a QLC device to less than half a year.

Takeaway 9: *Both nest packing and hot-cold object separation are essential to FairyWren's write reduction.*

The other two systems we compare in this breakdown are +Nest packing and +Hot-Cold (i.e., FairyWren with all optimizations). Nest packing reduces writes by at least 3.7× and hot-cold object separation reduces writes by another 3.4×. We also observe that, while hot-cold separation can increase miss ratios, the reduction in write rate outweighs this increase, leading to a 33× increase in QLC lifetime over the Kangaroo baseline and a 13× increase over +Physical Sep.

6.6 Operating on a fixed flash device

We now compare Kangaroo and FairyWren with respect to miss ratio given a fixed flash capacity. We enforce the same constraints of a 6-year flash lifetime, TLC flash density, and 32 GB of DRAM for both systems. Unlike prior figures where we minimize emissions, FairyWren cannot not gain an advantage for using denser flash, and Kangaroo cannot increase write endurance by using less-dense flash. We show that FairyWren under the same capacity constraints, and thus write rate constraints, improves miss ratio over Kangaroo through its reduction in writes allowing it to more effectively use the capacity.



(a) Capacity *(b) Write Rate* *(c) WA*

Fig. 17: Pareto curve of cache miss ratio at different flash device sizes and the corresponding write rate and write amplification of these points. The DRAM capacity is limited to 32 GB, the desired lifetime is 6 years, and the caches use TLC flash.

Takeaway 10: *FairyWren achieves the same miss ratio at lower flash capacities than Kangaroo.*

Fig. 17 shows the effects of changing the flash capacity on miss ratio for both traces. For each flash capacity, we also plot the write rate and WA of both systems. We find that FairyWren needs less flash capacity than Kangaroo to achieve a given miss ratio. FairyWren also requires less overprovisioning due to its lower write rate. This trend is more prominent in the Twitter trace than the Meta trace, which is less write-intensive. For the Twitter trace, Kangaroo's use of TLC prevents it from achieving higher miss ratios. Kangaroo's higher write rate requires much more overprovisioning, increasing the overall flash capacity needed to survive 6 years above 3.6 TB.

We also see that flash capacity sets the write budget for the flash device, defining the write rate that the system can tolerate for a desired lifetime. One might expect a similar relationship for write amplification. However, the systems have different miss ratios, causing Kangaroo to need to have a lower WA through massive overprovisioning.

Takeaway 11: *FairyWREN maintains its advantage under a DRAM constraint.*

We investigated how DRAM restrictions affect Kangaroo and FairyWREN when both caches use 3.6 TB of TLC flash for a 6-year lifetime. FairyWREN maintains a constant miss ratio advantage over Kangaroo from 16 GB to 64 GB of DRAM for both traces. FairyWREN's miss ratio only begins to increase when DRAM falls to 8 GB on the Twitter trace. However, Kangaroo cannot handle the Twitter workload for 6 years with only 8 GB of DRAM. Hence, FairyWREN always outperforms Kangaroo in these experiments.

7 Related Work

This section discusses additional related work with similar techniques and goals to FairyWREN.

Hot-cold objects and deathtime. In caching, hot objects are the most popular objects. Caches use eviction policies to retain popular objects [15, 45, 47, 83]. FairyWREN adapts Kangaroo's RRIP-based eviction policy [45, 67].

Popularity is different than *deathtime*, the time when an object will be deleted [41]. To minimize GC, many storage systems will physically separate objects by their deathtime [26, 28, 41, 54, 76, 94]. Grouping objects with similar deathtimes reduces WA. Hence, accurately predicting deathtimes is vital for minimizing write amplification within LBAD. Recent work uses ML to make these predictions [26, 94]. Unfortunately, ML solutions require additional hardware that can increase emissions and cost.

Caches have more control over deathtimes than storage systems. Deathtimes are set by the eviction policy, and thus determining an object's deathtime is more straightforward. For instance, in caches that evict based on TTLs, the TTLs can be used to group objects [93]. FairyWREN leverages its eviction policy's popularity rankings and the WREN interface to physically group objects by deathtime.

Eviction and garbage collection. Prior flash caches have attempted to reduce in-device garbage collection. Many log-structured caches [27, 35, 56, 61] group objects into large segments and trim these segments during eviction to minimize garbage collection. These systems attempt to evict segments before device-level GC rewrites them. Unfortunately, this does not ensure GC is prevented on LBAD devices, so some work has proposed leveraging newer interfaces to guarantee alignment. DidaCache [78], for example, uses an Open-Channel SSD [20] to guarantee its segments will align with erase units. Other proposals to use more expressive interfaces re-implement LBAD-like GC on top of a ZNS SSD [29], prohibiting optimizations like FairyWREN's nest packing. All of these log-structured approaches suffer from high DRAM overheads and cannot evict individual objects without additional writes.

Grouping by object size. FairyWREN separates objects into two object size classes, large and small, similar to Kangaroo [68] and CacheLib [16]. This grouping is used to minimize memory overhead. Allocating memory using size-based slab classes is often used to reduce fragmentation [25, 43, 77, 78, 93]. Introducing additional object size classes in FairyWREN would result in additional flash accesses, since FairyWREN does not index the size classes to save memory. Instead, FairyWREN reduces fragmentation by grouping objects into either large segments in the LOC or sets in FwSets. These segments and sets are periodically rearranged to prevent fragmentation.

8 Conclusion

FairyWREN reduces flash's carbon emissions and cost by integrating flash management with cache policies. Doing so requires redesigning the cache to transition from old LBAD flash interfaces to a WREN interface. Experiments show that FairyWREN decreases flash writes by 12.5× vs. the state-of-the-art, allowing longer flash lifetimes that reduce carbon emissions by 33% and cost by 35%.

9 Acknowledgements

Sara McAllister is supported by a NDSEG Fellowship. We thank the members and companies of the PDL Consortium (Amazon, Google, Hitachi, Honda, IBM Research, Intel, Jane Street, Meta, Microsoft Research, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital) for their interest, insights, feedback, and support. We thank our shepherd, Gala Yadgar, and our anonymous reviewers for their helpful comments and suggestions. We also thank Western Digital for providing resources and technical expertise; we especially thank Matias Bjørling, Ajay Joshi, and Hans Holmberg. We also specifically thank Javier Gonzalez and Mike Allison, at Samsung, and Ross Stenfort, at Meta, for providing their technical expertise on FDP. We thank the PDL staff, particularly Jason Bowles, for their support.

References

[1] Amazon sustainability. https://sustainability.aboutamazon.com/climate-solutions.

[2] Climate change is humanity's next big moonshot. https://blog.google/outreach-initiatives/sustainability/dear-earth/.

[3] Fatcache. https://github.com/twitter/fatcache.

[4] Flash prices. https://jcmit.net/flashprice.htm.

[5] Leveldb. https://github.com/google/leveldb.

[6] Memory prices. https://jcmit.net/memoryprice.htm.

[7] Rocksdb. http://rocksdb.org.

[8] Ssd over-provisioning and its benefits. https://www.seagate.com/blog/ssd-over-provisioning-benefits-master-ti/.

[9] Wd and tosh talk up penta-level cell flash. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/ 5/17/22.

[10] Is there a limited warranty for samsung ssds? https://semiconductor.samsung.com/us/consumer-storage/support/faqs/05/, 2023.

[11] Our path to net zero. https://sustainability.fb.com/wp-content/uploads/2023/07/Meta-2023-Path-to-Net-Zero.pdf, 2023.

[12] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon Explorer: A Holistic Framework for Designing Carbon Aware Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 118–132, Vancouver BC Canada, January 2023. ACM.

[13] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[14] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: Exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 79–85, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing hit density. In *USENIX NSDI*, 2018.

[16] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory G. Ganger. The CacheLib caching engine: Design and experiences at scale. In *USENIX OSDI*, 2020.

[17] Daniel S. Berger, Fiodar Kazhamiaka, Esha Choukse, Inigo Goiri, Celine Irvene, Pulkit A. Misra, Alok Kumbhare, Rodrigo Fonseca, and Ricardo Bianchini. Research avenues towards net-zero cloud platforms. *Workshop on NetZero Carbon Computing*, 2023.

[18] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, 2017.

[19] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021.

[20] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *USENIX Conference on File and Storage Technologies*, pages 359–374. USENIX-The Advanced Computing Systems Association, 2017.

[21] Netflix Technology Blog. Application data caching using ssds. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef, 2016.

[22] Netflix Technology Blog. Evolution of application data caching : From ram to ssd. https://bit.ly/3rN73CI, 2018.

[23] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *USENIX FAST*, 2010.

[24] Erik Brunvand, Donald Kline, and Alex K. Jones. Dark silicon considered harmful: A case for truly green computing. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018.

[25] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 353–362, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, Haifa Israel, June 2021. ACM.

[27] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: an embedded concurrent key-value store for state management. *VLDB*, 2018.

[28] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[29] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new LSM-style garbage collection scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[30] Amanda Peterson Corio. Five years of 100carbon-free future. `https://cloud.google.com/blog/topics/sustainability/5-years-of-100-percent-renewable-energy`.

[31] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, 2017.

[32] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, 2018.

[33] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *ACM SIGMOD*, 2011.

[34] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–10, 2012.

[35] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*, 2019.

[36] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. Mcdipper: A key-value cache for flash storage. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/`.

[37] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. What your dram power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018.

[38] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. ACT: designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, 2022.

[39] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021.

[40] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[41] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *ACM EuroSys*, 2017.

[42] Amy Hood, July 2022.

[43] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.

[44] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, February 2017. USENIX Association.

[45] Aamer Jaleel, Kevin Theobald, Simon Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.

[46] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[47] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[48] Nicola Jones et al. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–166, 2018.

[49] Lucas Joppa. Made to measure: Sustainability commitment progress and updates. `https://blogs.microsoft.com/blog/2021/07/14/made-to-measure-sustainability-commitment-progress-and-updates/`.

[50] Ajay Joshi. Cachelib on zns. `https://github.com/ajaysjoshi/CacheLib-zns`, 2022.

[51] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[52] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for Multi-Streamed SSDs using program contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 295–308, Boston, MA, February 2019. USENIX Association.

[53] Bran Knowles. Acm techbrief: Computing and climate change, 2021.

[54] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *USENIX FAST*, 2015.

[55] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[56] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[57] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage*, 13(3):24, 2017.

[58] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Xin Li, Greg Thompson, and Joseph Beer. How amazon achieves near-real-time renewable energy plant monitoring to optimize performance using aws. `https://aws.amazon.com/blogs/industries/amazon-achieves-near-real-time-renewable-energy-plant-monitoring-to-optimize-performance-using-aws/`.

[60] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *ACM SOSP*, 2011.

[61] Jian Liu, Kefei Wang, and Feng Chen. Tscache: An efficient flash-based caching scheme for time-series data workloads. 2021.

[62] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *USENIX FAST*, 2016.

[63] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3d nand flash memory lifetime by tolerating early retention loss and process variation. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018.

[64] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *2nd Workshop on Sustainable Computer Systems (HotCarbon23)*. ACM, July 2023.

[65] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, et al. Hyrax:{Fail-in-Place} server operation in cloud platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 287–304, 2023.

[66] Bill Martin, Yoni Shternhell, Mike James, Yeong-Jae Woo, Hyunmo Kang, Anu Murthy, Erich Haratsch, Kwok Kong, Andres Baez, Santosh Kumar, and et al. Nvm express technical proposal 4146 flexible data placement, Nov 2022.

[67] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *ACM SOSP*, 2021.

[68] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. *ACM Transactions on Storage*, 2022.

[69] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 461–477, Boston, MA, July 2023. USENIX Association.

[70] Christian Monzio Compagnoni, Akira Goda, Alessandro S. Spinelli, Peter Feeley, Andrea L. Lacaita, and Angelo Visconti. Reviewing the evolution of the nand flash technology. *Proceedings of the IEEE*, 105(9):1609–1633, 2017.

[71] Melanie Nakagawa. On the road to 2030: Our 2022 environmental sustainability report. `https://blogs.microsoft.com/on-the-issues/2023/05/10/2022-environmental-sustainability-report/`, 2022.

[72] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *ASPLOS*, 2014.

[73] Francisco Pires. Solidigm introduces industry-first plc nand for higher storage densities. `https://www.tomshardware.com/news/solidigm-plc-nand-ssd`, 2022.

[74] Martin Raab and Angelika Steger. Balls into Bins: A Simple and Tight Analysis. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Michael Luby, JosÃ© D. P. Rolim, and Maria Serna, editors, *Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.

[75] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*, 2017.

[76] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.

[77] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association.

[78] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.

[79] Shigeru Shiratake. Scaling and performance challenges of future dram. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–3, 2020.

[80] Billy Tallis. 2021 nand flash updates from isscc: The leaning towers of tlc and qlc. `https://www.anandtech.com/show/16491/flash-memory-at-isscc-2021`.

[81] Billy Tallis. Micron 3d nand status update. `https://www.anandtech.com/show/10028/micron-3d-nand-status-update`.

[82] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *USENIX OSDI.*, 2020.

[83] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, 2015.

[84] Swamit Tannu and Prashant J Nair. The Dirty Secret of SSDs: Embodied Carbon. In *HotCarbon*, 2022.

[85] Amanda Tomlinson and George Porter. Something Old, Something New: Extending the Life of CPUs in Datacenters. In *HotCarbon*, 2022.

[86] Ted Tso. Aligning filesystems to an ssd's erase block size. `https://tytso.livejournal.com/2009/02/20/`.

[87] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[88] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, Xingsheng Zhao, Weijun Li, and Song Jiang. OC-Cache: An Open-channel SSD Based Cache for Multi-Tenant Systems. In *2018 IEEE 37th International Performance*

Computing and Communications Conference (IPCCC), pages 1–6, Orlando, FL, USA, November 2018. IEEE.

[89] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, Santa Clara, CA, February 2022. USENIX Association.

[90] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *USENIX ATC*, 2015.

[91] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), oct 2017.

[92] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[93] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *USENIX NSDI*, 2021.

[94] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[95] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[96] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 1, USA, 2012. USENIX Association.

Appendix A Modeling of DLWA Under Random Writes

Our goal is to model the effect of EU size on DLWA. Specifically, we want to analyze the performance of the FIFO+ GC policy, which selects EUs for garbage collection in FIFO order and skips EUs which contain only valid data. Several prior papers [34,46,87] have noted that DLWA can be approximated using W Lambert functions, but this prior work tend to focus on the level of device overprovisioning rather than on the EU size. We use an approach similar to that of [46] to model the relationship between EU size and DLWA under FIFO+.

We define X to be the random variable representing the number of invalid pages an EU that is targeted for garbage collection. Because FIFO+ will erase an EU only if it contains invalid pages, our goal is to approximate $\mathbb{E}[X|X>0]$. This tells us the number of new pages that can be written every time GC is performed. Hence, if we let b be the number of pages in an EU, we can compute the DLWA as

$$\text{DLWA} = \frac{b}{\mathbb{E}[X|X>0]}. \tag{1}$$

Our approximation makes two simplifying assumptions.

First, we assume that each of the b pages in the target EU is invalid independently with probability p. This is reasonable when writes are random and the total number of pages in the device is large. This assumption implies that $X \sim \text{Binomial}(b,p)$. To approximate the expectation of X, we must approximate p.

Second, we assume that an EU is targeted for GC every k writes, where k is a constant. Specifically, we define t to be the total number of EUs in the device and assume $k = t\mathbb{E}[X]$. This is a reasonable approximation because k is the expected number of writes that occur between GC operations on a given EU and the total number of EUs, t is large. A particular page will be invalid if at least one of the k writes targets the page. Hence, the probability p that a page is invalid is

$$p = 1 - \left(1 - \frac{1}{ub}\right)^k$$

where u is the number of EUs available to store valid user data. Note that u is typically smaller than t, and $\frac{t}{u}$ represents the amount of overprovisioning in the device.

Combining these assumptions yields

$$\mathbb{E}[X] \approx b \cdot p \approx b\left(1 - \left(1 - \frac{1}{ub}\right)^k\right) \tag{2}$$

$$\approx b\left(1 - \left(1 - \frac{1}{ub}\right)^{t\mathbb{E}[X]}\right). \tag{3}$$

We can rewrite (3) using the W Lambert function to get the following approximation for $\mathbb{E}[X]$:

$$\mathbb{E}[X] = b - \frac{W\left(bt\left(1-\frac{1}{ub}\right)^{tb}\ln\left(1-\frac{1}{ub}\right)\right)}{t\ln\left(1-\frac{1}{ub}\right)}.$$

To compute $\mathbb{E}[X \mid X>0]$, we note that

$$\mathbb{E}[X \mid X>0] = \sum_{i=1}^{b} i \cdot \frac{P(X=i)}{P(X>0)} = \frac{1}{P(X>0)} \sum_{i=0}^{b} i \cdot P(X=i)$$

and thus

$$\mathbb{E}[X \mid X>0] = \frac{\mathbb{E}[X]}{P(X>0)} = \frac{\mathbb{E}[X]}{1-(1-p)^k}.$$

Hence, we now have an approximation that allows us to write DLWA as defined in (1) in terms of the device parameters t, u, and b. This approximation is validated against simulation in Figure 6.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Massively Parallel Multi-Versioned Transaction Processing

Shujian Qian
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Multi-version concurrency control can avoid most read-write conflicts in OLTP workloads. However, multi-versioned systems often have higher complexity and overheads compared to single-versioned systems due to the need for allocating, searching and garbage collecting versions. Consequently, single-versioned systems can often dramatically outperform multi-versioned systems.

We introduce Epic, the first multi-versioned GPU-based deterministic OLTP database. Epic utilizes a batched execution scheme, performing concurrency control initialization for a batch of transactions before executing the transactions deterministically. By leveraging the predetermined ordering of transactions, Epic eliminates version search entirely and significantly reduces version allocation and garbage collection overheads. Our approach utilizes the computational power of the GPU architecture to accelerate Epic's concurrency control initialization and efficiently parallelize batched transaction execution, while ensuring low latency. Our evaluation demonstrates that Epic achieves comparable performance under low contention and consistently higher performance under medium to high contention versus state-of-the-art single and multi-versioned systems.

1 Introduction

There has been a growing need for high-throughput online transaction processing (OLTP) systems capable of executing tens of thousands of transactions per second. In-memory database systems, specifically designed for workloads with datasets that fit entirely in DRAM memory and provide durability and high availability via logging and replication, have been developed to address this demand. Although these systems offer considerable performance advantages over traditional disk-based systems, they suffer under contention, leading to low performance and limited scalability across cores.

Multi-versioning offers a promising solution for contended and read-heavy workloads. Multi-version systems maintain recent past versions of each record, enabling concurrent reads and writes to the same record; reads do not block writes because writes can safely create new versions while reads are accessing the old versions. Consequently, transactions can be serialized in ways unattainable in single-version designs, thereby enabling greater parallelism. Previous work has shown that multi-version systems can outperform single-version systems under high contention [17].

However, current multi-version designs have several drawbacks, including increased overheads during transaction processing, data storage, allocation and garbage collection. These designs store record versions in linked lists, introducing an additional layer of indirection and necessitating list traversal to locate the appropriate version. Accessing the versions results in a larger working set, leading to higher cache miss rates and performance degradation. Multiple versions also lead to higher memory requirements. To reduce the memory footprint, versions are frequently garbage collected, which incurs additional overheads. As a result, a previous study that compared carefully tuned, state-of-the-art multi-version and single-version systems demonstrated that under low contention, a multi-version system has roughly *half* the throughput of single-version systems [14].

Current multi-version designs allocate versions dynamically because transactions may write and thus create versions at any time. Thus, versions are stored in linked lists, reads require searching for versions, and garbage collecting versions has poor locality and requires expensive synchronization.

Our key insight is that deterministic databases employing transaction batching and known transaction read-write sets can avoid most of these multi-versioning costs, thus enabling good performance for all workloads. The transaction batching and known read-write sets requirements are commonly met by most deterministic databases [11, 12, 18, 19, 26, 28, 31, 35].

We introduce Epic, the first multi-versioned, GPU-based deterministic transaction-processing database. Epic batches transactions into epochs and establishes a serial ordering of transactions within a batch before transaction execution, similar to other deterministic databases.

Transaction batching enables splitting an epoch into an *initialization phase* during which concurrency control operations are initialized using the read-write sets, followed by an *execution phase* during which transactions are executed concurrently and synchronized to ensure the deterministic ordering. During the initialization phase, Epic allocates versions based on the write set. These allocation operations are performed efficiently because they do not interfere with transaction execution. In addition, Epic calculates the version location of each read/write operation based on the ordering of transactions and the known read-write sets. This approach enables transactions to access versions directly during the execution phase, without requiring any version search.

Epic's epoch-based design enables efficient garbage collection as well. Since transactions in the next epoch are serialized after all transactions in the current epoch, only the final write to a record is visible to the transactions in the next epoch. Thus all versions except the last one become obsolete when an epoch ends. Epic stores all intermediate record versions separately from the last version and reclaims them efficiently at the end of an epoch.

The challenge is that Epic's initialization phase is expensive, requiring both significant computation and memory bandwidth. Fortunately, Epic's initialization phase is highly parallelizable. With the rapid commoditization of general-purpose GPU computing, Epic harnesses the thread parallelism offered by modern GPU architectures to significantly accelerate the initialization phase.

Modern GPUs are well suited for Epic's execution phase as well because they offer high-bandwidth memory for memory-bound workloads. In addition, they perform zero-overhead context switching between thread contexts, which allows hiding memory access latency. These advantages help to counter the increased memory footprint and lower cache utilization commonly associated with multi-version systems. Consequently, Epic achieves high throughput and ensures low transaction latency even with its epoch-based execution scheme.

While GPU transaction execution performs well, it is limited by datasets that fit in GPU memory. Thus, Epic also supports larger datasets with a CPU execution model in which the initialization phase runs on the GPU while the execution phase runs on the CPU.

To demonstrate the effectiveness of Epic's design, we conduct extensive evaluation using the TPC-C and YCSB benchmarks and show that Epic significantly outperforms recent single- and multi-version systems on most workloads.

2 Background

This work builds on a rich body of research on multi-version concurrency control, deterministic databases, and GPU-accelerated computation, as discussed below.

2.1 Multi-versioned Concurrency Control

Multi-version concurrency control (MVCC) has a long history [29, 30], with early work evaluating its performance [8], ensuring snapshot isolation [5], providing serializable snapshot isolation [7], using dynamic timestamp assignment [20] and enabling efficient indexing [32], for disk-based databases.

With the advent of machines equipped with high core counts and terabytes of DRAM memory, much work has focused on in-memory database designs, and several MVCC schemes optimized for them have been proposed [15, 16, 22]. MVCC schemes are popular because they provide robust performance under a wide range of workloads. As a result, many commercial in-memory databases implement MVCC [10, 24, 25, 34].

Wu et al. conduct a detailed study of the costs associated with concurrency control, version storage, garbage collection, and index management in various in-memory MVCC schemes [37]. Cicada [17] outperforms previous MVCC schemes with several optimizations, including optimistic multi-versioning, contention regulation, version inlining, and rapid garbage collection. However, a study comparing state-of-the art multi-version and single-version systems showed that while MVCC outperforms OCC under high contention, its throughput is significantly lower under low contention [14]. Epic aims to minimize multi-versioning costs associated with version storage, lookup and garbage collection.

2.2 Deterministic Database Systems

Deterministic databases have gained increasing attention in recent years, driven by the need for efficient replication and improved scalability for distributed transactions [35]. These systems execute transactions deterministically by ensuring that the serial ordering of operations remains consistent across different runs. Determinism enables efficient replication [27, 31, 33] and live migration [18, 19] since all replicas execute transactions independently without coordination. Furthermore, deterministic systems reduce the need for two-phase commit, helping scale the performance of distributed transactions [35]. They can also effectively handle skewed and contended accesses, e.g., orders for popular items [28].

Deterministic systems typically batch transactions into epochs to perform deterministic concurrency control before execution [11, 12, 28, 35]. Thus these system require the read and write sets of transactions to be known before execution. When they are not fully known, they can be determined using reconnaissance queries [35]. Calvin [35] and PWV [12] are single versioned, while Bohm [11] and Caracal [28] utilize MVCC. Calvin uses a centralized lock manager, while PWV employs a more-scalable per-core dependency analysis for concurrency control. Bohm and Caracal allocate versions scalably during the concurrency control initialization phase, but Bohm performs partitioned initialization, while Caracal performs shared memory initialization. Bohm partitions the records in a table across cores. During the initialization phase, all partitions analyze each transaction's write set and insert placeholder versions in a linked list for the records they own. During execution, a read operation traverses the list to find the correct version based on its total order ID. Then, it synchronizes with a write operation that fills the corresponding placeholder version. Caracal uses shared-memory initialization, which enables better handling of skewed workloads. It scales version allocation for contended records by batching the allocations. It stores versions as sorted arrays and uses binary search to reduce version lookup costs during execution. Epic performs shared-memory initialization similar to Caracal. However, Epic avoids any version lookup costs and minimizes version storage and garbage collection overheads.

2.3 GPU Accelerated OLTP Databases

General-Purpose computing on Graphics Processing Units (GPGPU) has become popular with the rapid commoditization of GPUs, the advent of user-friendly programming models and frameworks like CUDA and OpenCL, and the growing demand for high-performance computing on large datasets. Modern GPUs contain an array of streaming multiprocessors (SMs), each of which contains many CUDA cores or stream processors, allowing execution of thousands of active threads concurrently. GPUs uses the Single Program, Multiple Data (SPMD) parallel programming model in which multiple threads execute the same program on different data elements.

GPU-based databases are an active area of research, but most work has focused on accelerating Online Analytical Processing (OLAP) workloads since typical OLAP operators, such as join and sort, are a good fit for parallelization using the GPU's SPMD execution model.

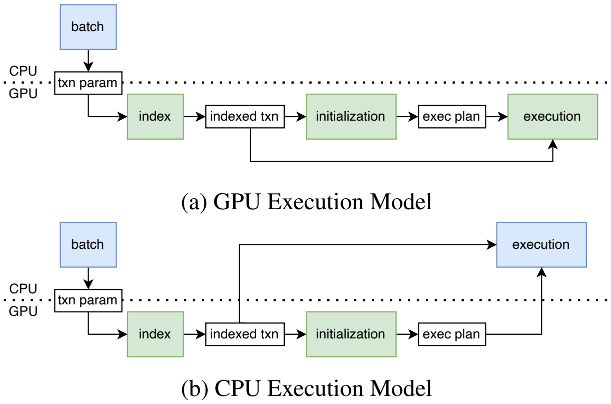
GPU-based transaction processing is relatively unexplored because transactional workloads comprise short-lived transactions with random accesses, and atomicity and isolation require significant synchronization. These requirements makes it hard to exploit the parallelism available in GPUs.

Previously, two GPU-based transaction processing systems, GPUTx [13] and GaccO [6], have been proposed. Similar to Epic, both batch transactions and use epoch-based concurrency control initialization and execution. GPUTx, an early attempt at executing OLTP workloads on GPUs, uses dependency tracking to group transactions into sets; transactions within each set are conflict-free and can execute without synchronization. However, we found that their efficient dependency tracking algorithm, K-Set, does not ensure that transactions in a set are conflict-free, thereby failing to guarantee correctness. GaccO is a deterministic database that uses single-version, deterministic locking, similar to Calvin. We describe Gacco in detail and compare it with Epic in Section 5.

3 Design

Epic is a GPU-accelerated, in-memory deterministic database that employs a novel multi-versioned concurrency control protocol. Epic assumes that transactions are one-shot and use stored procedures, similar to other high-performance in-memory databases [36].

Figure 1 shows the Epic architecture. Epic batches transactions into epochs and splits each epoch into indexing, initialization and execution phases. The transaction inputs, consisting of read-set and write-set keys and other transaction data, are batched on the CPU and then transferred to the GPU for indexing (shown as "txn param" in Figure 1). During indexing, the keys are used to retrieve and store the corresponding record IDs in a per-transaction data structure (shown as "indexed txn" in Figure 1). These record IDs are used during initialization and used as indices for accessing the record tables during transaction execution. The initialization phase performs multi-versioned concurrency control and generates a per-transaction execution plan, which consists of the locations of the record versions that a transaction then directly accesses during execution.



(a) GPU Execution Model

(b) CPU Execution Model

Figure 1: Epic Architecture

While indexing and initialization are always run on the GPU, Epic can execute transactions on the GPU (Figure 1a) or the CPU (Figure 1b). CPU execution is used to support databases larger than GPU memory. In this case, the GPU serves as an accelerator for indexing and initialization.

Sometimes a transaction's read and write sets are not fully known before the indexing phase. For example the TPC-C order-status transaction requires a secondary index to locate a customer's latest order. For these transactions, Epic runs an optional read-write set identification phase on the GPU before the indexing phase. The transaction inputs to the identification phase only contain the read-write keys that are known at transaction generation time. This phase runs reconnaissance queries [35] that use these partial transaction inputs to identify the remaining read-write keys.

The following sections describe Epic's storage scheme and then Epic's initialization and execution phases.

3.1 Epic storage scheme

Epic's storage scheme separates temporary versions created within an epoch from versions that exist across epochs. All writes to a record within an epoch, except the last one, are only read by other transactions within the epoch. This is because transactions from a later epoch are serialized after all transactions in the current epoch and thus can only read the last version of each record. We call the versions that are read by transactions within an epoch *temporary versions*. The final write to a record within an epoch may be read in later epochs and so this last version is saved across epochs.

Figure 2 shows an example of Epic's storage scheme with transactions T1 to T8 reading and writing Record 1 at Epoch 3. Epic places all temporary versions in a scratchpad area. During an epoch, the write transactions on a record, except

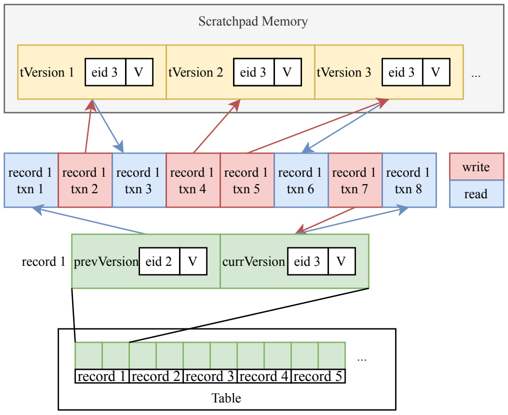


Figure 2: Epic Storage Scheme

the last one, fill these versions, and reads synchronize with the writes to ensure RAW dependencies are satisfied. At the end of an epoch, when all the reads for the temporary versions are done, the scratchpad is reclaimed and used in the next epoch, completely eliminating per-version garbage collection.

The final versions of each record are placed in a dense table area and do not require garbage collection. The last write in an epoch to each record updates the value in the table directly, leading to potential race conditions when transactions in the current epoch need to read data from the previous epoch. Epic addresses this problem by storing two versions for each record in the table: the previous version (*prevVer*) and the current version (*currVer*), as shown in Figure 2. In each epoch, *prevVer* holds the data from the previous epoch (original version). The last write (Txn 7) within an epoch updates *currVer* to avoid overwriting the original version. This write is performed directly on the table, so all temporary versions can be easily collected after an epoch. Reads after the last write to a record (Txn 8) read from *currVer*.

The locations of *prevVer* and *currVer* in each record depend on transaction history, as their positions only change when a record is written during an epoch. Therefore, Epic stores an epoch ID in each version, which helps distinguish the version from previous epochs (*prevVer*) from the version that should be updated in the current epoch (*currVer*). Algorithm 1 is used by transactions to distinguish between *prevVer* and *currVer*. In an epoch, before any write has happened to *currVer*, Epic ensures that the version with a larger epoch ID contains the more up-to-date value and should be used as *prevVer* (Lines 9–12). During the last write, the writer will update the epoch ID of *currVer* to the current epoch's ID (*current_eid*), after which *currVer* will have a larger epoch ID, but it is still distinguishable since its epoch ID matches the current epoch ID (Lines 4–7). The epoch ID is also used for synchronization between reads and writes, as discussed later in Section 3.3.

The record tables and the scratchpad memory are stored in GPU memory for the GPU execution model and in CPU memory for the CPU execution model.

Algorithm 1: Determining the *prevVer* and *currVer*

```
   // Takes the two table versions of a record
1  Function GetTableVersions ($V[2]$):
2      $eid0 \leftarrow atomicRead(V[0].eid)$
3      $eid1 \leftarrow atomicRead(V[1].eid)$
       // current_eid is the current epoch's ID
4      if $eid0 = current\_eid$ then
5          $prevVer \leftarrow V[1]$; $currVer \leftarrow V[0]$
6      else if $eid1 = current\_eid$ then
7          $prevVer \leftarrow V[0]$; $currVer \leftarrow V[1]$
8      else
9          if $eid0 > eid1$ then
10             $prevVer \leftarrow V[0]$; $currVer \leftarrow V[1]$
11         else
12             $prevVer \leftarrow V[1]$; $currVer \leftarrow V[0]$
13     return $\{prevVer, currVer\}$
```

3.2 Multi-Version Initialization

During the initialization phase, Epic uses the ordering of transactions and the knowledge of their read-write sets to allocate versions for all writes performed in the epoch. To avoid the expensive version search required in previous multi-versioned systems, Epic calculates the read-write version locations for each transaction in the epoch before any transactions execute. These operations are parallelizable because they are performed in a phase separate from transaction execution.

As shown in Algorithm 2, Epic employs a parallel GPU-based algorithm to perform concurrency control initialization efficiently. Figure 3 provides an example of this algorithm. The initialization phase starts by collecting all the read and write operations within the epoch (Step 1). Each entry in the *all_ops* operations array contains the *record_id* and the *txn_id* associated with the operation, the operation's index within the transaction (*op_id*), and the operation type (read/write). This operation is parallelizable because the order of operations does not matter for the next step, which sorts the operations array by *record_id* and *txn_id* (Step 2).

Then, Epic counts the number of write operations to each record that occur before and after each operation. Since the operations are already grouped by *record_id*, these operations use parallel prefix and postfix sum by key (Steps 3–4). Next, *GetOpType* in Algorithm 3 calculates the read-write location type for each operation (Step 5). A write operation writes to *currVer* for the last write to the record or else to *tempVer*. A read operation will read from the version written by the previous write as follows: *prevVer* if there is no preceding write preceding, *currVer* if there is no succeeding write, and *tempVer* otherwise.

The number of *tempVer* variables created in an epoch is equal to the number of *tempVer* writes. Thus, Epic places the *tempVer* variables in the scratchpad area in the same order as the *tempVer* write operations in the sorted operations array. To calculate the *tempVer* locations, Epic performs a parallel prefix sum over all operations, counting *tempVer* writes before each operation (Step 6). With this information,

Algorithm 2: Multi-Version Initialization Phase

```
Function Initialize (txns[NUM_TXN]):
    all_ops  // all read-write operations in the epoch,
    // contains tuples: {record_id, txn_id, op_id, read_write}
    // All local variables are arrays of size equal to all_ops size

    // Step 1: submit operations
    parallel foreach txn ∈ txns do
        op_id = 0
        foreach record_id ∈ txn.read_record_ids do
            op_id++
            all_ops.pushback({record_id, txn.id, op_id, Read})
        foreach record_id ∈ txn.write_record_ids do
            op_id++
            all_ops.pushback({record_id, txn.id, op_id, Write})
    // Step 2: sort first by record_id then by txn_id
    sorted_ops = Sort(all_ops, key = {record_id, txn_id})
    // Steps 3-4: count writes before/after each op on same record
    writes_before = PrefixSumByKey(sorted_ops,
                                   key = record_id,
                                   value = Write ? 1 : 0)
    writes_after = PostfixSumByKey(sorted_ops, key = record_id,
                                   value = Write ? 1 : 0)
    // Step 5: get operation type, can be:
    // prevVer read, currVer read/write, tempVer read/write
    op_types = GetOpType(sorted_ops, writes_before,
                         writes_after)
    // Step 6: count tempVer writes before each op in the epoch
    tw_before = PrefixSum(op_types, value=tempVerWrite ? 1 : 0)
    // Step 7: get read/write location for all ops
    rw_loc = GetRWLocation(op_types, tw_before)
    // Step 8: scatter rw_loc back to transactions
    parallel for i = 0 to sorted_ops.size do
        txn_id = sorted_ops[i].txn_id
        op_id = sorted_ops[i].op_id
        txns[txn_id].locations[op_id] = rw_loc[i]
```

GetRWLocation in Algorithm 4 calculates the read-write locations for all operations (Step 7). The ith *tempVer* write updates the ith *tempVer* in the scratchpad area. A read from *tempVer* reads the previous write in the sorted operations array. Finally, the read-write locations are scattered back to each transaction to be used in the execution phase (Step 8).

3.3 Transaction Execution

Epic's execution phase is considerably simpler than the initialization phase. A transaction accesses versions directly using the locations calculated during initialization, eliminating any version lookup during execution. Due to multi-versioning, write-after-read (WAR) and write-after-write (WAW) dependencies do not require explicit coordination. Epic uses the epoch ID associated with each version to synchronize read-after-write (RAW) dependencies between transactions.

Algorithm 5 shows Epic's transaction execution phase. The transactions in an epoch are scheduled in their predetermined serial order as thread resources become available, as explained further in Section 4.4. The *RunTxn* function shows an example of a transaction. The transaction accesses the versions directly using the location information calculated in the initialization phase (lines 11–25 for reads and lines 26–34 for writes). A transaction read waits for a version to be written by an earlier transaction by spinning on the epoch ID of the version until it matches the current epoch ID (lines 21–22). However, reads from *prevVer* do not need any synchronization since this version was updated in a previous epoch. A transaction writes to the data of the version before updating the version's epoch ID (lines 32–34). The GPU weak memory consistency model requires a memory fence between the data write and the epoch ID update to ensure that the data is visible to other threads before the updated version.

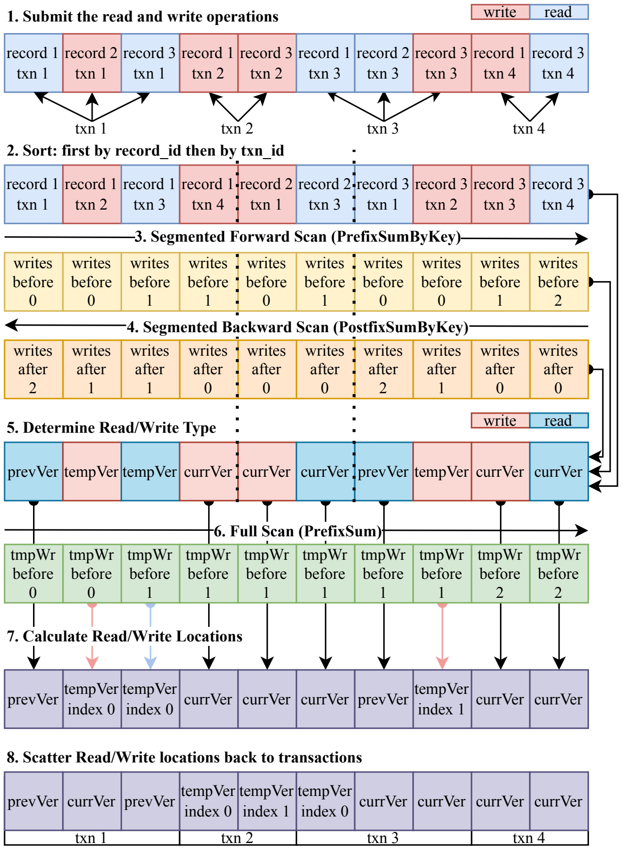


Figure 3: Example of Epic's Initialization Phase

CPU-side Execution Epic can also execute transactions on the CPU, which is particularly useful when the database size exceeds GPU memory capacity. In this case, Epic transfers the output of indexing (read and write record IDs) and the initialization phase (read-write locations) to the CPU, as shown in Figure 1. CPU-side execution utilizes the same synchronization mechanism as GPU execution.

Handling Inserts and Deletes Epic treats record insertions and deletions the same way as updates. Both insert and delete operations are considered write operations, so they create a new version of the record, similar to an update. For each version, Epic uses a valid flag to mark whether it contains

Algorithm 3: Calculate Read/Write Type

```
Function GetOpType (sorted_ops, writes_before, writes_after):
    op_types[sorted_ops.size]  // type of operations
    parallel for $i$ = 0 to sorted_ops.size do
        if sorted_ops[i].read_write == Write then
            if writes_after[i] == 0 then
                op_types[i] = currVerWrite
            else
                op_types[i] = tempVerWrite
        else // read operation
            if writes_before[i] == 0 then
                op_types[i] = prevVerRead
            else if writes_after[i] == 0 then
                op_types[i] = currVerRead
            else
                op_types[i] = tempVerRead
    return op_types
```

Algorithm 4: Calculate Read/Write Locations

```
Function GetRWLocation (op_types, tw_before):
    rw_loc[op_types.size]  // locations of read/write operations
    parallel for $i$ = 0 to sorted_ops.size do
        if op_types[i] $\in$ {currVerRead, currVerWrite} then
            rw_loc[i] = currVer
        else if op_types[i] == prevVerRead then
            rw_loc[i] = prevVer
        else // tempVer read/write, return tempVer index
            if op_types[i] == tempVerRead then
                // index is zero-based
                rw_loc[i] = {tempVer, index=tw_before[i]$-1$}
            else
                rw_loc[i] = {tempVer, index=tw_before[i]}
    return rw_loc
```

Algorithm 5: Transaction Execution Phase

```
1  Function Execute (txns[NUM_TXN]):
2      parallel for $i$ = 0 to txns.size do
3          RunTxn(txns[i])

4  Function RunTxn (txn):
5      value1 = ReadFromTable(txn.record_id1, txn.read_loc1)
6      value2 = ReadFromTable(txn.record_id2, txn.read_loc2)
       // perform transaction logic
7      if value1 is None or value2 is None then
8          abort()
9      result = SomeOperation(value1, value2)
       // no aborts can happen beyond this point
10     WriteToTable(txn.result_record_id, txn.write_loc, result)

11 Function ReadFromTable (rec_id, read_loc):
12     if rec_id = INVALID_RECORD then
13         return None
14     prevVer, currVer = GetTableVersions(table[rec_id])
15     if read_loc == prevVer then
16         read_ver = prevVer
17     else if read_loc == currVer then
18         read_ver = currVer
19     else // tempVer read
20         read_ver = tempVers[read_loc.index]
21     while read_loc $\neq$ prevVer and
           atomicRead(read_ver.eid) $\neq$ current_eid do
22         Spin()  // Wait until version is ready
23     if not read_ver.is_valid then
24         return None
25     return read_ver.data

26 Function writeToTable (rec_id, write_loc, data):
27     prevVer, currVer = GetTableVersions(table[rec_id])
28     if write_loc == currVer then
29         write_ver = currVer
30     else // tempVer write
31         write_ver = tempVers[write_loc.index]
32     PerformWrite(write_ver.data, data)
33     __threadfence()
34     atomicWrite(write_ver.eid, current_eid)
```

valid (V) data, as shown in Figure 2. An update or insert sets and a delete unsets the valid flag of the corresponding version. Read operations use the valid flag to determine if the record exists at the timestamp of the read, preventing transactions from reading invalid data (Algorithm 5, lines 23–24).

Deletion of records can happen at any point within an epoch, and a later write operation to a deleted record will re-insert it. Consequently, the record should be freed only when the last write operation to a record in an epoch is a delete. Epic tracks records that are deleted in an epoch by setting a per-record deleted flag when deletions occur to *currVer*. At the end of the epoch, these flags are scanned to generate a list of deleted records that are subsequently freed, as described later in Section 4.2. A full scan after each epoch is acceptable because the flag is one bit per record and parallel scans are efficient on GPUs.

Handling Aborts Epic eliminates concurrency-control related aborts because transactions are serialized in a predetermined order, similar to other deterministic databases. Epic allows application-level aborts (e.g., constraint violations) before any writes are performed to the database. Transactions are expected to perform their reads, buffer writes and issue aborts before any database writes. Since aborts do not occur after the first write, the writes of a transaction are made visible immediately [12].

In previous multi-versioned systems, a sentinel value is used to indicate an aborted version. Subsequent reads skip such versions and read the previous non-aborted version. This approach is not suitable for Epic since there is no version search. Instead, the aborted write operations must copy the previous version to the current version. Thus, for transactions that may abort, Epic also calculates the read location (i.e., of the previous version) for write operations during initialization.

3.4 Field Splitting

Database records often consist of multiple fields. Since Epic eliminates version search, each version of a record must contain a full copy of all of its fields. This approach adds copying overhead when a transaction updates only a few fields of a record since all of its fields must be copied from the previous version. In addition, it introduces unnecessary dependencies because every field update becomes a read-modify-write operation for the record.

Epic implements a field splitting optimization by storing different fields of a record separately. Each version now comprises only a single field. As a result, a write to a field does not require copying other fields and introduces no additional dependencies. However, the field splitting optimization adds overhead for full record operations, which need to be split into multiple per-field operations, leading to increased initialization and synchronization costs.

3.5 Recovery

Currently, Epic does not support recovery and replication. However, it can provide durability and high availability by using techniques similar to previous deterministic databases [35]. In each epoch, transaction inputs can be logged to storage on the CPU side concurrently with transaction execution. Once all inputs are logged, transaction results can be made externally visible to applications. Currently, Epic returns these results conservatively at the end of the epoch, which enables handling certain problematic transaction logic, such as infinite loops, by aborting the relevant transaction and its dependent transactions [12].

For recovery, the transaction inputs are used to replay all transactions deterministically until the last logged epoch. The replay uses the same mechanism as normal transaction processing. To reduce recovery time, Epic's two-version tables allow checkpoints to be created efficiently. The checkpointing process can run in parallel with an epoch and create a consistent database snapshot by copying the *prevVer* of each record to a different memory area (e.g., CPU memory). However, the next epoch must start after the checkpointing completes or else the resulting snapshot may be inconsistent. After creating a copy of the tables, they can be transferred to persistent storage in the background. The index and allocation information also needs to be checkpointed or rebuilt during recovery.

4 Implementation

This section describes Epic's GPU-based implementation of indexing, initialization and transaction execution phases.

4.1 Transaction Batching and Ordering

Currently, Epic batches transactions when they are generated and serially orders them by assigning a transaction ID to each transaction. In practice, the batching and ordering process can be performed without contention by batching transactions separately on each core and ordering them using a local counter. Before an epoch starts, transactions from all cores can be serialized based on the core ID and the local counter value. This method is similar to Calvin [35].

4.2 Indexing and Allocation

Epic is capable of executing tens of millions of transactions per second. Its index needs to handle hundreds of millions of operations per second, and so we use GPU-based indexing. Epic uses a hash table index to map keys to record IDs. When needed, range queries are performed in the read-write set identification phase using a range index to obtain all the keys for the read and write sets. The keys are then used to look up the record IDs in the hash table index. Epic implements indexing using CuCollection [23], a GPU-based concurrent hash table. Epic uses a modified version of a GPU B-tree [2,4] for the range index.

Since Epic's indexing operates in parallel, we ensure that read operations see all previously inserted records by performing insert operations before any indexing operations, which also prevents phantom reads. Epic does not distinguish between insert and write operations, and so it first indexes all write operations in an epoch to find the keys to be inserted (keys that are in the write set but are not found in the hash table). To allocate a record for each to-be-inserted key, Epic maintains a ring buffer of free record IDs on the GPU. To ease allocation, these keys are uniquified. Then, Epic allocates record IDs for them by removing the same number of record IDs from the ring buffer. The key-record ID mappings are then inserted in the hash table. Next, Epic indexes all read and write operations. For read operations, if a key is not found, Epic marks the read as invalid by returning a sentinel `invalid_read` value for the record ID. This value is treated as any other record ID during initialization, and then reads detect it during execution (Algorithm 5, lines 12–13). Since Epic performs inserts before read operations, a read of a non-existing record may see an index entry from a later write. A read operation detects this version as invalid during execution (see Section 3.3).

At the end of an epoch, Epic's execution phase returns the deleted record IDs (see Section 3.3). Epic garbage collects these records by appending them to the ring buffer. To free the index entries for these records, Epic also keeps a back-link array that maps record IDs to keys. The hash table and the back-link are stored in GPU memory and are only accessed by the GPU during indexing.

4.3 Multi-Version Initialization

Epic's multi-version concurrency control initialization is implemented using the CUB and Thrust parallel algorithms library. As shown in Algorithm 2, all operations, such as sorting and prefix sum, are highly parallelizable. Epic performs initialization for each table separately for ease of implementation. Each operation's record ID, transaction ID, operation ID and read-write type are stored in a 64 bit integer for efficient sorting. It is possible to prefix the record ID with a table ID and perform initialization for all tables together.

We implemented an optimized CPU-based initialization phase using Intel's TBB library but its performance was at least an order of magnitude slower than the GPU implementation, motivating our GPU-based approach.

4.4 Transaction Execution

After the concurrency control initialization phase, Epic executes the entire batch of transactions concurrently on the GPU using *warp-cooperative execution*, an approach motivated by previous work on GPU-based concurrent data structures [1, 3, 39]. Next, we provide some background on GPUs to motivate our execution approach.

GPUs provide an array of multi-threaded Streaming Multiprocessors (SMs), with each SM containing simple cores (typically 64–128 per SM). The GPU executes instructions from a group of threads, called a *warp*, in a Single Instruction, Multiple Threads (SIMT) lockstep manner on the cores of an SM, with threads executing the same instruction on different data elements. A warp typically consists of a fixed number of threads, such as 32 threads in Nvidia GPUs.

The warp-based execution model makes branch divergence an important aspect of GPU algorithm design. Branch divergence occurs when thread execution diverges due to control flow statements, such as branches, for threads within a warp. In this case, the GPU serializes the execution of the divergent paths, causing longer execution times per warp.

Instead of running a different transaction on each thread of a warp, Epic's warp-cooperative execution model uses all the threads in a warp to cooperatively execute a *single* transaction, which avoids branch divergence altogether. The threads in a warp read and write versions by accessing consecutive locations of a record. The GPU can coalesce (or combine) these contiguous memory accesses into a single request, which improves memory bandwidth utilization and is especially beneficial when transactions access large records. For example, 32 threads in a warp running the same instruction can access 128 contiguous bytes in parallel from global memory.

Although warp-cooperative execution can lead to reduced concurrency, the amount of parallelism available on modern GPUs is more than sufficient for Epic's transaction processing requirements. For example, Nvidia's A6000 GPU has 84 SMs, each capable of scheduling 1536 threads (48 warps) at a time. With the warp-cooperative execution scheme, Epic can execute $84 \times 48 = 4032$ transactions concurrently. We believe that transaction execution will not benefit from higher concurrency due to dependencies between transactions. Therefore, the benefits of avoiding branch divergence and coalesced memory access outweigh the reduced concurrency.

GPU Transaction Scheduling The GPU hardware scheduler dispatches threads on an SM at the granularity of a group of threads called a thread block. While the GPU does not provide control over the scheduling order of thread blocks (or threads within a thread block), it guarantees that an active thread runs to completion without being preempted.

Since Epic assigns a serial order to each transaction before execution, transactions must be scheduled based on their serial order. Otherwise, a later transaction may depend on an earlier transaction, which never gets to run because the later transaction holds the hardware resources. Epic schedules transactions in serial order by dynamically assigning transactions to threads when they become active. To do so, it uses a next-transaction global counter, that it increments once per block to allocate transactions for all warps within a block. Threads within the block then distribute the allocated transactions using a local counter.

4.5 Other Optimizations

Epic exploits parallelism within a transaction by splitting transactions, when possible, into multiple independent pieces. Due to its deterministic nature, these pieces can be executed concurrently while still ensuring isolation [12, 28].

Epic aims to overlap data transfer and computation on the GPU whenever possible by launching asynchronous tasks on different non-blocking CUDA streams. This approach effectively hides the latency associated with transferring transaction parameters and data. As shown in Figure 1, Epic transfers transaction parameters to the GPU. This transfer is overlapped with the execution of the previous batch of transactions. With CPU-side execution, Epic overlaps the transfer of the indexed transactions to the CPU with the initialization phase.

It is possible to pipeline Epic's CPU-side execution with GPU indexing and initialization. However, this approach complicates the index garbage collection mechanism. If a record is deleted in epoch N, its index information cannot be garbage collected until epoch $N+2$ because the indexing in epoch $N+1$ runs concurrently with the execution of epoch N. However, the same key may be re-inserted in epoch $N+1$. In this case, the index information for the record deleted in epoch N cannot be garbage collected. This issue can be resolved by tracking the epoch ID in an index entry when it is created. Epic currently does not implement this pipelined execution.

5 Evaluation

We compare the overall performance of Epic with several state-of-the-art in-memory transaction processing databases using the TPC-C, TPC-C NP and the YCSB benchmarks. Then, we provide a more detailed analysis of Epic's design.

All experiments are run on cloud server with a 32-core Epyc CPU and 512GB of memory. For all the CPU-based databases except Aria, we use 1 thread per core for a total of

32 threads. For Aria, we use the default 12 worker threads because this configuration achieves the highest throughput. We use the Nvidia A6000 GPU with 10752 CUDA cores and 48GB GDDR6 memory. The operating system is Ubuntu 22.04. All experiments are compiled with NVCC 12.0 with CUDA run time version 12.0.

5.1 Database Systems Comparison

We compare Epic against four state-of-the-art in-memory databases: STOv2 [14], Caracal [28], GaccO [6] and Aria [21]. We use the publicly available implementations of Caracal, STOv2 and Aria. Since GaccO's implementation is not publicly available, we implemented GaccO's GPU-side transaction execution based on the description in their paper. We use the default epoch sizes of 500 for Aria, 100K for Caracal, and 32768 for GaccO as specified in their papers for all experiments except for the latency experiment in Section 5.7. We use an epoch size of 100K transactions for Epic because throughput improvements become smaller beyond this epoch size, which balances throughput and latency.

STOv2 is a state-of-art in-memory CPU database. STOv2 implements and compares three concurrency control mechanisms: OCC-based Silo [36], timestamp-based TicToc [38], and a variant of MVCC-based Cicada [17]. These mechanisms are called OSTO, TSTO, and MSTO respectively. STOv2's implementations of TicToc and Cicada perform well thanks to careful attention to implementation choices. We enable both the timestamp splitting and deferred updates optimizations in STOv2. Timestamp splitting behaves similar to our field splitting optimization.

Caracal is a multi-versioned, deterministic CPU in-memory database. Similar to Epic, Caracal batches transactions and splits each epoch into an initialization phase and an execution phase. Caracal uses a version array to implement multi-version concurrency control (MVCC). Each record contains an array of versions that are created during the initialization phase and read during the execution phase. Caracal performs well under contention due to transaction batching and MVCC. However, Caracal's concurrency control mechanism keeps the version array sorted by the version ID, which imposes overhead during the initialization phase, and read operations need to perform a binary search through the version arrays. Additionally, the version array requires expensive garbage collection.

GaccO is a single-version, deterministic GPU database that uses lock-based concurrency control [6]. To support databases larger than GPU memory, GaccO proposes running transactions on both the GPU and the CPU. This CPU-GPU co-execution model requires keeping copies of CPU memory tables in GPU memory when the tables are accessed by GPU-side transactions, synchronizing updates to the tables at epoch boundaries, and delaying CPU-side transactions that conflict with GPU-side transactions.

We only compare with GaccO's GPU-based execution, so no synchronization with the CPU is needed. Similar to Epic, GaccO requires transactions' read-write sets in advance. GaccO initializes an epoch by creating a per-record lock table. For each record, all operations are sorted based on the serial ID of the transactions. The corresponding serial IDs are stored in the lock table, representing the order of lock acquisition. During the execution phase, transactions acquire locks on records deterministically by checking the lock table and waiting until the lock value matches the transaction's ID. Upon release, the lock value is advanced to match the next transaction that accesses the record. However, this lock-based concurrency control does not permit readers to share locks.

GaccO executes a transaction per thread and batches transactions by type (e.g., NewOrder in TPC-C) within an epoch to minimize warp divergence (see Section 4.4). This batching also enables GaccO to use a *commutative optimization* when highly-contended items are accessed commutatively. If an operation updates a data item commutatively then the order of performing such updates is flexible, provided the data item is not otherwise observed by its transaction and there are no other conflicting operations on the item. For instance, a transaction that increments a counter in the database row but never reads the value of the counter can implement the update using atomic instructions, without using the deterministic locking protocol. Since GaccO batches transactions by type, conflicts do not occur with other types of transactions.

However, due to this batching of transactions by type, we do not implement the full TPC-C benchmark for GaccO. For the OrderStatus and StockLevel transactions, batching by type would cause these transactions to execute on a snapshot of the database and return the same results within an epoch. Therefore, we only evaluate GaccO on the TPC-C NP and YCSB benchmarks.[1]

Aria is a deterministic database that does not require advance knowledge of read-write sets [21]. It achieves determinism by executing all transactions in a batch against a database snapshot from the previous epoch, while buffering writes and delaying commit until the end of the epoch. After all transactions have executed, Aria deterministically aborts transactions that conflict with an earlier transaction based on transaction ID ordering, and it uses a deterministic reordering optimization to reorder transactions in a batch to reduce the number of aborts. Aria assumes that the read-write sets of transactions are known after the execution phase, and uses Calvin's deterministic locking as a fallback strategy to rerun the aborted transactions after the execution phase.

Aria only implements TPC-C NP. We evaluate the variant with the fallback strategy since their paper reports that it performs better than without the fallback strategy under all contention levels on TPC-C NP.

[1] The GaccO paper also evaluates TPC-C NP on the GPU.

5.2 TPC-C

We use the TPC-C OLTP benchmark to evaluate Epic. The TPC-C benchmark simulates an OLTP workload for a warehouse management system. It consists of five transactions: NewOrder, Payment, OrderStatus, Delivery, and StockLevel.

The NewOrder transaction creates a new order for a customer by incrementing the nextOrderID field in the District table to obtain the order ID. This makes the write-set of NewOrder dependent on the execution-time value of the order ID. OrderStatus retrieves the status of the last order placed by a customer; StockLevel checks the stock level of items ordered in the last 20 transactions in a district; and Delivery processes the oldest undelivered order in a district.

To identify the read-set and write-set keys of these transactions, Epic runs the read-write set identification phase before the indexing phase. Initially, the order ID used by NewOrder is calculated using a per-district counter, which also helps determine the latest order ID for OrderStatus and StockLevel. Then, for each NewOrder transaction, the order information is inserted into a secondary index. The secondary index uses a range index keyed by the customer ID and the order ID. The secondary index also stores the items ordered in each order. OrderStatus performs a backward range scan using the customer ID and the latest order ID in the district as the key to find the last order ID for a customer. StockLevel uses the latest order ID to lookup the ordered item information to check for stock levels. Lastly, Delivery uses a per-warehouse counter to find the oldest undelivered order.

During execution, transactions can validate the read-write sets determined by the identification phase and abort transactions if they do not match the keys that would be accessed during the execution phase [35]. However, since Epic does not cause any concurrency-control related aborts, the read-write sets always match in TPC-C and so no aborts occur [11].

Furthermore, the Payment and OrderStatus transactions in the original TPC-C benchmark can be provided with a customer ID or the customer's last name. In the latter case, the customer ID is retrieved by scanning a read-only index of customers. Since existing GPU range indexes do not support variable length keys needed for scanning the last name, we simplified Payment and OrderStatus to only use the customer ID for all the databases. Other than this change, the behavior and contention level of Epic's TPC-C implementation conforms to the TPC-C specification.

TPC-C has low contention when each warehouse is assigned a separate CPU core. We vary the number of warehouses to evaluate performance under different contention levels. With a single warehouse, TPC-C becomes highly contended due to the per-warehouse Warehouse, District, and Stock tables.

STOv2 and Caracal implement the TPC-C benchmark and we compare Epic against them. Figure 4 shows the throughput of the systems. Epic outperforms the other systems under all contention levels. Under low contention, Epic benefits from the high memory bandwidth and parallelism offered by the GPU, enabling it to outperform all other systems. The two multi-versioned CPU systems, MSTO and Caracal, perform poorly under low contention due to the high overhead of MVCC. However, they perform better under high contention compared to the single version systems. As expected, Epic's performance degrades under high contention. However, due to the deterministic ordering of transactions and its efficient multi-versioning implementation, Epic outperforms the other systems under high contention as well.

5.3 TPC-C NP

The TPC-C NP benchmark is a subset of the TPC-C benchmark that consists of 50% NewOrder and 50% Payment transactions. We use this benchmark to compare with GaccO and Aria as well. The left graph in Figure 5 shows the throughput of the GPU and then the CPU systems for TPC-C NP.

The Epic, STOv2 and Caracal TPC-C NP results are qualitatively similar to TPC-C results. These databases have higher throughput on TPC-C NP under low contention because TPC-C NP has shorter transactions than TPC-C. However, they have lower throughput on TPC-C NP under high contention because TPC-C NP has higher contention than TPC-C. Caracal and Aria have lower throughput than other CPU based databases, but they also support distributed operation.

GaccO performs poorly under all contention levels because it batches transactions by type. For the Payment transaction, updates on the warehouse table require GaccO to serialize all transactions. Also, GaccO cannot run NewOrder transactions concurrently with Payment transactions, resulting in the GPU being underutilized. Additionally, GaccO's lock-based concurrency control has high overhead under contention.

Epic's performance under low contention for TPC-C NP is much higher than for TPC-C for two reasons. First, TPC-C NP does not require scanning for the latest order of a customer and lookup for ordered items and so the overhead of read-write identification is significantly lower. Second, and more importantly, TPC-C NP has short transactions that can be scheduled on GPU thread blocks (see Section 4.4) more efficiently. With TPC-C's mix of short and long transactions, a block needs to wait for the longest transaction to complete. We plan to explore scheduling strategies that co-locate long read-only transactions within blocks.

To implement GaccO's commutative optimization for TPC-C NP, we changed the NewOrder transaction to use atomic CAS instructions to update the District and Stock tables, and we changed the Payment transaction to use atomicAdd to increment the balances of the warehouse, district, and customers tables. Since the updated values are not used after the update or read by other transactions, the order of updates is flexible. The right graph in Figure 5 shows that GaccO with this optimization outperforms all systems. The throughput

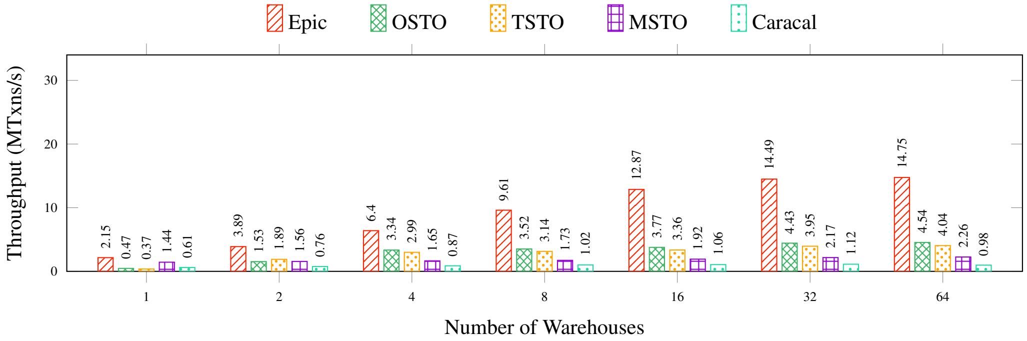


Figure 4: TPC-C Throughput

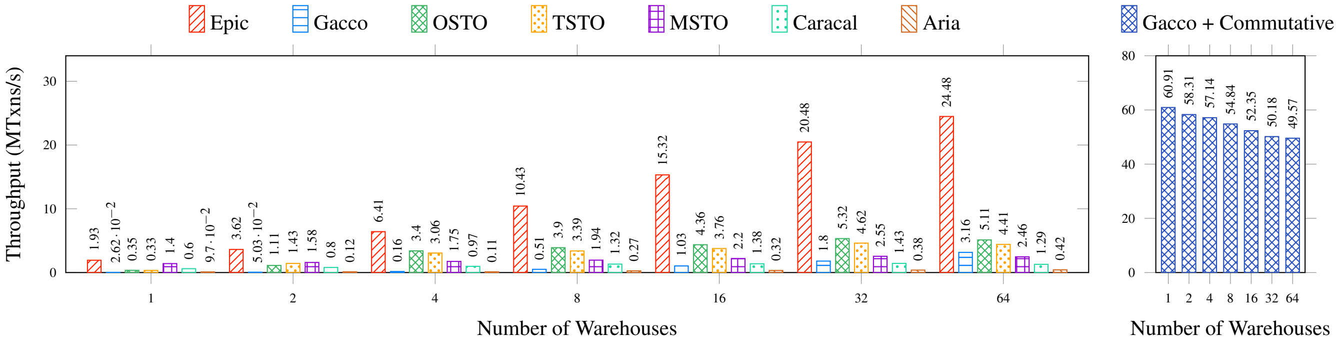


Figure 5: TPC-C NP Throughput

drops slightly with more warehouses due to decreased cache locality. This optimization eliminates concurrency control in TPC-C NP since both the NewOrder and the Payment transactions do not hold any locks. However, this optimization is not general-purpose, e.g., it doesn't allow reading the District table to validate the order ID in the NewOrder transaction.

5.4 YCSB

Next, we conduct experiments using the Yahoo! Cloud Serving Benchmark (YCSB) [9]. For the experimental setup, we use a single table consisting of 1,000,000 records. We used the standard record size in YCSB, where each record is 1000 bytes and consists of ten 100 byte fields. We performed experiments using four YCSB workloads, as shown in Figure 6. In all workloads, a read operation reads the entire record. An update operation replaces the value of one randomly chosen field. A read-modify-write (RMW) operation reads a record and updates a randomly chosen field. For our evaluation, we group 10 operations to form a transaction. We vary the Zipfian skew factor θ from 0 to 0.99 to vary contention levels.

Figure 7 shows the throughput of the six databases for the four YCSB workloads with increasing contention levels. Epic outperforms all other databases for all workloads. In YCSB-A, Epic's performance drops significantly under high contention. Epic performs a read-modify-write operation for each update operation. Even when an update only writes to a part of the record, the entire record needs to be copied from the previous version. As a result, the read-modify-write operations form long dependency chains under high contention. In the YCSB-B benchmark, where the write ratio is low, Epic's performance drops more gently under high contention. In the read-only YCSB-C benchmark, Epic achieves high throughput due to the high memory bandwidth of GPUs. Finally, in the YCSB-F benchmark, Epic shows a similar trend as YCSB-A, where performance drops significantly under high contention because Epic performs the same read-modify-write operations for both YCSB-A and YCSB-F. In some workloads, Epic's throughput increases slightly from low to medium contention level (skew factor 0.0 to 0.5) due to better cache locality that improves GPU indexing performance. The execution phase in Epic also benefits from this better cache locality, especially for read-only YCSB-C.

Workload	Description	Operations
YCSB-A	Update heavy	Read: 50%, Update: 50%
YCSB-B	Read heavy	Read: 95%, Update: 5%
YCSB-C	Read only	Read: 100%
YCSB-F	Read-modify-write	Read: 50%, RMW: 50%

Figure 6: YCSB Workload Configurations

We also evaluate the performance of Epic with field splitting, as described in Section 3.4. In this case, each record is divided into ten fields, and each field is treated as a separate

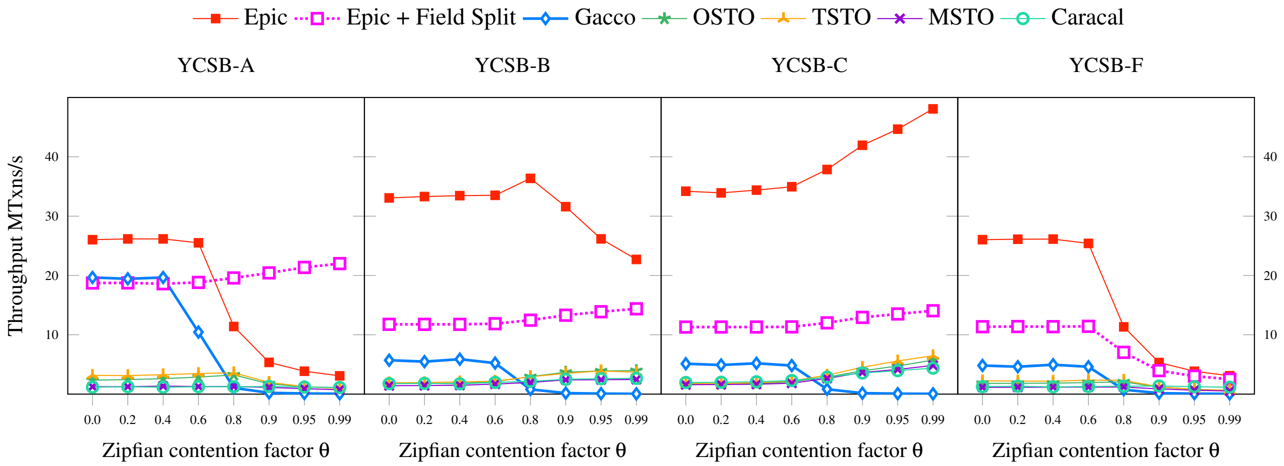


Figure 7: YCSB Throughput

data item from the perspective of concurrency control. As a result, each full-record read operation needs to perform 10 field reads, each requiring separate synchronization. As a result, the number of read operations in the initialization phase increases by 10x, and read performance decreases. On the other hand, since each field is treated separately, an update operation on a single field does not require copying the rest of the fields from the previous versions, improving update performance. As shown in Figure 7, Epic with field splitting performs better than default Epic under YCSB-A with high contention. However, Epic's performance is lower in YCSB-B, YCSB-C, and YCSB-F, where the read ratio is higher.

GaccO shows similar trends under all workloads, performing well under low contention, but its performance drops significantly under high contention due to its lock-based concurrency control. GaccO's initialization phase is simpler and faster than Epic's MVCC initialization but its lock-based concurrency control does not allow readers to share locks, causing its performance to drop significantly under contention, even under a read-only workload. GaccO's assigns each transaction to a single GPU thread, which causes non-coalesced memory accesses that reduce memory bandwidth utilization. As a result, GaccO's performance decreases when the ratio of read operations increases (YCSB-A and YCSB-B) because read operations retrieve the entire record. GaccO's commutative operation optimization cannot be applied to YCSB workloads (except YCSB-C) because other transactions read the values of the data items updated. Therefore, we did not implement this optimization for the YCSB workloads.

Both multi-versioned systems (MSTO and Caracal) suffer from the same extra dependency as Epic in YCSB-A. Therefore, they exhibit similar trends for YCSB-A and YCSB-F. OSTO and TSTO perform well under low contention, but their performance drops significantly under high contention with write-heavy workloads (YCSB-A and YCSB-F). This is due to increased aborts resulting from a high conflict rate. In read-heavy workloads (YCSB-B and YCSB-C), OSTO and TSTO outperform MSTO and Caracal due to their lightweight concurrency control mechanisms. However, Caracal achieves higher throughput than OSTO and TSTO in YCSB-A and YCSB-F under high contention because its MVCC-based concurrency control allows readers to run in parallel with writers.

5.5 CPU-side Execution

Next, we evaluate the performance of Epic's CPU-side execution using the same setup for the TPC-C, TPC-C NP and YCSB benchmarks. As mentioned in Section 3.3, the GPU performs indexing and initialization for the epoch and then transfers the execution plan to the CPU. This data transfer takes roughly 4 ms for the TPC-C NP and YCSB benchmarks and 6 ms for TPC-C, which contains long running queries with more operations. The transactions are then executed on the CPU. The throughput reported in Figure 8 includes the time for indexing, initialization, data transfer and execution because Epic currently does not implement pipelining.

With TPC-C and TPC-C NP, CPU-side execution achieves higher throughput than GPU-side execution with a single warehouse. We believe that the contended Payment transaction limits Epic from utilizing the parallelism of the GPU effectively. On the CPU, Epic's execution time synchronization is more efficient as the atomic flags can be directly communicated through the CPU cache. However, with more warehouses, GPU-side execution achieves higher throughput due to the higher parallelism and memory bandwidth of the GPU.

With CPU-side execution, Epic achieves lower throughput in TPC-C than TPC-C NP under low contention due to the longer data transfer time. However, Epic performs better for TPC-C with a single warehouse because TPC-C has lower contention than TPC-C NP.

With YCSB, each transaction reads several records, and so CPU-side execution is limited by memory bandwidth and la-

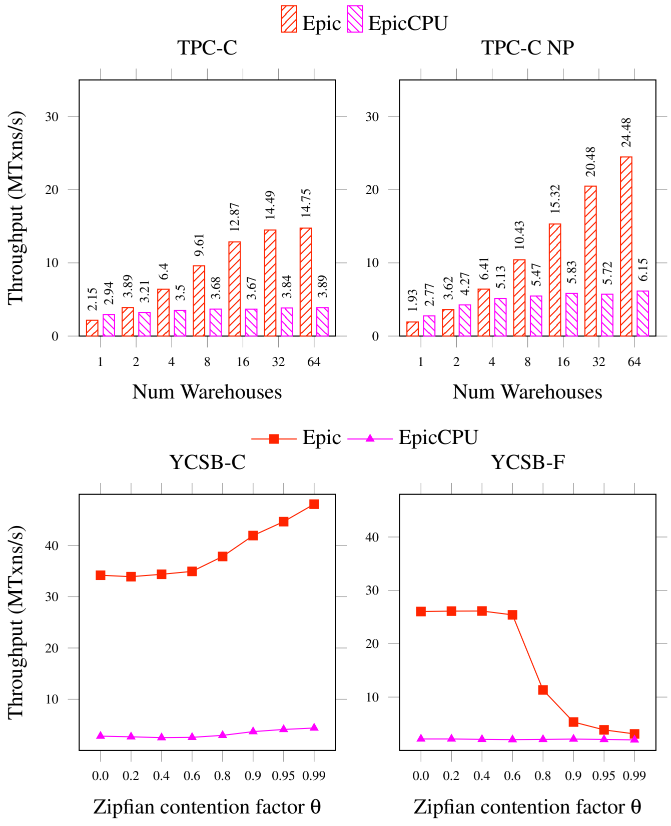


Figure 8: Throughput with CPU-side Execution

tency. For read-only YCSB-C, CPU-side execution has much lower throughput than GPU-side execution. Throughput increases slightly under contention due to cache locality. For YCSB-F, CPU-side execution throughput is bottlenecked by memory bandwidth at low contention and achieves similar throughput as GPU-side execution under high contention. YCSB-A and YCSB-B show similar trends so we omit them.

For all the three benchmarks, Epic's CPU-side execution achieves comparable throughput to OSTO and TSTO under low contention because Epic's GPU initialization is efficient. Under high contention, Epic outperforms OSTO by 6.2x and TSTO by 7.9x for TPC-C single warehouse and both by 3.2x for YCSB-F with a 0.99 skew factor due to its multi-versioning. Epic-CPU outperforms both multi-version systems, MSTO and Caracal, under all workloads because Epic's MVCC initialization is efficient and, unlike MSTO and Caracal, Epic's CPU-side execution runs without performing expensive version search.

5.6 Run Time Breakdown

Figure 9 shows the breakdown of per-epoch run time for Epic running TPC-C with the CPU- and GPU-execution model. The figure shows that the initialization time is similar for both low and high contention levels because Epic's initialization phase is unaffected by the contention level. The GPU execution time is significantly longer under high contention because transaction dependencies reduce GPU utilization.

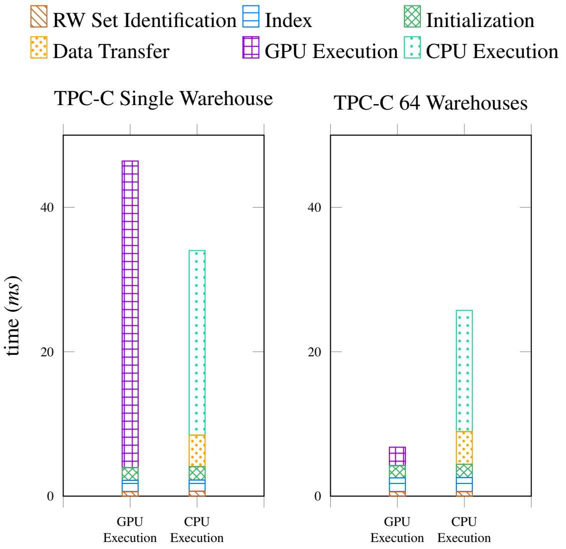


Figure 9: Epic Run Time Breakdown

For CPU execution, the indexed transactions and the transaction execution plans need to be transferred from the GPU to the CPU. Depending on the complexity of the transaction, the data transfer time can vary but is a significant portion of the total run time. Pipelining the GPU and CPU phases will help reduce the epoch run time.

5.7 Latency

In this experiment, we evaluate Epic's throughput and latency for different epoch sizes by comparing against the GaccO, Caracal, and Aria deterministic databases. We show TPC-C NP results because our GaccO implementation and Aria implement TPC-C NP. We also show YCSB-F results (but not for Aria, which doesn't implement it). For both workloads, we show results under low and high contention. Epic's results for TPC-C are not shown but they are similar to TPC-C NP.

We vary the epoch size from 500 to 200K transactions/epoch. Epic batches transactions during the previous epoch and the benchmarks do not cause aborts, so Epic's average transaction latency is 1.5× the epoch run time.

Figure 10 shows the throughput and average latency of the four systems. Each point on a line represents an epoch size. The lines start at 5000 for Caracal (which crashes at lower epoch sizes) and 500 for all other systems. The lines also show some key epoch sizes, e.g., at maximum throughput and at the knee of the curve. In all workloads, Epic achieves higher throughput with increasing epoch size. Intuitively, a larger epoch enables higher parallelism and amortizes overheads at the cost of transaction latency. Similarly, Caracal's throughput increases with larger epoch sizes.

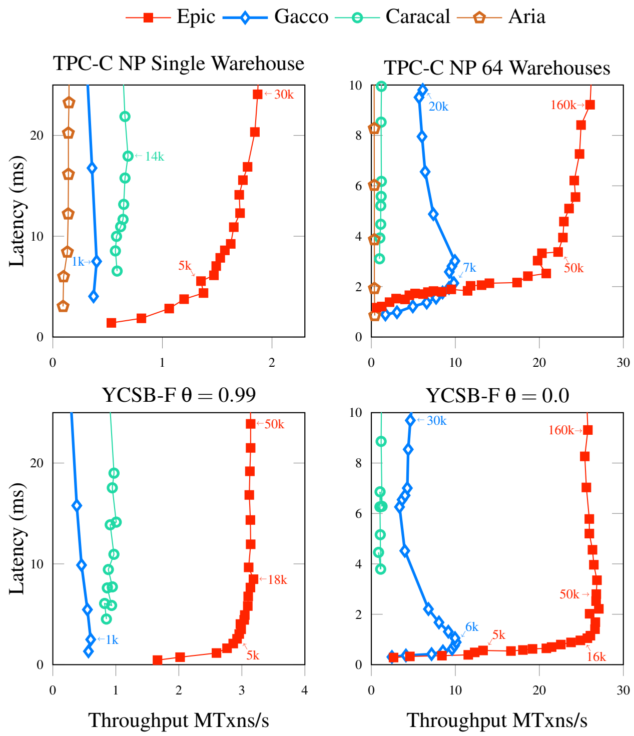


Figure 10: Latency vs. Throughput

GaccO's throughput increases with larger epoch sizes initially but then decreases. We believe that GaccO's lock-based scheduling performance degrades with increasing number of concurrent transactions. We plan to investigate this issue.

Aria's throughput decreases with larger epoch sizes under low contention because more transactions are deterministically aborted. However, Aria benefits from a larger epoch size under high contention. In this case, Aria's deterministic scheduling mechanism aborts a majority of transactions. The aborted transactions are rerun using the deterministic locking fallback strategy, which is more efficient at larger epoch sizes.

Overall, Epic achieves comparable latency to other systems at small epoch sizes. Epic has higher latency than GaccO at small epoch sizes because its multi-version initialization phase is slower and the small epoch size does not allow it to amortize this overhead. However, beyond roughly 2 ms average transaction latency, Epic outperforms all other systems.

5.8 Impact of Aborts

To evaluate the impact of aborts on Epic's performance, we run a micro-benchmark where each transaction reads and updates 10 records. The keys are generated using a Zipfian distribution with $\theta = 0.8$ for medium contention. Transactions abort when the read-set or the write-set is predicted incorrectly, and aborted transactions are rerun in the next epoch. We vary the abort rate for the experiments. We assume that the read-set and write-set are known after a transaction executes, and so an aborted transaction will not abort again when rerun, similar to Aria's assumption for its fallback strategy [21].

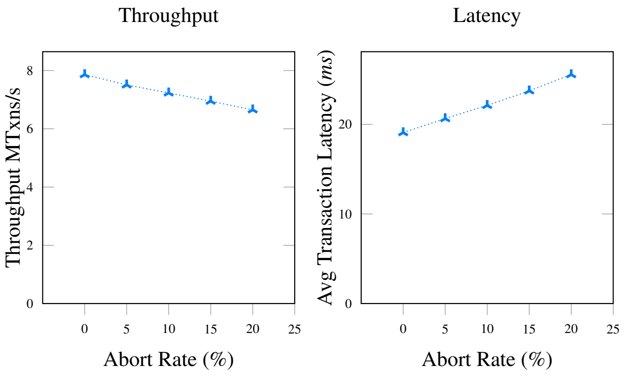


Figure 11: Impact of Abort Rate

Figure 11 shows Epic's throughput and average latency. As the abort rate increases, Epic's throughput decreases and latency increases roughly linearly. Aborted transactions are rerun in the next epoch, which increases their latency and requires additional work.

6 Conclusions

Multi-versioning schemes for transaction processing systems have traditionally been popular because they provide good performance for a range of workloads, including for long-running transactions and contended workloads. With in-memory databases increasingly being used for applications requiring high-throughput transaction processing, several multi-version schemes have been proposed for in-memory databases. However, these schemes have significant costs associated with version search and storage, garbage collection, index management.

This work proposes a novel design for multi-versioning that takes advantage of the predetermined ordering of transactions and known read-write sets in deterministic databases to eliminate version search by efficiently pre-calculating the version location of each read/write operation. Our batching design helps reduce version allocation, garbage collection and indexing overheads as well. Our design is parallelizable and so we explore accelerating transaction processing on GPUs. Our evaluation shows that our multi-versioned, GPU database performs well under both low and high contention workloads and significantly outperforms state-of-the-art systems.

Acknowledgments

We thank our shepherd, Eddie Kohler, and the anonymous reviewers for their valuable feedback.

References

[1] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–429, May 2018.

[2] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a high-performance GPU B-tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2019, pages 145–157, February 2019.

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 145–157, New York, NY, USA, February 2019. Association for Computing Machinery.

[4] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. A GPU multiversion B-tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT 2022, October 2022.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[6] Nils Boeschen and Carsten Binnig. GaccO - A GPU-accelerated OLTP DBMS. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1003–1016, New York, NY, USA, June 2022. Association for Computing Machinery.

[7] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), dec 2009.

[8] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, September 1986.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, June 2010. Association for Computing Machinery.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the International Conference on Management of Data - SIGMOD*, pages 1243–1254. ACM, 2013.

[11] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[12] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, January 2017.

[13] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, February 2011.

[14] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *The VLDB Journal*, January 2022.

[15] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687, New York, NY, USA, June 2016. Association for Computing Machinery.

[16] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, December 2011.

[17] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, New York, NY, USA, May 2017. Association for Computing Machinery.

[18] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. Mgcrab: Transaction crabbing for live migration in deterministic database systems. *Proc. VLDB Endow.*, 12(5):597–610, jan 2019.

[19] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1156–1168, New York, NY, USA, 2021. Association for Computing Machinery.

[20] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version Concurrency via Timestamp Range Conflict Management. In *2012 IEEE 28th International Conference on Data Engineering*, pages 714–725, April 2012.

[21] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment*, 13(12):2047–2060, July 2020.

[22] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, New York, NY, USA, May 2015. Association for Computing Machinery.

[23] Nvidia. Cucollection. https://github.com/NVIDIA/cuCollections, 2023.

[24] Oracle. TimesTen In-Memory Database FAQ. https://www.oracle.com/database/technologies/timesten-faq.html, 2021.

[25] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, aug 2012.

[26] Thamir M. Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 13–25, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment*, 10(13):2025–2036, September 2017.

[28] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 180–194, New York, NY, USA, October 2021. Association for Computing Machinery.

[29] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Massachusetts Institute of Technology, 1978.

[30] David P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, feb 1983.

[31] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, jul 2019.

[32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing database locking contention through multi-version concurrency. *Proceedings of the VLDB Endowment*, 7(13):1331–1342, August 2014.

[33] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 69–84, 2022.

[34] Vishal Sikka, Franz Färber, and Wolfgang Lehner. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, May 2012. Association for Computing Machinery.

[36] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, November 2013. Association for Computing Machinery.

[37] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, mar 2017.

[38] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, June 2016. Association for Computing Machinery.

[39] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, July 2015.

A Artifact Appendix

Abstract

We implement Epic, the first multi-versioned GPU-based deterministic OLTP database. Epic batches transactions into epochs and establishes a serial ordering of transactions within a batch before transaction execution. Epic performs concurrency control initialization for a batch of transactions before execution, avoiding version search and reducing version allocation and garbage collection overheads. Epic runs on the GPU to accelerate concurrency control initialization and parallelize batched transaction execution. In addition, Epic supports larger datasets with a CPU execution model. We evaluate Epic using the TPC-C and YCSB benchmarks and compare it with state-of-the-art systems: STOv2, Caracal, Gacco, and Aria.

Scope

The artifact allows reproduction of the results of the paper, including the performance evaluation of Epic using the TPC-C and YCSB benchmarks, the latency and throughput comparison, and the performance evaluation of Epic with varying abort rates.

All the experiments except the runtime breakdown in Figure 9 can be reproduced using the artifact. The runtime breakdown is created by retrieving the runtime information manually, and we do not have a script to automate this process.

Additionally, the artifact cannot perform the performance evaluation for Aria due to the conflict of dependencies. Therefore, the Aria results in Figure 5 and Figure 7 are not reproducible using the artifact.

Contents

The artifact repository contains the source code of Epic, STOv2, and Caracal as separate submodules. We used our best-effort implementation of Gacco, and the source code is included in the Epic submodule. The repository contains scripts to run the experiments and generate the figures in the paper. The repository also contains scripts to install the necessary dependencies and set up the experiment environment. The README file in the repository provides detailed instructions on how to run the artifact.

Hosting

Our artifact repository is hosted on GitHub at `https://github.com/ShujianQian/epic-artifact/commit/9303f4d2b1fa8368de0dbdc24bcd798585ceb920`.

More details on how to set up the experiment environment, run the experiments, and reproduce the results are provided in the README file in the repository.

Requirements

Our experiments require running on servers equipped with GPUs. We used FluidStack to host on-demand virtual GPU servers. Our artifact repository contains instructions on how to set up the virtual servers and run the experiments.

Alternatively, the experiments can be run on machines with NVIDIA GPUs. The artifact repository is tested for machines with more than 32 CPU cores, 128GB of RAM, and NVIDIA GPUs of compute capability 8.6 and GPU memory of 48GB. The artifact repository contains scripts to install the necessary dependencies and run the experiments.

Burstable Cloud Block Storage with Data Processing Units

Junyi Shu*† Kun Qian† Ennan Zhai† Xuanzhe Liu* Xin Jin*

*School of Computer Science, Peking University †Alibaba Cloud

Abstract

Cloud block storage (CBS) is a key pillar of public clouds. Today's CBS distinguishes itself from physical counterparts (e.g., SSDs) by offering unique burst capability as well as enhanced throughput, capacity, and availability. We conduct an initial characterization of our CBS product, a globally deployed cloud block storage service at public cloud provider Alibaba Cloud. A key observation is that the storage agent (SA) running on a data processing unit (DPU) which connects user VMs to the backend storage is the major source of performance fluctuation with burst capability provided. In this paper, we propose a hardware-software co-designed I/O scheduling system BurstCBS to address load imbalance and tenant interference at SA. BurstCBS exploits high-performance queue scaling to achieve near-perfect load balancing at line rate. To mitigate tenant interference, we design a novel burstable I/O scheduler that prioritizes resource allocation for base-level usage while supporting bursts. We employ a vectorized I/O cost estimator for comprehensive measurements of the consumed resources of different types of I/Os. Our evaluation shows that BurstCBS reduces average latency by up to 85% and provides up to 5× throughput for base-level tenants under congestion with minimal overhead. We verify the benefits brought by BurstCBS with a database service that internally relies on CBS, and show that up to 83% latency reduction is observed on customer workloads.

1 Introduction

Cloud Block Storage (CBS) is a fundamental storage service on public clouds. It provides virtualized block-level storage volumes that can be dynamically provisioned and attached to compute instances. Beyond what an SSD can already offer, a CBS disk can provide additional benefits, including millions of IOPS, tens of terabytes capacity, higher durability with data replication, and out-of-box encryption support [1–4].

CBS adopts storage disaggregation to achieve better elasticity [5–9]. The disaggregated architecture of CBS empowers public clouds to independently scale storage and compute resources. Block storage volumes can be created, resized, and destroyed on demand without disrupting compute instances. Such agility and flexibility allow enterprises to right-size storage for different workloads in the cloud.

Due to the wide adoption of storage disaggregation, researchers have studied various technical aspects of it, including SSD co-optimization [10–12], kernel improvement [13, 14], kernel bypassing [7, 8, 10–12], storage-oriented network [5–8], performance analysis [15], and applications [16]. The performance of disaggregated storage is rapidly improving with the aforementioned system advances as well as the adoption of faster network/storage devices.

Through our analysis of the operational statistics of our production clusters and characterization of the storage agent (SA) which connects user VMs to the backend storage, we draw three key insights of CBS at Alibaba Cloud.

First, the bottleneck is shifted to compute nodes. Many existing systems target the bottleneck of SSDs [10, 11, 17]. We show that the distributed nature of cloud storage backend design and the over-provisioning tendency of cloud users result in relatively low utilization of storage servers and devices in terms of throughput. These characteristics of CBS have shifted performance bottleneck from the backend to the compute nodes. While backend traffic is well balanced by design and enjoys the benefit of the law of large numbers, compute nodes experience frequent traffic fluctuation due to unpredictable usage patterns of users.

Second, the burst capability of cloud block storage amplifies the chance of congestion. Cloud providers offer on-demand access to extra CPU and storage resources for running instances [18–20]. User VMs are provided with a *base-level* performance and the burst capability allows them to handle temporary traffic *bursts* above the base level. However, it can put a strain on the underlying server and cause performance degradation. When a single VM bursts to an extremely high throughput or multiple VMs on the same server burst concurrently, VMs compete for the limited available resources,

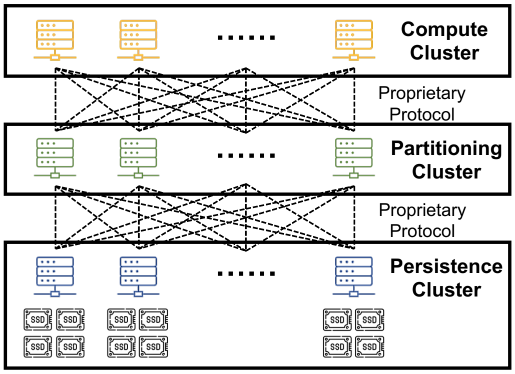


Figure 1: CBS architecture.

which leads to congestion. Moreover, heavy bursting VMs can crowd out other VMs running at a steady state.

Third, lack of resource scheduling at data processing units (DPUs) is the root cause of performance interference. Today, DPUs have become a standard component on compute nodes in the cloud [21–25]. A DPU runs hypervisor and network/storage functions freeing up host CPUs for customer usage. However, handling bursts on a DPU is extremely challenging for SA, as a DPU only possesses limited processing capability. Therefore, SA must strive for high resource utilization and provide performance isolation under congestion.

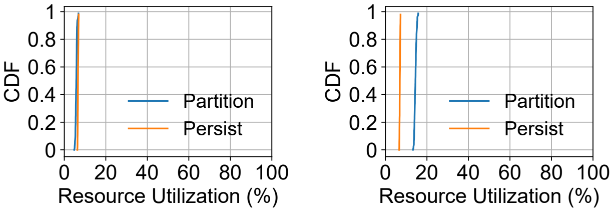
With a brief summary, there are two extra implicit requirements for CBS when supporting burst. (i) *Comprehensive resource utilization:* SA should fully leverage the available processing capacity of the DPU to support higher bursts and avoid congestion in the first place. (ii) *Base-level performance guarantee:* SLOs must not be violated for a tenant who does not exceed its base-level provisioning.

However, the existing SA meets neither of the goals by default. First, SA maps user queues statically to I/O threads running on DPU cores. An I/O thread can become congested while other threads are still idle when a certain user queue starts bursting. Second, an I/O thread serves I/Os in a First-Come-First-Serve (FCFS) manner as long as a VM is within its burst limit. VMs running at a steady state suffer high queuing delay as a result when other VMs are bursting.

In this paper, to overcome load imbalance and tenant interference induced by bursts, we present BurstCBS, a storage I/O scheduling system that leverages the hardware features of our custom xDPU and software characteristics of our SA. The design of BurstCBS offers a number of benefits. First, it equally distributes I/Os to all I/O threads. Second, it allows high bursts, while providing guarantees on base-level performance per VM. Third, it detects different types of bottlenecks on xDPU, avoiding triggering false congestion control. Finally, BurstCBS achieves the above benefits while keeping high resource utilization and minimal scheduling delay.

BurstCBS integrates three key techniques to realize the aforementioned benefits. First, considering the significant overhead introduced by inter-thread messaging, we leverage hardware capability and extensively optimize SA to implement I/O-granularity load balancing at line rate. With limited DPU memory, we design a two-tier memory pool that dynamically adjusts shared and queue-dedicated memory to achieve high I/O performance and memory efficiency. Second, we design a novel burstable I/O scheduler that is aware of the base-level/burst provisioning of each tenant. Rather than achieving high resource utilization with a work-conserving scheduler or guaranteeing strong performance isolation among multiple tenants, it dynamically attempts to provide burst capability to tenants that have excessive demand while monitoring and protecting base-level performance for other tenants in real time. Third, we design a vectorized I/O cost estimator that decouples potential resource bottlenecks that SA may encounter on xDPU. It vectorizes I/O cost and adjusts its estimation with a delay-based approach to allow the scheduling algorithm to react to resource contention effectively.

We implement BurstCBS as a standalone system package running on xDPU, and integrate it into the existing I/O workflow of SA. We conduct a comprehensive evaluation with various types of workloads and show that BurstCBS effectively protects base-level tenants while incurring a negligible overhead. Overall, BurstCBS reduces average latency by up to 85% and achieves up to 5× throughput for base-level tenants under congestion. A database service that internally relies on CBS reports that up to 83% reduction of SQL query average latency is observed with BurstCBS deployed.



(a) IOPS

(b) BPS

Figure 2: CDF of IOPS and BPS (bits per second) utilization in a production cluster. 100% IOPS utilization is defined as the maximum IOPS a node can saturate on a 4KB random I/O workload and 100% BPS utilization is defined as the maximum bandwidth that a node can saturate on a 128KB sequential I/O workload.

2 Background

CBS architecture. Figure 1 shows the three-layer architecture of CBS at Alibaba Cloud. User VMs are hosted in the compute cluster, and all I/Os generated by VMs are forwarded to the partitioning cluster for further processing. The partitioning cluster controls data placement and failover, hiding the complexity from the compute cluster [5, 9]. The persis-

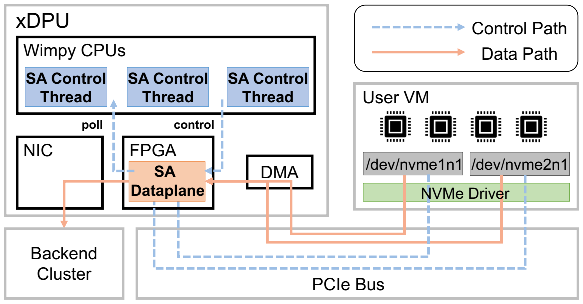
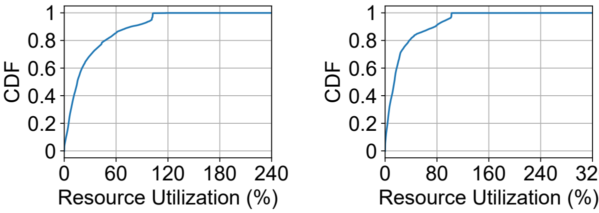


Figure 3: I/O workflow on xDPU.

Table 1: Block storage burst capability of public clouds.

	CloudA	CloudB	Alibaba Cloud
Burst support	✓	✓	✓
Credit-based burst	✓	✓	✓
Paid burst	✗	✓	✓
Max burst IOPS	3k	30k	1000k
Max burst BPS (MB/s)	N/A	1000	4096



(a) IOPS

(b) BPS

Figure 4: CDF of base-level throughput utilization in a production cluster.

tence cluster has tens of SSDs equipped on each node and is responsible for data persistence. Nodes in different layers are fully connected through a proprietary storage-optimized protocol to achieve load balancing at the backend clusters (i.e., partitioning cluster and persistence cluster).

CBS characteristics. A virtualized user disk is divided into multiple segments which are then distributed among all partitioning nodes. A partitioning node again divides each segment into smaller chunks which are evenly stored and replicated among persistence nodes. With this two-layer load-balancing design, storage I/Os are evenly distributed among all partitioning nodes and persistence nodes. An interesting phenomenon in the backend clusters is the asymmetry between IOPS/BPS utilization and disk capacity utilization. Figure 2 shows the distribution of IOPS/BPS per backend node in one of our most active production clusters in a peak hour. Storage accesses are balanced among all backend nodes resulting in low IOPS/BPS utilization on them while 78% disk capacity is utilized.

xDPU. At Alibaba Cloud, we design and build xDPU, an SoC that offloads infrastructure services from CPUs of compute nodes. It consists of its own compute resources (CPU, memory), programmable hardware accelerators (FPGA), network interfaces, and a DMA engine that can directly access guest VM memory over PCIe. The latest version of xDPU integrates eight 2.0GHz cores which are shared among storage, network, and administration functions. There are two 100Gbps network Ethernet ports available for use, and the DMA engine has about the same data movement throughput.

Storage agent. A storage agent (SA) is installed on an xDPU. It abstracts virtualized storage for VMs and connects the backend storage. SA consists of a control plane and a data plane. The SA control plane runs in wimpy cores of xDPU in user space. Among all cores, 2–4 of them are dedicated to SA control threads. The SA datapath logic is burned into FPGA, which moves data with the assistance of the DMA engine. A total of 100Gbps NIC/PCIe bandwidth is made available for SA. In Figure 3, we take NVMe WRITE operation as an example to explain how an I/O is processed by SA on xDPU. When a VM issues an NVMe WRITE command, the command is directly forwarded to FPGA on xDPU. The control threads of SA keep polling FPGA for new I/Os. When a control thread receives a new I/O, it splits the I/O into packets and constructs headers for them. Meanwhile, FPGA fetches the actual data via DMA. Once the header and body are ready, FPGA sends the packet to the backend via fabric.

We adopt an FPGA-CPU cooperative SA design instead of a fully FPGA-offloaded solution for three reasons. First, there is complex branching in I/O splitting and packet encapsulation for cloud block storage, which can hardly utilize the parallelism provided by FPGA. The wimpy CPU cores on xDPU yet have a much higher clock speed than FPGA, which makes CPU cores the right place to handle that part of logic. Second, SA must maintain a significant amount of states including thousands of connections to the distributed backend. The FPGA on xDPU does not have that much memory, while FPGA with more memory does not justify the cost. Third, from an engineering perspective, development and testing of FPGA code require much more effort which prevents us from rolling out new features quickly. We only consider offloading a software feature when it becomes mature enough.

3 Key Observations and Implications

We explain why the partitioning and persistence clusters are not the bottleneck in §2. In this section, we share two observations on the compute nodes of production CBS at Alibaba Cloud and reveal the corresponding challenges of providing predictable performance for cloud block storage.

Observation 1: The Burst capability of CBS makes compute nodes a common bottleneck. Conventionally, cloud providers provision VMs with fixed CPU, memory, and I/O resources to customers. However, a fixed amount of resources can barely match the dynamic workload faced by cloud users.

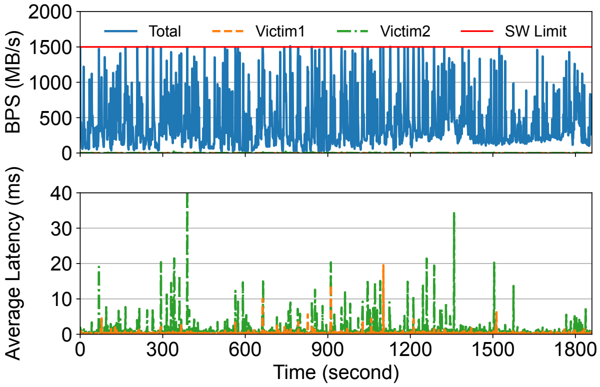


Figure 5: An incident of tenant interference under burst.

To bridge this gap, public clouds provide burstable VMs [26–29] as an option. Essentially, burstable VMs provide a base level of resources with the ability to burst above that when needed. Burstable VMs often come with a credit system. A VM accumulates credits when running below its base-level throughput while spending them to burst when it needs extra resources to saturate its demand.

Making block storage burstable is crucial for delivering burstable VMs because many workloads are bottlenecked on storage I/Os. In Table 1, we compare the burst capability of three widely used public clouds. All three cloud providers support credit-based burst which accumulates tokens for a user that does not use all of its base-level throughput. These tokens allow a user to burst when its desired throughput is beyond its provisioning. CloudB and Alibaba Cloud further allow users to burst on demand and pay for the extra throughput. What makes our case unique is that we allow a disk to burst up to 1 million IOPS and 4GB/s read/write BPS (subject to VM instance types and configurations).

We decide to allow this extreme burst capability for two reasons. First, it is a substantial requirement of our customers, as some of our CBS users have extremely bursty traffic. Figure 4 shows the distribution of base-level throughput utilization per disk in a production cluster. Some disk bursts over 300% of its base-level throughput. Second, fulfilling the burst requirement can further improve the overall resource utilization as well. The majority of our users over-provision their base-level throughput. In Figure 4, over 80% of disks use less than half of their base-level throughput. This is not a unique phenomenon at Alibaba Cloud. There is also previous work that reports the over-provisioning tendency of cloud users [30].

Although credit-based burst is attractive, it raises a strong challenge for us to provide predictable performance. Because users keep accumulating tokens as long as they are below base-level throughput, many VMs may possess tokens and start to burst at the same time. When it happens, every tenant on the impacted compute node observes higher latency and lower throughput due to congestion.

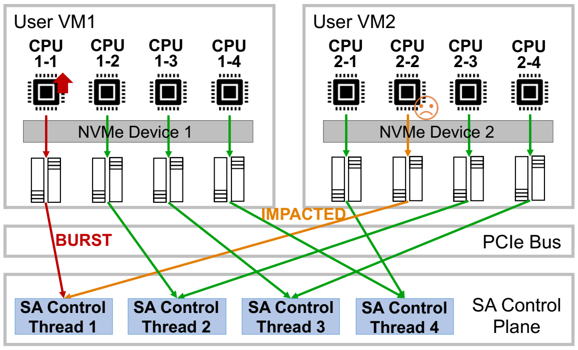


Figure 6: Impact of imbalanced load.

Our existing system addresses this problem by limiting the number of co-located VMs. Under a simplified model, assume a VM has the probability p to burst in a time interval, and we have N VMs on the compute node. We know that VMs will experience performance interference if k out of N VMs burst concurrently. The chance of performance interference is:

$$P(X \geq k) = 1 - \sum_{i=0}^{k-1} \binom{N}{i} (1-p)^{N-i} p^i \tag{1}$$

If we want to limit this probability to a small number, we have to limit the number of VMs that can co-locate on a compute node, which hurts the overall resource utilization. In production, over 95% of the compute nodes have 32 or fewer VMs allocated, which helps us maintain our SLA during 99% of the time. Indeed, although we have made this number small enough, we still observe performance interference occasionally. Figure 5 shows one such incident. During half an hour, multiple tenants were bursting and the software limit on xDPU was frequently triggered resulting in throttling. Although Victim1 and Victim2 ran below their base-level BPS steadily, they both observed many unexpected millisecond-scale average latency spikes during that time.

Observation 2: Inter-thread load imbalance and intra-thread resource contention are the major sources of performance interference on xDPU. In a VM, each vCPU corresponds to a queue pair. These queue pairs are mapped to an SA control thread in a round-robin manner by FPGA on xDPU. However, this mapping cannot adapt to dynamic workload in real time and is prone to cause load imbalance among SA control threads. Figure 6 gives an example. CPU 1-1 of VM 1 and CPU 2-2 of VM 2 are both mapped to SA control thread 1. When CPU 1-1 is creating a burst I/O stream, control thread 1 becomes congested. I/Os from CPU 2-2 experience a high queuing delay even if other control threads are idle. In short, uneven I/O intensity per vCPU created by users and static vCPU to SA control thread mapping through FPGA together lead to inter-thread load imbalance.

Bursting from a single thread in a VM is a common pattern of many I/O intensive applications because they are generally

Table 2: Parallel execution support of popular databases. Non-modifying ops refer to operations that do not change any database records.

	PE support	Enabled by default
Oracle	✓	✗
MySQL	✗	✗
SQL server	non-modifying ops	✓
PostgreSQL	non-modifying ops	✓
MongoDB	✓	✓
Redis	✗	✗
Elasticsearch	✓	✓
Db2	✓	✗
SQLite	non-modifying ops	✗
Access	✗	✗

built under the assumption that host CPU is not the bottleneck. In Table 2, we surveyed parallel execution support of the top 10 database management systems in terms of popularity [31]. Relational databases either do not fully support parallel subqueries on multiple CPUs or do not enable parallel execution by default. Redis is also known for its single-threaded design.

Our online experience also confirms this phenomenon. Figure 7 shows the I/O intensity per vCPU for 1000 randomly sampled 4-core VMs in a minute in a production cluster. Over 80% of the total I/Os are from the most I/O-intensive core.

Besides inter-thread load imbalance, intra-thread resource contention is another cause of unexpected performance degradation. Processing an I/O consumes CPU cycles in the SA control plane to construct the packet header and interconnect bandwidth in the SA data plane to transmit the actual data. When there is a resource contention, an I/O has to wait in queues until the resources are available.

We observe there are two typical cases that an I/O stream is impacted as an undesired result. First, a burst tenant with a high I/O parallelism has a significantly better chance of acquiring the resources than a base-level tenant. In Figure 8a, when we increase the I/O parallelism of a burst tenant which creates a mixture of 4KB to 128KB I/Os in the background, serial write I/Os from the base-level tenant are also queued up and average latency increases sharply due to HoL blocking.

Second, various CBS product offerings are available to customers. Through our measurement and analysis, different CBS product offerings differ in their capability on competing SA resources since they have significantly different I/O processing pipelines. For example, in Figure 8b, we start a ProductA disk and let it burst in the background, and no matter how much I/O parallelism we add to the ProductB disk, it cannot reach a similar level of throughput to ProductA.

Summary of implications. Based on our observations, we draw a few important implications for designing BurstCBS:

- The performance bottleneck is generally on compute nodes rather than backend servers and devices.
- The burst capability of cloud block storage that we must support is a main trigger of this bottleneck.

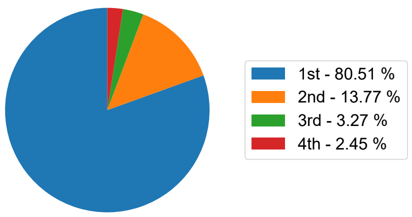
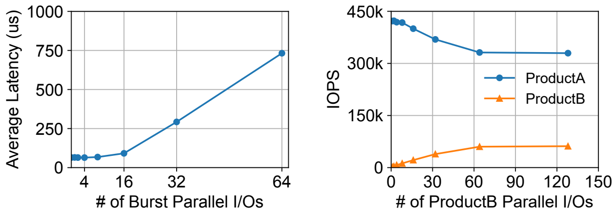


Figure 7: Distribution of I/Os from each vCPU of 4-core VMs (1st is the busiest core, 4th is the least busy core).



(a) Burst vs. base-level

(b) ProductA vs. ProductB

Figure 8: Resource competition on a control thread.

- The root causes are load imbalance among control threads and resource competition within a thread.

4 BurstCBS Overview

BurstCBS is designed and implemented as a standalone system package on xDPU (Figure 9). BurstCBS consists of three key components: high-performance queue scaling, burstable I/O scheduler, and vectorized I/O cost estimator.

High-performance queue scaling (§5.1). We rely on xDPU hardware features to balance I/O distribution among SA control threads. However, it creates an extra challenge for SA control threads to achieve high-performance I/O processing. We propose a two-tier memory pool where BurstCBS moves free buffers between the shared pool and queue-dedicated pools for efficient use of the limited memory.

Burstable I/O scheduler (§5.2). Burst capability with performance isolation requires non-uniform and dynamic resource allocation among multiple co-located tenants. We design a burst-capable I/O scheduler that periodically runs on every SA control thread for resource allocation. It allows each tenant to burst when possible while keeping performance interference among tenants within an acceptable range.

Vectorized I/O cost estimator (§5.3). The key to allocating the right amount of resources to tenants is an accurate estimation of the resource consumption of each I/O. SA manages multiple resources including CPU cycles and interconnect bandwidth. Any of these resources can become the bottleneck under various I/O patterns. We design a vectorized I/O cost estimator that decouples the estimation of each resource.

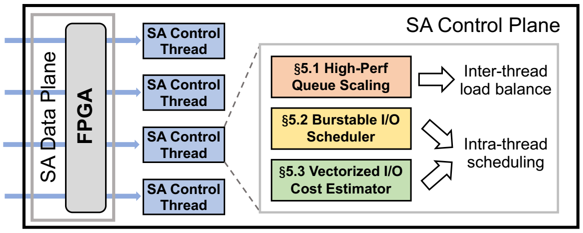
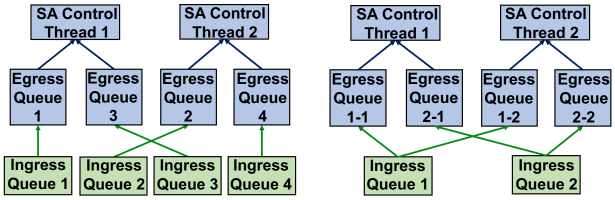


Figure 9: BurstCBS overview.



(a) M egress queues (b) $M \times N$ egress queues

Figure 10: 1 : 1 binding vs. 1 : N binding.

5 BurstCBS Design

The goal of BurstCBS is to address the two aforementioned issues: (i) lack of load balancing among control threads; (ii) lack of resource scheduling among tenants. In this section, we describe the design choices we make for each system module, as well as the considerations behind them.

5.1 High-Performance Queue Scaling

When we design the queue scaling mechanism among threads, there are two key requirements. First, load balancing must be achieved with low overhead to avoid hurting latency and throughput. Second, the mechanism should provide near-perfect load balancing to avoid further thread synchronization.

Today's DPUs are in the early stages and evolving fast. Unlike NICs supporting Receive Side Scaling (RSS) [32], there are no ASIC-based scaling solutions between host and DPU cores. Instead, DPUs provide programmable hardware on the datapath so that users can implement custom logic. For example, NVIDIA BlueField-3 consists of a set of embedded RISC-V cores named datapath accelerator (DPA) [23], and our xDPU has FPGA as the equivalent. FPGA is capable of performing lookup operations at a very high rate, which resembles a programmable switch that controls packets through match-action tables, making it an attractive candidate for offloading logic such as load balancing and rate control [33].

Evolution of load balancing on xDPU. Due to the limitation of FPGA resources on early versions of xDPU, it does not support load balancing by any means. We first make a compromise on the software side by creating one egress queue for every ingress queue, and equally assign the egress queues to all the SA control threads (Figure 10a). Assuming we have many ingress queues (one ingress queue per vCPU) and the throughput of each ingress queue only varies in a narrow range, we expect to see near-perfect load balancing.

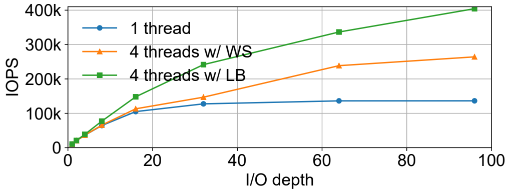


Figure 11: Throughput comparison of different approaches. WS refers to the work stealing prototype, and LB means that each thread receives an equal amount of load.

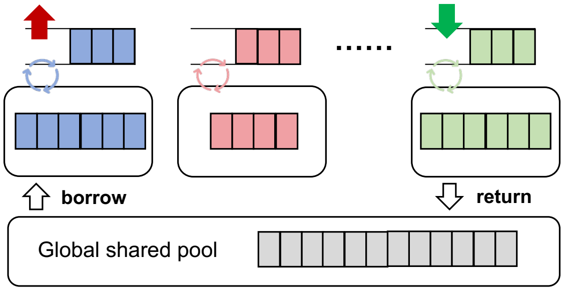


Figure 12: Two-tier memory pool.

This design works relatively well with our non-bursting CBS product types. After we launch burstable disks, we start to observe frequent load imbalance between control threads. Although we can dynamically adjust queue binding, it takes a few seconds to drain inflight I/Os and reconfigure I/O queues, making it impossible to handle transient bursts.

For early versions of xDPU, we have explored two potential software-based approaches to mitigate this issue: (i) designating a thread as a centralized dispatcher [34] and (ii) allowing idle threads to steal I/Os from others [14, 35]. However, both approaches create additional overhead that we cannot bear. They require intensive messaging between threads, which occupies a significant amount of time on wimpy cores. Figure 11 shows a 35% throughput loss if we switch to a work stealing prototype which we develop using DPDK's lockless ring buffer [36] with a reasonable level of batching.

The newest version of xDPU adds support for load balancing by allowing mapping one ingress queue to multiple egress queues (Figure 10b). We leverage this capability to realize queue scaling to multiple threads. Although no system assumptions are broken with multiple egress queues, it significantly changes how we manage DPU memory.

Two-tier memory pool for fast I/O processing. In the early years of SA development, we kept a shared pool of buffers for I/O processing because DPU memory was very limited. When an I/O arrives, I/O metadata which is required for packet header generation is written to a buffer retrieved from the memory pool. When we take the leap to support mil-

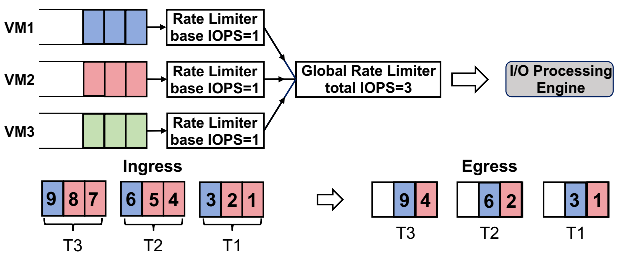


Figure 13: Lack of burst support for BaseCBS.

lions of IOPS per node, we discover that a lot of CPU cycles are wasted on writing disk/queue specific metadata into the buffers. To achieve the best I/O performance (i.e., latency and maximum IOPS), we make a design choice to let each queue maintain its own memory pool, so every memory buffer can be prefilled and will not be overwritten.

Adding extra queues makes each memory pool have even fewer buffers, which means that the number of inflight I/Os a queue can support becomes very limited. Not having enough buffers diminishes the burst maximum we can support for a disk. To remedy this loss, we move to a two-tier memory pool design (Figure 12). Each queue still keeps its own memory pool with a few dedicated buffers, but we add a global shared pool. When a queue experiences increased I/O depth, it acquires extra buffers on demand from the global pool and prefills them with its disk/queue specific metadata. It keeps the extra buffers in its own memory pool and only returns them when the burst terminates. With this design, we avoid slow I/Os caused by repetitive metadata filling to the buffers but keep memory allocation flexible enough.

5.2 Burstable I/O Scheduler

With load imbalance among threads addressed, we next focus on resource scheduling within a thread. There are two essential requirements guiding our design of the scheduler. First, a tenant should be able to use its base-level provisioning (i.e., base-level IOPS and BPS) with bounded latency no matter how other co-located tenants behave. Second, a tenant should be able to burst, but not exceed its burst provisioning. Every tenant should have an equal chance and ability to burst when they share the same burst provisioning and I/O pattern. Note that providing each tenant an equal fraction of the resources is a *non-requirement* for BurstCBS.

Base-level performance guarantee. Achieving base-level performance consists of two implications: (*i*) a tenant can achieve base-level IOPS/BPS with enough I/O parallelism; (*ii*) average read/write latency is bounded for a tenant if it is within its base-level IOPS/BPS. The latency guarantee is particularly important because it also determines the maximum throughput that an application can achieve if synchronous system calls are mostly used. In Figure 8a, we show that the latency of a victim tenant starts increasing dramatically with more parallelism of other tenants on a compute node. The reason is that a large amount of queuing delay is added when a hardware/software bottleneck is hit.

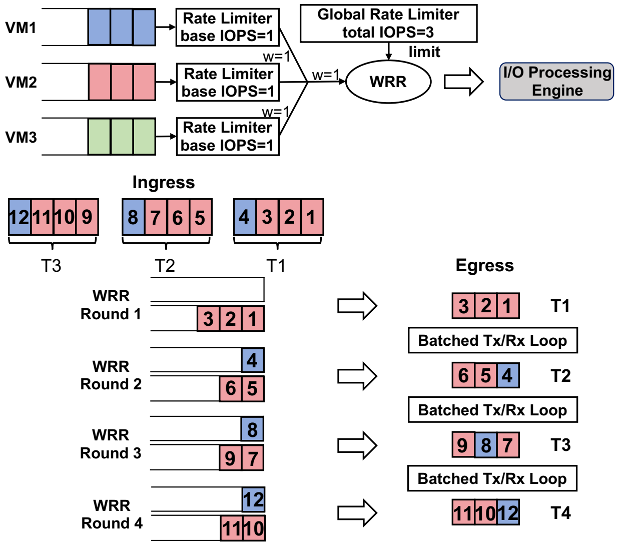


Figure 14: Queuing delay for WildCBS.

These two requirements intuitively translate to a group of rate controllers that limit the admission rate to prevent congestion, which resembles Gimbal [11]. We call this design BaseCBS, which provides performance isolation for base-level performance. There are two major differences between BaseCBS and Gimbal. First, Gimbal strives for black-box SSD congestion avoidance, so it limits the number of in-flight I/Os. On the contrary, we try to avoid congestion on DPU. When an inflight I/O is being processed by the backend servers and devices, it does not consume any resources on DPU. Therefore, BaseCBS limits the I/O admission rate instead. Second, Gimbal enforces strict fair sharing of resources among tenants to achieve absolute fairness while we need each tenant to get a share in proportion to its purchased base-level provisioning as a cloud provider.

Bounded burst support. Although BaseCBS provides a strong base-level performance guarantee, it does not provide burst support because each tenant has a static throughput limit. In Figure 13, even if VM3 is completely idle with no incoming I/O, and admitting an additional I/O will not result in congestion, VM2 is only allowed to process one I/O per window. To this end, BaseCBS is only applied to some of the legacy non-burstable CBS disks.

An easy modification that lets this design work is to assign only part of the total resource Res_{base} to BaseCBS while keeping the rest Res_{burst} in a shared pool for potential bursts. However, this design does not work out for CBS products. On the one hand, this design limits the burst capability we can provide on a compute node to Res_{burst}. On the other hand,

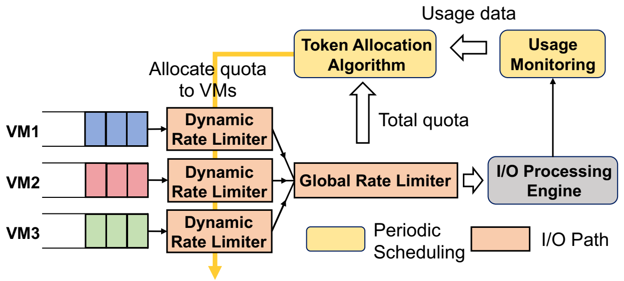


Figure 15: Burstable I/O scheduler workflow.

keeping separate and smaller resource pools directly leads to lower resource utilization, which would force us to provide CBS products at a higher price.

As Figure 4 shows, many of the tenants run way below their base-level IOPS/BPS. An ideal design is to harvest the idle base-level resources to support bursts and return the resources when they are needed. Therefore, bounded burst support essentially requires a work-conserving fair queuing scheduler with a per-tenant rate limiter which enforces provisioning limits (WildCBS). To this end, we use a weighted round-robin (WRR) scheduler, a classic work-conserving fair queuing scheduler that iterates through all ingress queues and processes requests in proportion to their weights.

Figure 14 shows an example of three tenants that are provisioned 1 base IOPS and 3 burst IOPS. 1 base IOPS per tenant is achieved by having a global rate limiter of 3 IOPS and each tenant is weighted equally in WRR scheduling. In this example, VM1 runs at base-level usage, VM2 tries to burst to 3 IOPS, and VM3 is completely idle. Due to the work-conserving nature of WRR, WildCBS is able to fully utilize idle resources from VM3, and the base-level IOPS of VM1 is still guaranteed. WildCBS is integrated into the existing version of SA to enable burst capability in production.

However, a side effect of WildCBS is that a VM that runs at or below base-level usage observes a significantly higher latency while the bursting VM is barely impacted. We observe that if some VMs dispatch I/Os at a much higher rate than others, they can quickly consume all the resource budget, leaving the rest of the VMs to wait until the next window. In Figure 14, VM2 causes all I/Os from a base-level tenant VM1 to delay by a time window. It becomes very common when the bursting VMs employ an extremely high I/O parallelism.

Burstable I/O scheduler. To remedy inadequacies of BaseCBS and WildCBS, we leverage resource usage history to instruct dynamic rate limiting. Figure 15 shows the design of our burstable I/O scheduler (BIOS). BIOS actively collects usage data and allocates resources in proportion to user demands. It provides strong protection on base-level performance by (i) enforcing the total resource allocation limit and (ii) resuming base-level provisioning as soon as it discovers insufficient resource allocation to under-utilizing tenants.

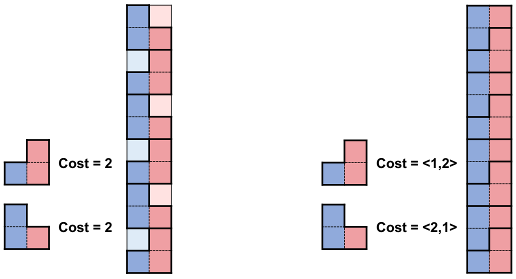
Algorithm 1 BIOS algorithm

```
 1: procedure RUN_SCHEDULING( )
 2:     $unused = total\_alloc - reserved$
 3:     for all tenants $i = 1..n$ do
 4:         if $status_i$ = burst or $throttle_i > 0$ then
 5:             $alloc_i \leftarrow res\_base_i$
 6:         else
 7:             $alloc_i \leftarrow alloc\_hist_i \times \alpha$
 8:         $unused \leftarrow unused - alloc_i$
 9:     for all tenants $i = 1..n$ do
10:         $w_i = \frac{usage_i + throttle_i \times weight\_throttle}{\sum usage_i + \sum throttle_i \times weight\_throttle}$
11:         $alloc_i \leftarrow min(alloc_i + unused \times w_i, res\_burst_i)$
12:         if $alloc_i > res\_base_i$ then
13:             burst_tenants.append(tenant$_i$)
14: function THROTTLE_IO(tenant$_k$, io)
15:     if $alloc_k \geq$cost(io) then
16:         $alloc_k \leftarrow alloc_k -$cost(io)
17:     else
18:         tenant$_b$ ←power_of_two_choices(burst_tenants)
19:         if $alloc_b \geq$cost(io) then
20:             $alloc_b \leftarrow alloc_b -$cost(io)
21:         else if reserved≥cost(io) then
22:             reserved←reserved−cost(io)
23:         else
24:             return true
25:     return false
```

Algorithm 1 depicts how resources are allocated. The algorithm first allocates base-level provisioning to all tenants unless they were below base-level and did not consume all resources in the last window (lines 4-5). Otherwise, the algorithm tries to lower their allocated resources to allow others to burst (line 7). In the second round of allocation, it allocates the remaining resources in proportion to the weighted sum of consumed and throttled I/Os in the last window, but it cannot exceed burst provisioning (lines 10-11). We add a higher weight to throttled I/Os because they may have more follow-up I/Os. The running time of this algorithm only increases linearly with the number of tenants. This is important because (i) there is not enough headroom on wimpy cores for a complex algorithm, and (ii) we may have hundreds of tenants running on the same compute node in extreme cases.

Fast base-level performance recovery. A scheduling algorithm that relies on historical statistics needs to tolerate bad predictions. The consequence of harvesting idle base-level resources is that if a tenant suddenly starts dispatching I/Os after being idle for a long time, it may observe high queuing delay and inadequate IOPS, which breaks our commitment on base-level performance guarantee.

A fast recovery mechanism is added as compensation before the algorithm catches its mis-prediction in the next scheduling cycle. As shown in Algorithm 1 lines 18-20, a base-level tenant that runs out of resources first tries to re-



(a) Scalar cost

(b) Vectorized cost

Figure 16: Scalar cost vs. vectorized cost.

claim the extra resources from bursting tenants. Ideally, we should reclaim resources from the tenant with the most remaining resources, which involves sorting all bursting tenants. However, applying sorting on a per-I/O basis hurts I/O performance significantly. We adopt power of two choices as an alternative to eliminate sorting on the I/O processing path.

This mechanism could fail to take effect if bursting tenants quickly consume all the allocated resources, leaving nothing left for a base-level tenant to reclaim. Therefore, we add another layer of protection of a shared resource pool (Algorithm 1 lines 21-22). The intuition behind this optimization is that, through our online measurements, it is unlikely multiple base-level tenants would resume their usage at the same time. So we reserve a small amount of pooled resources that are just enough for two tenants to resume base-level provisioning. It diminishes a little of burst maximum we can support, but greatly helps us guarantee base-level performance.

5.3 Vectorized I/O Cost Estimator

BIOS is a proper framework for serving a mixture of base-level and burst tenants. A key assumption of BIOS is that consumed resource per I/O is known. However, IOPS and BPS that an SA control thread on DPU can saturate are dynamic subject to hardware specification, software implementation, system configuration, and I/O pattern. To avoid overloading SA, it is necessary to estimate the I/O cost accurately. Previous storage systems focus on I/O cost estimation of SSDs [10, 11, 17]. Because manufacturers of SSDs reveal limited design details of their products, existing work estimates SSD I/O cost by profiling each device with synthetic workloads.

For CBS, the bottlenecks are on xDPU and SA, rather than SSDs. Based on our online measurement, we identify three major bottlenecks in xDPU and SA. (*i*) CPU: it takes a few microseconds to process an I/O on an SA control thread on average, so a control thread can handle a few hundreds of thousands of IOs per second at maximum. (*ii*) Interconnect: SA is able to use 100Gbps NIC bandwidth and ∼100Gbps DMA bandwidth. (*iii*) Software: the read/write bandwidth of SA is also constrained by a software rate limiter which protects other non-storage services from resource starvation.

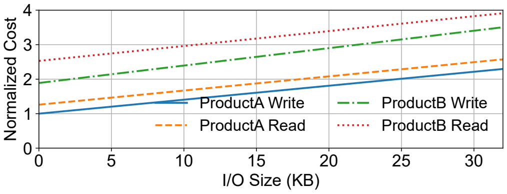


Figure 17: Normalized CPU cost estimation of four I/O types.

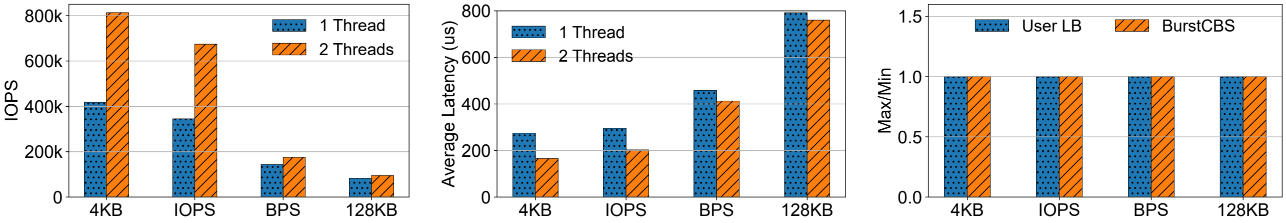
Table 3: Effectiveness of estimation adjustment.

	WildCBS	BurstCBS w/o adjustment	BurstCBS with adjustment
Read Lat (us)	1096.89	848.57	248.97
Write Lat (us)	868.79	749.57	270.49

Furthermore, the relative ratios across consumed resources are divergent on a per I/O basis. Our experiments show that a 4KB write I/O consumes 8× higher CPU time per byte compared to a 128KB write I/O, while they always consume the same egress bandwidth per byte. Therefore, the bottlenecks are CPU and egress bandwidth for 4KB and 128KB write I/Os respectively. This result implies that it is necessary to model the cost of different I/O types independently.

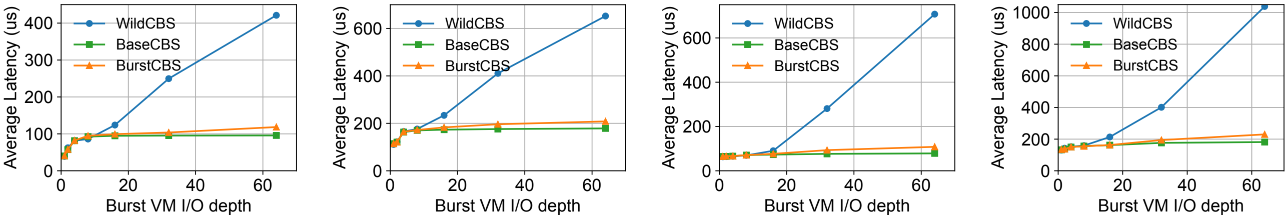
If we describe I/O cost as a scalar, it creates resource underutilization. In Figure 16, a scalar cost must be given the value of the most consumed resource, which unnecessarily leaves other resources idle, limiting our ability to burst. In contrast, if we decouple the costs of different resource types, higher resource utilization can be achieved without breaching the latency target. Therefore, we describe the cost of an I/O as a vector of 4 dimensions: *CPU time*, *ingress*, *egress*, and *software limit*. Ingress, egress, and software limits are shared among all threads. We simply divide the global limit by the number of threads to get the per-thread limit. Out of them, only CPU time requires profiling. We derive the CPU time of (product_type, rw, size) tuple from the maximum IOPS it can achieve. We only profile I/O sizes from 4KB to 16KB, because the bottleneck is no longer on CPU beyond 16KB. We fit the observed values into a linear model so that we can estimate CPU time for all sizes. Figure 17 shows the normalized estimation of different I/O types on the newest xDPU. ProductA and ProductB are both burstable CBS product classes. Compared to ProductB, ProductA is further optimized for higher throughput and lower latency by adopting advanced hardware features and optimized software implementation.

Unpredictable misestimation handling. Although vectorized cost estimator accurately reflects I/O cost in common cases, there are many circumstances that we cannot foresee and integrate into our cost model in advance. A typical misestimation happens when SA takes a different path for I/O processing when the FPGA of xDPU experiences transient hardware failures. During uncommon failures, SA enters a



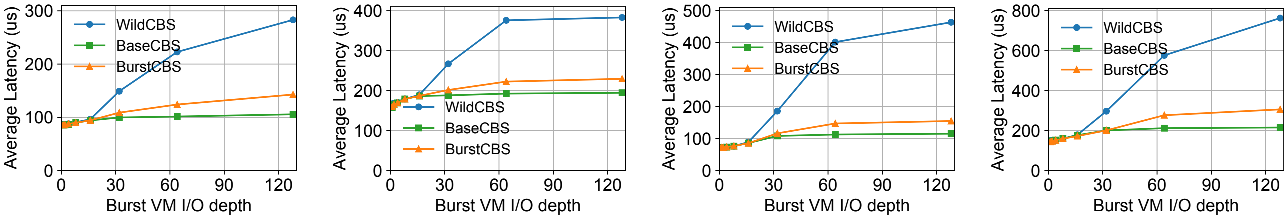
(a) Throughput (b) Average lantecy (c) Uniformity

Figure 18: Effectiveness and overhead of FPGA-based load balancing with high-performance queue scaling.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 19: I/O latency (I/O depth=1) with a BPS-intensive burst VM in the background.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 20: I/O latency (I/O depth=1) with an IOPS-intensive burst VM in the background.

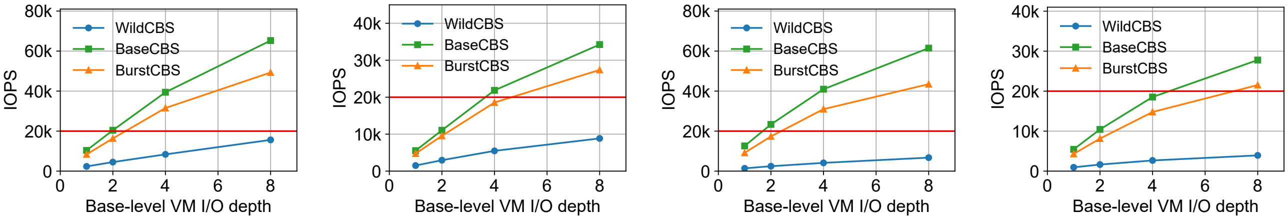
heavy error handling branch which retries failed I/Os multiple times until they reach the configured timeout and generates error logs for further investigation. Doing so at a per I/O basis consumes more CPU resources and bandwidth than a normal code path usually does. Based on our observation, the amortized cost per successful I/O can be doubled when SA experiences such unexpected failures.

To alleviate this impact, we introduce a delay-based cost adjustment mechanism. BurstCBS is anchored to a target delay. When the target delay is breached, we gradually increase the cost. And we reduce the cost when the delay drops below the target. Note that backend time is excluded from this delay. The reason is that SSD is notorious for its high tail latency [37–39], which may wrongly trigger the cost adjustment mechanism. To allow the cost adjustment mechanism to react quickly with only a few data points, irrelevant outliers should be avoided. In Table 3, we show a case that SA keeps detecting FPGA failures. BurstCBS without the cost adjustment mechanism mis-estimates I/O cost and admits more I/Os than its capacity, which results in high latency on the base-level tenant. Adding cost estimation adjustment significantly reduces the latency by admitting the right amount of I/Os.

6 Evaluation

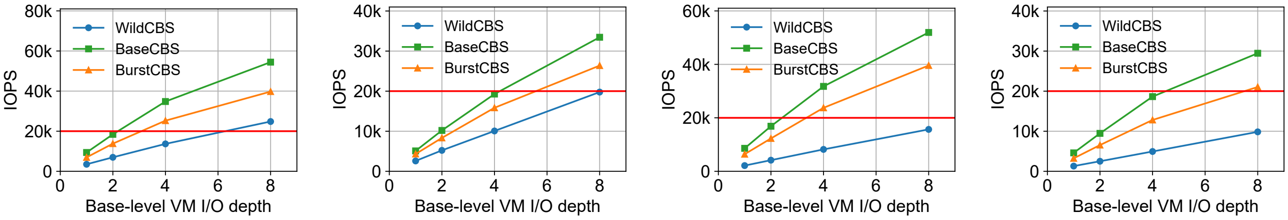
In this section, we evaluate the performance of BurstCBS. Our main baseline is WildCBS which combines WRR scheduling and per-tenant rate limiters. Currently, WildCBS is the most widely deployed version in our production clusters. We also compare BurstCBS with BaseCBS which is a variant of Gimbal and provides strong performance isolation between tenants. All of the experiments are conducted on a compute node with the newest version of xDPU.

Our experiments run two different FIO [40] workloads which we typically use at Alibaba Cloud to make sure our products can adapt to different usage patterns. One is IOPS-intensive and contains a mix of 4KB-16KB I/Os. This workload consists of small I/Os which resembles the I/O pattern of many transactional databases. The other is BPS-intensive



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 21: IOPS with a BPS-intensive burst VM (I/O depth=64) in the background.



(a) ProductA write (b) ProductA read (c) ProductB write (d) ProductB read

Figure 22: IOPS with a IOPS-intensive burst VM (I/O depth=128) in the background.

and contains a mix of 4KB-128KB I/Os. Most of the I/Os are within this range in our production environment. We include both ProductA and ProductB I/Os in each workload.

We mainly evaluate three aspects of BurstCBS: thread load balancing with high-performance queue scaling (§6.1), latency (§6.2) and IOPS (§6.3) of base-level tenants with bursting neighbors, and overall resource utilization at burst (§6.4). We also evaluate effectiveness of the fast base-level throughput recovery mechanism (§6.5) and the scalability of BIOS (§6.6). At last, we perform a set of database experiments to evaluate how BurstCBS performs under real use cases (§6.7), and collect results from a node that serves an internal database service (§6.8).

6.1 Inter-thread Load Balancing

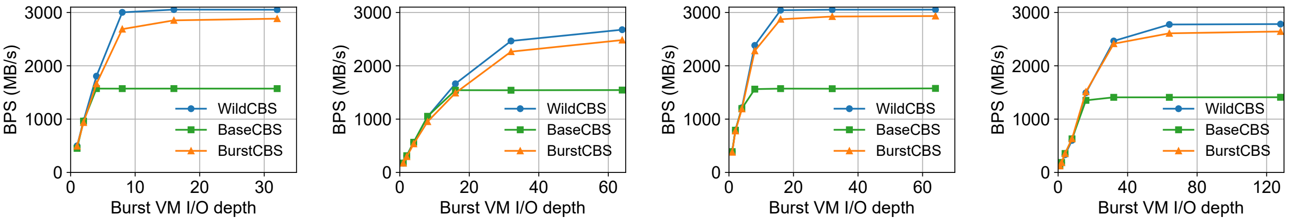
We first evaluate inter-thread load balancing to understand the effectiveness and overhead of FPGA-based load balancing with high-performance queue scaling. Figure 18a shows the maximum write IOPS that a tenant can achieve with one or two SA control threads behind the FPGA load balancer. We observe near-linear scaling on pure 4KB workload and IOPS-intensive workload because the bottleneck is on the SA control threads which run on CPU cores when a large amount of small I/Os are being processed. Scaling on BPS-intensive workload and pure 128KB workload is limited because the NIC is already congested while the SA control threads are not fully occupied. In Figure 18b, we repeat the experiment with I/O depth unchanged and compare average latency, from which we can draw a similar conclusion.

To show how well the load is balanced among multiple threads, in Figure 18c, we compare BurstCBS with the case that a user intentionally dispatches I/Os equally to each ingress queue (by using multiple vCPUs which are mapped to different queues) through FIO. We start six SA control threads in total and take the ratio of maximum to minimum values of throughput across threads to reflect uniformity. The results demonstrate that they are almost equivalent in terms of inter-thread load balancing.

6.2 Base-level Tenant Latency

A main goal of BurstCBS is to keep average latency under SLO for tenants running below their base-level IOPS/BPS. In this experiment, we run an I/O stream with different I/O depths from a background VM. While the background VM is running, we start a 4KB I/O stream of depth 1 on the victim VM and observe its average latency. In Figure 19, we use a read/write BPS-intensive I/O workload in the background VM with I/O depths from 1 to 64, and we record the average latency of four types of I/Os. We evaluate both read and write I/Os from two burstable CBS product classes, ProductA and ProductB. With the current implementation, ProductA is faster than ProductB, and write is faster than read. Overall, average latency is reduced by 68%–85% compared to WildCBS, and is very close to BaseCBS which shows the ideal latency we can possibly achieve with strong isolation.

Figure 20 shows the results on the IOPS-intensive workload in the background. The latency reduction ranges from 40% to 66% which is slightly less than that of the BPS-intensive workload. This difference is because the IOPS-intensive work-



(a) BPS-intensive, ProductA (b) IOPS-intensive, ProductA (c) BPS-intensive, ProductA+B (d) IOPS-intensive, ProductA+B

Figure 23: Overall resource utilization during a burst.

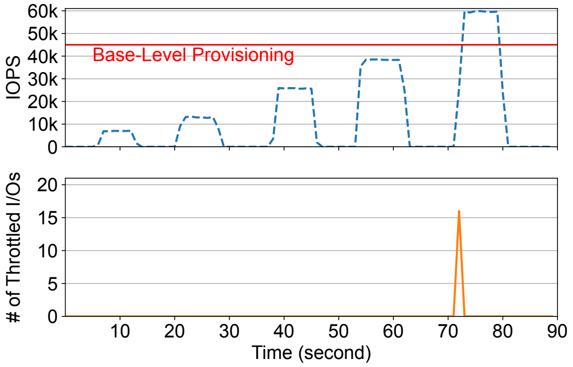


Figure 24: Responsiveness to sudden tenant activation.

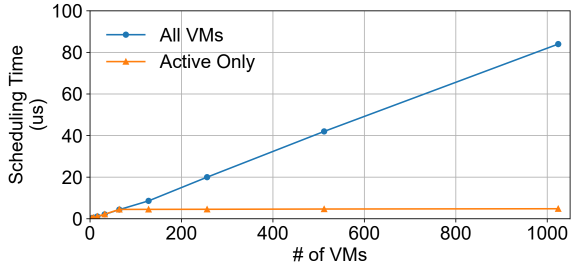


Figure 25: Scheduler scalability with number of VMs.

load shifts bottleneck to the CPU cores. When the CPU cores are congested, the performance gap between BaseCBS and WildCBS is not as significant.

6.3 Base-level Tenant Throughput

Another critical requirement for BurstCBS is that a tenant should be able to reach its base-level IOPS with a relatively small I/O depth. In this experiment, we set the base-level IOPS of the victim VM to 20k, and we expect it to reach that IOPS on a 4KB I/O stream within I/O depth 8. We again validate the effectiveness of BurstCBS on both BPS-intensive workload and IOPS-intensive workload with all four different I/O types. Figure 21 and Figure 22 show that BurstCBS can achieve the desired IOPS for all the cases, while WildCBS at I/O depth 8 fails to meet our goal for seven out of eight cases, and the IOPS is as low as 4,000. Similar to §6.2, there is a smaller performance gap between BaseCBS and WildCBS on the IOPS-intensive workload, which leaves limited space for us to optimize. And ProductB read I/O is the most costly operation out of the four I/O types, so BurstCBS barely reaches 20k IOPS in Figure 22d, which is the lowest.

6.4 Burst Resource Utilization

Although protecting base-level performance is our first priority, we also seek for high resource utilization during bursts. In this experiment, the available bandwidth is limited to 3GB/s, which is the maximum that an SA control thread can handle. We start two VMs and let one VM burst as much as possible while keeping the other VM running at I/O depth 2 which produces an I/O stream below its base-level provisioning. In Figure 23, the results demonstrate that BurstCBS loses about 5%–8% throughput compared to WildCBS, which meets our expectation because we keep 5% of the total resources in the shared pool for fast recovery. BaseCBS enforces fair resource distribution at all times, which limits the resources that the burst VM can use to half of the limit. Note that the maximum bandwidth cannot be achieved with the IOPS-intensive workload due to the extra CPU overhead incurred by small I/Os. And mixing ProductA and ProductB I/Os slightly improves resource utilization because ProductA and ProductB use different polling loops and idle loops waste CPU cycles.

6.5 Responsiveness to Sudden Activation

We next evaluate the fast base-level provisioning recovery mechanism. A tenant should be able to recover its base-level provisioning seamlessly even when the resources are lent to other tenants. In this experiment, we create a VM with base-level provisioning of 45k IOPS. We run I/O streams of depths 1-16 on this VM. Before we run each stream, we let the VM stay idle for a few seconds to make sure that its resource allocation drops to zero. When we run an I/O stream, the metric we record is the number of throttled I/Os. We expect to see no throttled I/Os when an I/O stream is within the base-level provisioning. In Figure 24, we do not observe any

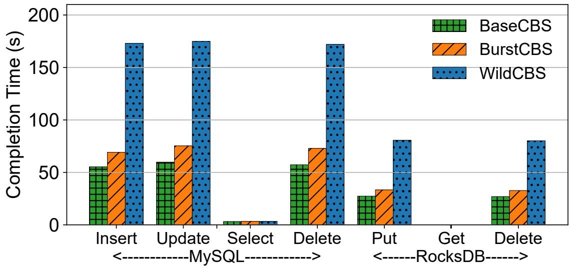


Figure 26: Completion time of 100k DB operations.

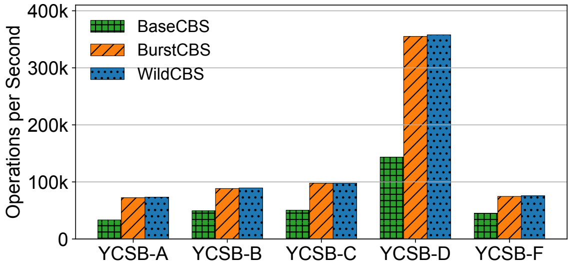


Figure 27: YCSB throughput.

throttled I/Os until we kick off a 60k IOPS stream which already makes the VM a burst tenant.

6.6 Scalability

Scalability of the scheduler is also our concern because if the periodic scheduler runs for too long, successive I/Os will experience high latency and it will also hurt overall throughput by occupying too many CPU cycles. In this experiment, we solely run the scheduler on xDPU and vary the number of VMs from 4 to 1024. 1024 is the theoretical maximum number of VMs on a compute node in any near future. In Figure 25, the scheduling time increases linearly with the number of VMs and is always below 100μs. We introduce an additional optimization that removes a VM from scheduling after it becomes idle for a while. We run the experiment again with 64 active VMs at maximum. It takes less than 5μs to run the scheduler once no matter how many VMs are there in total.

6.7 Application Performance

Transactional databases are one of the targeted use cases of burstable CBS. We evaluate both a SQL database (MySQL [41]) and a NoSQL database (RocksDB [42]) to validate its improvement on real-world use cases. Following our production specification, we provision 16 I/O-burstable VMs with 100k base-level IOPS and 200k burst IOPS on a compute node. The corresponding base-level BPS and burst BPS we provision are 1400MB/s and 2800MB/s respectively. Each VM has 8 vCPUs, 16GB memory, a 40GB ProductB virtual disk as the OS disk, and another 1TB ProductB virtual disk as the database data disk.

We first evaluate the latency of DB operations. For the MySQL experiment, we install MySQL 8.0 on one of the VMs. We develop a simple C program that connects to the MySQL database, executes each type of operation 100k times, and records the execution time. For RocksDB, we write a C++ program and again run put/get/delete 100k times with the sync flag enabled to force an immediate disk I/O per write request. When we execute the programs, we let the other 15 VMs burst by starting a BPS-intensive workload of I/O depth 32 on each of them. In Figure 26, for all write operations (insert, update, put, and delete), we observe about 60% latency reduction. Databases extensively use cache for read operations (select and get). Therefore, we do not observe significant improvement on read operations in this experiment.

We next evaluate the performance improvement brought by burst capability. In this experiment, we focus on YCSB [43] over RocksDB which creates higher throughput than MySQL with the same compute power. We leave the other 15 VMs idle and set up eight RocksDB instances to better leverage burst capability. For each DB instance, we create 10 million 1KB entries and execute 1 million operations. As shown in Figure 27, 2× burst on BurstCBS results in 1.7×-2.5× throughput improvement on various workloads over BaseCBS, and the results are close to those of WildCBS.

6.8 Practical Benefits

We last confirm that BurstCBS improves our database user experience. Here, we deploy BurstCBS to a production compute node that serves the internal Relational Database Service (RDS) and evaluate performance. RDS creates VM instances with CBS disks and manages databases on the VMs for users. A typical RDS VM instance comes with two data disks: one ProductB disk as the main storage, and one ProductA disk as a buffer pool extension. Previously, RDS has noticed neighbor interference and informed us.

In this experiment, an RDS VM instance is started and preloaded with 100 tables of 10 million rows. Another VM is used to connect to the RDS instance through a virtual private network and run single-threaded sysbench [44]. In Figure 28, 8 or 16 burst tenants of I/O depth 32 run in the background. The distribution of I/O sizes and the read/write ratio follow the same pattern in production [7]. The results show that average query latency is reduced by up to 83%. Note that RDS read operations can also trigger disk writes because the ProductA disk is used as a buffer pool extension.

This experiment is also run against a more fluctuant trace for 30 minutes. A 30-minute second-scale monitoring history of a burst disk in production is amplified to the scale of 10GB/s to generate the I/O trace. The trace is replayed in the background, and sysbench is used to record average query latency. In Figure 29, while WildCBS creates latency spikes of 20-50ms, BurstCBS is able to keep it under 10ms.

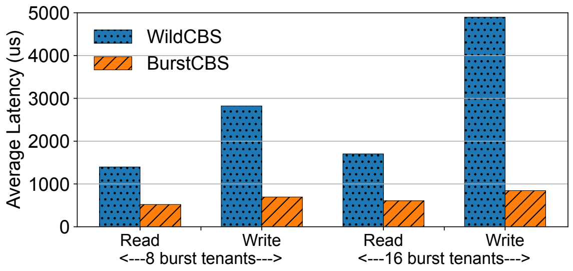


Figure 28: RDS query latency with burst neighbors.

7 Discussion

We discuss a number of future improvements on our roadmap that will further enhance BurstCBS.

Co-optimization with user OS kernel. Rate control at SA requires a significant amount of DPU memory to buffer the requests and may eventually cause out-of-memory. Limiting I/O rate on the user side can mitigate this issue. Furthermore, newer Linux kernel supports NVMe WRR [45], a feature that allows a user to prioritize certain I/Os, which helps protect the performance of critical I/Os when SA is congested.

Automated cost profiling. We currently maintain three different versions of xDPU and many more system configurations in our production environment. It brings a heavy operational burden if we need to manually profile I/O cost every time we make a software/configuration update. Instead, we are developing an automated I/O cost profiler which profiles I/O cost offline at system bootstrap and adjusts cost adaptively online.

Inter-server scheduling. The best way to handle congestion is always to avoid it in the first place. Once we detect multiple co-located VMs often burst at the same time, we can signal the control plane of VM instances to migrate them when possible. Because the time to migrate a VM ranges from a few seconds to several minutes and it causes temporary unavailability, we still rely on BurstCBS to handle short-term congestion.

8 Related Work

Cloud storage systems. There is a large body of work on cloud storage systems [5–9, 16, 17, 46–52]. Despite the different interfaces these systems expose (e.g., block store, object store), most of them are distributed systems to meet the scale of cloud. Existing research mainly focuses on the backend system design. Tectonic [51] and Pangu [52] provide unified backend storage for a large number of tenants and different use cases. To overcome the inefficiency of traditional in-kernel network stacks, RDMA [5, 6, 48] and other kernel bypassing network stacks [7, 8] are deployed. BurstCBS instead focuses on burst capability support of cloud block storage. We reveal and tackle the major challenges to achieve both extreme burst and base-level performance protection.

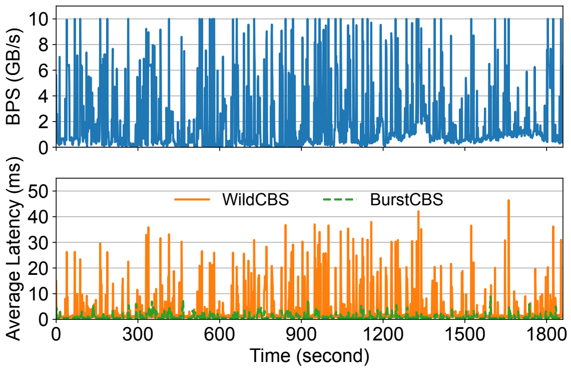


Figure 29: RDS query latency with replayed production background traffic.

Resource sharing for storage systems. Previous work has explored how to achieve work-conserving scheduling and/or fair sharing for storage systems [11,17,53–62]. Many of them also involve estimating per I/O cost for efficient resource scheduling. However, previous work mainly targets the bottleneck of storage media (e.g., HDD and SSD), while the bottleneck we encounter is on DPU. A highly related system is Gimbal [11] which designs a fair queuing scheduler for performance isolation on the server side of disaggregated storage with DPUs. BurstCBS distinguishes itself by supporting dynamic bursts and addressing unique challenges (i.e., load imbalance and cost estimation) on client-side DPUs.

9 Conclusion

This paper presents BurstCBS, a hardware-software co-designed storage I/O scheduling system that achieves inter-thread load balancing and intra-thread resource scheduling. BurstCBS applies three techniques: a high-performance queue scaling mechanism, a burstable I/O scheduler, and a vectorized I/O cost estimator. We implement and evaluate BurstCBS on xDPU-based servers. We show that BurstCBS provides base-level performance protection while allowing tenants to burst as much as possible.

Acknowledgments. We thank our shepherd Geoffrey M. Voelker and the anonymous reviewers for their valuable feedback. We also thank Erci Xu for his precious comments. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, National Natural Science Foundation of China under the grant numbers 62172008 and 62325201, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and Alibaba Cloud through Alibaba Research Intern Program. Xin Jin is the corresponding author. Junyi Shu, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "Amazon EBS features." https://aws.amazon.com/ebs/features/, 2023. Retrieved Nov 15, 2023.

[2] "Azure managed disk types." https://learn.microsoft.com/en-us/azure/virtual-machines/disks-types/, 2023. Retrieved Nov 15, 2023.

[3] "Google storage options." https://cloud.google.com/compute/docs/disks, 2023. Retrieved Nov 15, 2023.

[4] "Alibaba Cloud ECS Limits." https://www.alibabacloud.com/help/en/ecs/product-overview/limits, 2023. Retrieved Nov 15, 2023.

[5] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill, "Empowering Azure Storage with RDMA," in *USENIX NSDI*, 2023.

[6] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu, "When Cloud Storage Meets RDMA," in *USENIX NSDI*, 2021.

[7] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, R. Liu, C. Shi, B. Fu, J. Zhu, J. Wu, D. Cai, and H. H. Liu, "From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud," in *ACM SIGCOMM*, 2022.

[8] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, Z. Yang, Z. Chen, W. Lin, Y. Hou, R. Liu, C. Shi, J. Zhu, and J. Wu, "Deploying User-space TCP at Cloud Scale with LUNA," in *USENIX ATC*, 2023.

[9] W. Zhang, E. Xu, Q. Wang, X. Zhang, Y. Gu, Z. Lu, T. Ouyang, G. Dai, W. Peng, Z. Xu, S. Zhang, D. Wu, Y. Peng, T. Wang, H. Zhang, J. Wang, W. Yan, Y. Dong, W. Yao, Z. Wu, L. Zhu, C. Shi, Y. Wang, R. Liu, J. Wu, J. Zhu, and J. Wu, "What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store," in *USENIX FAST*, 2024.

[10] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash ≈ Local Flash," in *ACM ASPLOS*, 2017.

[11] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, "Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs," in *ACM SIGCOMM*, 2021.

[12] J. Shu, R. Zhu, Y. Ma, G. Huang, H. Mei, X. Liu, and X. Jin, "Disaggregated RAID Storage in Modern Datacenters," in *ACM ASPLOS*, 2023.

[13] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, "TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10," in *USENIX NSDI*, 2020.

[14] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting Linux Storage Stack for µs Latency and High Throughput," in *USENIX OSDI*, 2021.

[15] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash Storage Disaggregation," in *EuroSys*, 2016.

[16] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building An Elastic Query Engine on Disaggregated Storage," in *USENIX NSDI*, 2020.

[17] T. Heo, D. Schatzberg, A. Newell, S. Liu, S. Dhakshinamurthy, I. Narayanan, J. Bacik, C. Mason, C. Tang, and D. Skarlatos, "IOCost: Block IO Control for Containers in Datacenters," in *ACM ASPLOS*, 2022.

[18] "Understanding Burst vs. Baseline Performance with Amazon RDS and GP2." https://aws.amazon.com/blogs/database/understanding-burst-vs-baseline-performance-with-amazon-rds-and-gp2/, 2023. Retrieved Nov 15, 2023.

[19] "Azure managed disk bursting." https://learn.microsoft.com/en-us/azure/virtual-machines/disk-bursting, 2023. Retrieved Nov 15, 2023.

[20] "Alibaba Cloud block storage performance." https://www.alibabacloud.com/help/en/ecs/user-guide/block-storage-performance, 2023. Retrieved Nov 15, 2023.

[21] "AWS Nitro System." https://aws.amazon.com/ec2/nitro/, 2023. Retrieved Nov 15, 2023.

[22] "A Detailed Explanation about Alibaba Cloud CIPU." https://www.alibabacloud.com/blog/a-detailed-explanation-about-alibaba-cloud-cipu_599183, 2023. Retrieved Nov 15, 2023.

[23] "Nvidia BlueField-3." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2023. Retrieved Nov 15, 2023.

[24] "Intel Infrastructure Processing Unit." https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2023. Retrieved Nov 15, 2023.

[25] "Marvell LiquidIO 3." https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2023. Retrieved Nov 15, 2023.

[26] "AWS burstable performance instances." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html, 2023. Retrieved Nov 15, 2023.

[27] "Azure B-series burstable virtual machine sizes." https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable, 2023. Retrieved Nov 15, 2023.

[28] "Google Shared-core VMs." https://cloud.google.com/compute/docs/general-purpose-machines#sharedcore, 2023. Retrieved Nov 15, 2023.

[29] "Alibaba Cloud Burstable Instances." https://www.alibabacloud.com/help/en/ecs/user-guide/overview-5, 2023. Retrieved Nov 15, 2023.

[30] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *ACM ASPLOS*, 2023.

[31] "DB-Engines Ranking." https://db-engines.com/en/ranking, 2023. Retrieved Nov 15, 2023.

[32] "Introduction to Receive Side Scaling." https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2022. Retrieved Nov 15, 2023.

[33] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing," in *USENIX NSDI*, 2022.

[34] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency," in *USENIX NSDI*, 2019.

[35] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks," in *ACM SOSP*, 2017.

[36] "DPDK Ring Library." https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2023. Retrieved Nov 15, 2023.

[37] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi, "IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage," in *ACM SOSP*, 2021.

[38] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," *ACM Transactions on Storage*, 2017.

[39] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments," in *USENIX FAST*, 2016.

[40] "Flexible I/O Tester." https://github.com/axboe/fio, 2023. Retrieved Nov 15, 2023.

[41] "MySQL." https://www.mysql.com/, 2023. Retrieved Nov 15, 2023.

[42] "RocksDB." https://rocksdb.org/, 2023. Retrieved Nov 15, 2023.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *ACM Symposium on Cloud Computing*, 2010.

[44] "sysbench." https://github.com/akopytov/sysbench, 2024. Retrieved Apr 15, 2024.

[45] K. Joshi, K. Yadav, and P. Choudhary, "Enabling NVMe WRR support in Linux Block Layer," in *USENIX HotStorage Workshop*, 2017.

[46] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li, and J. Wu, "Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems," in *USENIX FAST*, 2023.

[47] S. Zhou, E. Xu, H. Wu, Y. Du, J. Cui, W. Fu, C. Liu, Y. Wang, W. Wang, S. Sun, X. Wang, B. Feng, B. Zhu, X. Tong, W. Kong, L. Liu, Z. Wu, J. Wu, Q. Luo, and J. Wu, "SMRSTORE: A Storage Engine for Cloud Object Storage on HM-SMR Drives," in *USENIX FAST*, 2023.

[48] Q. Li, Y. Gao, X. Wang, H. Qiu, Y. Le, D. Liu, Q. Xiang, F. Feng, P. Zhang, B. Li, J. Dong, L. Tang, H. H. Liu, S. Liu, W. Li, R. Miao, Y. Wu, Z. Wu, C. Han, L. Yan, Z. Cao, Z. Wu, C. Tian, G. Chen, D. Cai, J. Wu, J. Zhu, J. Wu, and J. Shu, "Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs," in *USENIX OSDI*, 2023.

[49] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3," in *ACM SOSP*, 2021.

[50] Q. Li, L. Chen, X. Wang, S. Huang, Q. Xiang, Y. Dong, W. Yao, M. Huang, P. Yang, S. Liu, Z. Zhu, H. Wang, H. Qiu, D. Liu, S. Liu, Y. Zhou, Y. Wu, Z. Wu, S. Gao, C. Han, Z. Luo, Y. Shao, G. Tian, Z. Wu, Z. Cao, J. Wu, J. Shu, J. Wu, and J. Wu, "Fisc: A Large-scale Cloud-native-oriented File System," in *USENIX FAST*, 2023.

[51] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, S. S. P, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. Tipton, E. Katz-Bassett, and W. Lloyd, "Facebook's Tectonic Filesystem: Efficiency from Exascale," in *USENIX FAST*, 2021.

[52] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu, "More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba," in *USENIX FAST*, 2023.

[53] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *USENIX FAST*, 2009.

[54] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-Queue Fair Queuing," in *USENIX ATC*, 2019.

[55] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu, "VFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation," in *ACM Symposium on Cloud Computing*, 2015.

[56] M. Nanavati, J. Wires, and A. Warfield, "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage," in *USENIX NSDI*, 2017.

[57] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *USENIX FAST*, 2012.

[58] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *USENIX ATC*, 2013.

[59] D. Shue and M. J. Freedman, "From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra," in *EuroSys*, 2014.

[60] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *ACM SOSP*, 2013.

[61] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *USENIX FAST*, 2007.

[62] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs," in *USENIX OSDI*, 2023.

Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory

Ming Zhang, Yu Hua*, Zhijun Yang
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

In modern datacenters, memory disaggregation unpacks monolithic servers to build network-connected distributed compute and memory pools to improve resource utilization and deliver high performance. The compute pool leverages distributed transactions to access remote data in the memory pool to provide atomicity and strong consistency. Existing single-versioning designs have been constrained due to limited system concurrency and high logging overheads. Although the multi-versioning design in the conventional monolithic servers is promising to offer high concurrency and reduce logging overheads, which however fails to work in the disaggregated memory. In order to bridge the gap between the multi-versioning design and the disaggregated memory, we propose Motor that holistically redesigns the version structure and transaction protocol to enable **m**ulti-versi**o**ning for fast distributed **t**ransacti**o**n processing on the disagg**r**egated memory. To efficiently organize different versions of data in the memory pool, Motor leverages a new consecutive version tuple (CVT) structure to store the versions together in a continuous manner, which allows the compute pool to obtain the target version in a single network round trip. On top of CVT, Motor leverages a fully one-sided RDMA-based MVCC protocol to support fast distributed transactions with flexible isolation levels. Experimental results demonstrate that Motor improves the throughput by up to 98.1% and reduces the latency by up to 55.8% compared with state-of-the-art systems.

1 Introduction

Memory disaggregation in modern datacenters receives extensive attentions [2, 3, 35, 46, 53, 62]. Specifically, memory disaggregation decouples the compute and memory resources from traditional monolithic servers to build independent and scalable compute and memory pools. These pools are connected via fast network (e.g., RDMA [75] or CXL [7]). A compute pool contains many powerful compute units to run tasks and small DRAM-based memory to maintain metadata. Moreover, a memory pool consists of substantial memory modules to store application data and a small number of weak compute units only for memory allocations and network interconnections [84, 86]. With the aid of efficient resource pooling, memory disaggregation significantly improves the resource utilization, elasticity, and failure isolation [65, 72].

To provide atomicity and strong consistency guarantees for applications on the disaggregated memory, the compute pool leverages distributed transactions to access remote data in the memory pool. A recent design, i.e., FORD [84], is able to run distributed transactions on the disaggregated memory. To simplify the data store in the memory pool, FORD maintains one version of each data. However, this single-versioning design limits the concurrency since the reads need to wait for the writes to become visible during transaction commit. Moreover, to guarantee atomicity, FORD writes many undo logs to back up the old data, which consumes the network bandwidth and decreases throughput.

Enabling multi-versioning is expected to efficiently address the above limitations. By storing multiple versions of each data in the memory pool, the read requests are able to fetch existing versions of data rather than waiting for the writes to complete, thus improving the concurrency. Moreover, with multi-versioning, the old versions of data are retained to provide the atomicity, thus eliminating the need of writing undo logs. Prior multi-versioning based distributed transaction processing systems have been proposed in the traditional monolithic architecture [57, 64, 76]. Unfortunately, these systems are difficult to work on the new disaggregated memory architecture due to two challenges, as presented below.

1) Incompatible Transaction Protocol. Prior systems working on monolithic architecture assume that each server has strong CPUs to execute compute tasks in the transaction protocol, e.g., locking [64], validation [57], and timestamp calculation [76]. In general, a single task is not computationally expensive. However, when the number of requests increases, these tasks become substantial and frequent. The CPU in a memory pool is too weak to frequently poll massive tasks and execute them [45, 46, 66, 69, 75, 84, 86]. Therefore, legacy multi-versioning based transaction protocols are not compatible with the disaggregated memory pool.

2) Inefficient Version Structure. To store different versions of data, existing schemes leverage pointer-based structures to dynamically link the versions, called *linked chains* in this paper. In general, there are two types of the linked chains. (1) The old-to-new chain links the versions from the oldest to the newest version [10,25,38,76], as shown in Fig. 1a. (2) The new-to-old chain links the versions from the newest to the oldest version [9,32,57,64,81], as shown in Fig. 1b. To read a specific version, CPU performs *chain walking* that leverages the pointers to fetch the versions one by one until the target version. In fact, the linked chains work well in monolithic servers, since each server contains enough CPUs to quickly perform chain walking in its local memory. However, the linked chains become inefficient in disaggregated memory, since all the application data are stored in the remote memory pool, which does not contain powerful CPU to execute the chain walking. As a result, the compute pool has to perform the chain walking by consuming multiple network round trips to fetch remote versions one after another until the target version, leading to high overheads. Fig. 1c shows that when increasing the number of steps in the chain walking from 1 to 20, the RDMA read latency significantly increases by 24.8× in our testbed (§ 7.1). Moreover, to prevent long chains, the garbage collection (GC) is required to delete the obsolete versions that are no longer used by any transaction [16]. However, when using linked chains, GC is difficult to carry out on disaggregated memory, since the compute pool needs to frequently track the oldest transaction and reclaim the unused versions. Such tracking consumes many round trips for synchronizations and wastes the compute power.

To address the above challenges, we propose Motor, which holistically redesigns the version structure and transaction protocol to enable **m**ulti-versi**o**ning for distributed **t**ransacti**o**n processing on the disagg**r**egated memory. Instead of using linked chains, Motor leverages a new *consecutive version tuple* (CVT) structure to efficiently organize multiple versions of one data in the memory pool. CVT consecutively stores several versions together to fill in continuous address space. In this way, the compute pool is able to fetch all the versions of the same data by reading a CVT in a single round trip, instead of fetching the remote versions one by one, thus reducing the networking overheads to achieve low latency. When the CVT is filled up, Motor leverages a lightweight coordinator-active garbage collection (GC) scheme that reclaims the old versions in a preemptive manner without tracing transaction states. In the presence of GC, Motor also enables the applications to easily identify the consistency between the data value and its version in CVT to guarantee the correctness.

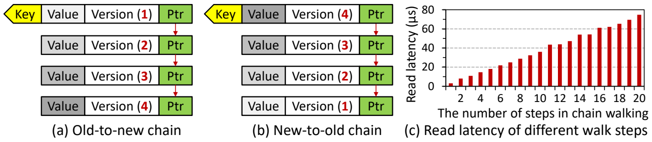


Figure 1: The linked chain based version structures (a, b), and the latency of using RDMA `READ` for chain walking (c).

On top of the CVT structure, Motor designs a fast multi-version concurrency control (MVCC) based transaction protocol. This protocol fully leverages one-sided RDMA to bypass the weak compute units in the memory pool. Our protocol allows the reads not to be blocked by writes, and avoids writing logs, thus improving the concurrency and saving network bandwidth. Moreover, our protocol supports various isolation levels (e.g., serializability and snapshot isolation) to flexibly meet the requirements of different OLTP applications.

In summary, this paper makes the following contributions:

- We propose Motor that enables multi-versioning for distributed transactions on the disaggregated memory.
- Motor designs a new consecutive version tuple (CVT) structure to efficiently organize multiple versions of data in the memory pool. CVT enables the compute pool to obtain the target version in one round trip, and provides lightweight garbage collection without the overhead of tracking (§ 4).
- Motor leverages a fast MVCC transaction protocol that fully exploits one-sided RDMA and CVT to meet the CPU-less memory pool with various isolation-level supports (§ 5).
- We implement[1] Motor and compare it with two state-of-the-art systems [64,84]. The experimental results demonstrate that Motor significantly improves the transaction throughput by up to 98.1% and reduces the latency by up to 55.8%.

2 Background and Motivation

2.1 Memory Disaggregation

Traditional datacenters consist of many monolithic servers, each of which contains a set of compute and memory units. However, this monolithic architecture suffers from low resource utilization and coarse failure domain [65,72]. Specifically, even if a user only needs more *compute* power, we have to add more entire servers in which the *memory* modules are wasted. Moreover, if a CPU is broken, the whole server becomes unusable, which expands the failure domain.

To improve resource utilization and failure isolation, memory disaggregation [20,35,46,50,51] becomes a promising solution, which decouples the compute and memory resources from a monolithic server to build separate resource pools. These pools are connected via fast network, e.g., RDMA [29] or CXL [7]. A compute pool contains many strong CPUs to intensively execute computing tasks. There are small amounts of DRAM in the compute pool to cache some metadata. Moreover, a memory pool consists of substantial memory modules to store the large-volume application data. The memory pool does not contain strong compute capability [46,65,69,72,75], but have some low-power compute units only for memory allocation and network interconnection [84,86]. By efficient resource pooling, datacenters are able to provide appropriate amounts of compute and memory units to meet the requirements of different applications in an on-demand manner, thus

[1] Source code is available at https://github.com/minghust/motor.

improving the resource utilization and reducing costs [48]. Moreover, even if a CPU fails in the compute pool, the decoupled memory modules in the memory pool are not affected due to the separate architecture, thus narrowing the failure domain. Therefore, memory disaggregation is a promising solution for modern datacenters and cloud providers. Without loss of generality, this paper considers that the compute pool leverages one-sided RDMA verbs (including `READ`, `WRITE`, and atomics such as `CAS` an `FAA`) to access the application data in the memory pool to bypass remote CPUs like existing studies [53, 66, 75, 84].

2.2 Transactions on Disaggregated Memory

System Model. To provide atomicity and strong consistency for applications on the disaggregated memory, the compute pool is required to employ distributed transactions to access remote data in the memory pool [84]. Specifically, the CPU threads in a compute pool run many ***coordinators***, which execute a transaction protocol to read data, handle conflicts, and commit updates. The compute pool does not store application data, but contains a small amount of DRAM to buffer some metadata (e.g., remote data addresses). The memory pool stores all the application data without running compute tasks. Each data is replicated into multiple replicas for high availability. In practice, the fail-stop failure [36] could occur at any time to cause the data in the memory pool inaccessible[2] [27]. To tolerate such failures, we adopt the $(f+1)$-way primary-backup replication [42] to generate 1 primary replica and f backup replicas for each data in the memory pool. Each replica can be accessed by multiple coordinators. During transaction processing, coordinators in compute pool read/write remote replicas via network at the byte granularity, and the compute units in memory pool are not involved. Since the coordinators and replicas are fully separated by network, all transactions become distributed in our system model.

Limitations of Single-Versioning. Recently, FORD [84] supports distributed transactions on the disaggregated memory and stores the latest version of each data in the memory pool. This *single-versioning* design simplifies the memory store but incurs two limitations. (1) *Low concurrency.* During transaction commit, the data being updated cannot be read. FORD makes these data invisible until completing the write, thus blocking the read operations; (2) *High logging overheads.* FORD writes the undo logs to all replicas to guarantee atomicity. These undo logs consume the network bandwidth, and the coordinator needs to wait for all ACKs of the logging requests before committing the updates to remote replicas.

2.3 Enabling Multi-Versioning

To address the limitations of single-versioning, we adopt a *multi-versioning* methodology to store multiple versions of each data in the memory pool. By doing so, the writes do not block reads, since the read request obtains an existing version of data, instead of waiting for the update operation, thus improving the concurrency. Moreover, the multi-versioning design does not need to additionally write logs to back up data in replicas, since the old versions naturally act as "undo logs" to guarantee atomicity. In this way, we eliminate the logging overheads to accelerate transaction commit.

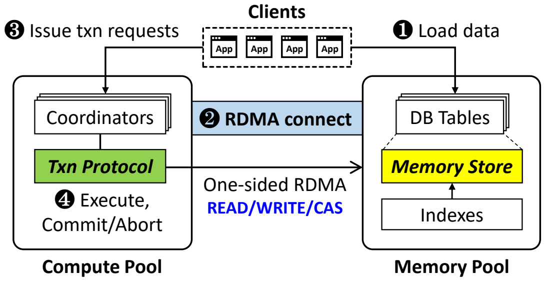


Figure 2: The system overview of Motor.

Challenges. Existing studies have adopted multi-versioning in transaction processing [16, 43, 57, 64, 76]. However, as analyzed in § 1, these studies do not fit the new disaggregated memory architecture due to two reasons. (1) Their transaction protocols target on traditional monolithic servers, which requires powerful CPUs in each server to execute substantial compute tasks [57, 64, 76]. However, in the disaggregated memory architecture, the compute units in the memory pool are too weak to frequently handle compute tasks [75, 84, 86]. (2) The version structures of new-to-old and old-to-new linked chains incur substantial RDMA round trips for chain walking and high overheads for garbage collection.

To address the above challenges, we propose Motor to efficiently enable multi-versioning for fast distributed transaction processing on the disaggregated memory.

3 Motor Overview

Fig. 2 illustrates the system overview of Motor, which contains two parts working in harmony. First, the *Motor memory store* (§ 4) efficiently organizes multiple versions of data in the memory pool. Second, the *Motor transaction protocol* (§ 5) handles multi-versioning based distributed transactions in the compute pool.

Workflow. We outline the workflow of Motor. ❶ The client initially leverages the CPUs in the memory pool to allocate memory to load the application data into relational database (DB) tables. These tables are organized by our consecutive version tuple (CVT) structure, as described in § 4.1. The CVTs can be quickly accessed using indexes, e.g., hash table [86] or B+tree [75]. ❷ We establish RDMA connections between the compute and memory pools. Moreover, the memory pool sends some metadata (e.g., the address of the RDMA memory region and descriptions of indexes) to the compute pool. These metadata help coordinators locate the remote data at runtime. ❸ The clients issue transactions to the compute pool to be executed. ❹ The compute pool uses CPU threads to simultaneously run many coordinators, which leverage our

[2] In line with existing studies [27, 38, 39, 64, 77, 84], we currently do not consider the byzantine failures [37].

transaction protocol to process transactions. In general, the coordinators fetch and lock remote data, and then execute the transaction logic. After execution, the coordinators validate that the data versions are not changed. Finally, the coordinators commit the updates to remote memory pool and unlock data. Our protocol enables coordinators to fully use one-sided RDMA to bypass the weak CPUs in memory pool during transaction processing.

4 Motor Memory Store

4.1 Consecutive Version Tuple

Key Idea. Motor proposes a *consecutive version tuple* (CVT) structure to maintain different versions of data in the memory pool. Unlike the linked chains using pointers to link versions, CVT consecutively stores the versions together to fill in continuous address space. By using CVT, the coordinator is able to fetch multiple versions in a single RDMA `READ`, instead of performing the chain walking to read remote versions one by one until the target version. After fetching the CVT, the coordinator locally searches for the target version, which is fast due to not involving any network I/O.

Structure. Fig. 3 shows the structure of the memory store in the memory pool, which is organized by CVTs. All the CVTs form a CVT region. A CVT consists of a header and several version cells (Vcells). In a header, `TableID` indicates the DB table this record belongs to. A record is a row of user data, containing the key and value, in a DB table. Moreover, `Key` is the unique identifier of this record, and `Lock` is used for concurrency control in transaction processing (§ 5.1). The `AttrBarPtr` points to an attribute bar in the value region. An attribute bar stores the modified attributes of different versions of a record's value, as described in § 4.2. The `VpkgPtr` points to a value package (Vpkg) in value region. A Vpkg contains the actual data value, which is wrapped by a `VpkgSA` and a `VpkgEA` to indicate whether the value is completely written, as explained in § 4.4. Moreover, in a Vcell, the `VcellSA` and `VcellEA` work with the `VpkgSA` and `VpkgEA` to check the consistency between a version and its value (§ 4.4). The `Valid` indicates whether this version of value is available, and the `Version` represents a version number. In addition, the `Bitmap` indicates the modified attributes at the current version, and the `StartOffset` represents the offset of attributes stored in the attribute bar (more details are presented in § 4.2).

Number of Versions in CVT. Motor needs to configure the number of versions (VNum) to hold in CVT. Considering that the memory pool does not contain powerful CPU to dynamically adjust VNum in transaction processing, Motor sets VNum to be fixed, i.e., a record has a fixed maximum number of versions. In fact, it is challenging to determine an efficient VNum due to the tradeoff among read latency, memory footprint, and transaction abort rate. Specifically, if VNum is too small, the CVT size becomes small, which decreases the RDMA transmission payload to decrease the read latency, and also reduces the memory footprint in memory pool. However, due to limited available versions in CVT, the garbage collection (§ 4.3) can be frequently triggered, and this may increase transaction aborts to hamper the throughput when the contention is high. In contrast, if VNum is too large, it helps mitigate transaction aborts, but would waste memory in read-intensive workloads that do not require many versions of data. Moreover, since an entire CVT is read at a time, a large CVT increases the payload to lengthen the RDMA read latency. We explore such tradeoff in § 7.2 and § 7.6, and observe that a suitable VNum significantly depends on the characteristics of workloads (e.g., the access contention and the number of records to read in a transaction). In general, setting VNum to 2 is sufficient for low-contention workloads with short-running transactions (e.g., TATP [1]). For high-contention workloads with long-running transactions (e.g., TPCC [13]), a slightly larger VNum (e.g., 4) efficiently reduces transaction aborts without heavy memory footprint and high read latency.

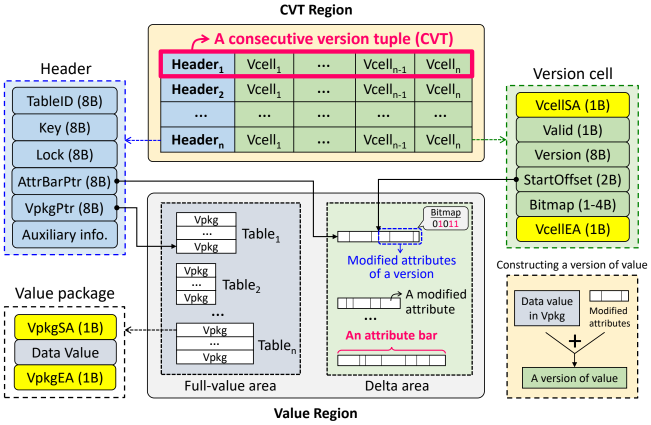


Figure 3: The structure of the Motor memory store, which is organized by CVTs in the disaggregated memory pool.

Indexes Supports. Motor provides unified interfaces for coordinators to quickly access remote CVTs by leveraging indexes (e.g., hash table [86] and B+tree [75]). Motor stores CVTs within the index. For example, when using B+tree indexes, CVTs are stored in leaf nodes, and the internal pointer nodes are cached in compute pool to reduce remote tree traverses. When using hashing indexes, CVTs are stored in hash tables by hashing `Key`s. Therefore, writing CVTs simultaneously modifies the index. Without loss of generality, our paper considers to use the hash table as a case in point to present the details of indexing remote data like existing studies [26, 78, 84]. To address hash collisions, Motor reserves multiple slots in a hash bucket [86]. Each slot stores one CVT. Given a key (e.g., `K0`) of a record, the coordinator hashes `K0` to obtain the ID of hash bucket and calculate the remote address of this bucket. The coordinator then reads the bucket and locally traverses slots to search for the target CVT whose `Key` is equal to `K0`.

CVT Address Cache. In practice, it is expensive to fetch an entire hash bucket each time when reading a CVT. To address this issue, Motor enables each coordinator to leverage a small private *CVT address cache* in the compute pool to store the remote addresses of CVTs. When reading the same CVTs next time, the coordinator can quickly use the cached addresses to directly read the CVTs instead of hash buckets. However, if the `Key` of fetched CVT mismatches the queried key, the cached address becomes stale. The coordinator addresses this issue by re-reading the hash bucket to confirm the existence of the target CVT, and then updates its address cache. To store millions of addresses (each one is 8B), an address cache only consumes several MBs of DRAM space, which is acceptable for the compute pool [65, 84].

4.2 Separate Value Region

Some prior studies like FORD [84] and Silo [71] store the value together with its version, so that coordinators can fetch the value and version in one read. However, this design becomes inefficient in our context, because storing the value together with its version significantly increases the CVT size, leading to high read latency and network bandwidth waste (all values are transmitted but only one is needed). Such drawbacks become even worse when the value size gets larger.

To tackle the above challenge, Motor separates the CVTs from data values in memory pool. The coordinator first reads a CVT to determine the target version, and then reads the corresponding value. In this way, the CVT size is not affected by the value size to achieve stable low read latency, and only one data value is transmitted to mitigate bandwidth wastes.

Reducing Memory Overhead. In the value region, storing a full-sized data value for each version simplifies the data store but wastes memory space. To alleviate the memory overhead, we have two observations. (1) The records in a relational DB table follow the same schema, which defines the number of attributes of the value and the size of each attribute. (2) When updating a record, a transaction can modify only one or several attributes. For example, in TPCC, the value of a record in `DISTRICT` table contains 9 attributes (100B in total), but in `NEW_ORDER` transaction only one attribute is modified, i.e., `D_NEXT_O_ID` (4B). Based on these observations, Motor stores the variable-sized modified attributes, instead of full-sized values, to maintain different versions of values for any record, thus reducing the memory overhead. Fig. 3 shows that the value region contains a full-value area plus a delta area. The full-value area stores the newest version of full-sized values, and the delta area stores old attributes being modified by transactions (like "undo logs"). Therefore, an updated record has only one full value and different versions of variable-sized attributes that are actually modified. To construct an old-version value, we only need to apply the attributes at the old target version into the newest full value.

Attribute Bar. In the delta area, Motor leverages a new structure, called *attribute bar*, to consecutively and compactly store the modified attributes of a record across transactions, as illustrated in Fig. 3. Motor uses the following metadata in CVT to efficiently manage attributes bars.

1) `AttrBarPtr` in Header. When a record is updated for the first time, the coordinator allocates an attribute bar in the delta area, and keeps the remote address of the attribute bar (i.e., `AttrBarPtr`) in the CVT's header.

2) `Bitmap` in Vcell. The coordinator uses a bitmap in Vcell to represent the modified attributes at the current version. For example, if a value has 8 attributes and the 1st, 2nd, and 4th attributes are modified by a transaction, the coordinator writes a bitmap of "00001011" (the rightmost bit represents the first attribute, i.e., the little-endian style) into the Vcell. The length of bitmap depends on the number of attributes.

3) `StartOffset` in Vcell. This is used to represent the offset of a group of modified attributes at the current version inside the attribute bar. The initial `StartOffset` is 0. The coordinator calculates a new `StartOffset` by using the last-written Vcell's `StartOffset` and `Bitmap`. Specifically, according to the positions of "1" in the last-written bitmap, the coordinator accumulates the total size of attributes in the last write, and adds this total size with the last-written `StartOffset` to obtain a new `StartOffset`.

Attribute Bar Size. A coordinator needs to allocate a proper-sized attribute bar to hold modified attributes to alleviate memory wastes. By sampling transaction execution, we observe that for records in a DB table, the total sizes of attributes being updated per transaction (called TotAttrSizes) are different but occur at specific frequencies. For example, in TPCC's `CUSTOMER` table, the TotAttrSize can be 512B, 12B, and 4B, respectively occurring at frequencies of 10%, 88%, and 2% across transactions. This is because in OLTP scenarios, the transaction logic specifies the attributes to update, and different transactions follow the standard execution ratio in the transaction mix [1, 4, 13]. According to the frequencies of different TotAttrSizes, Motor reserves corresponding proportions of space in the attribute bar to hold these attributes of VNum versions (i.e., if some attributes are more frequently updated, Motor reserves more space for these attributes). Hence, Motor approximately estimates the attribute bar size (ABS) = $\sum_{i=1}^{n}(\max(VNum \times Frequency_i, 1) \times TotAttrSize_i)$, where n is the number of TotAttrSizes. For example, when VNum = 4, the ABS of records in `CUSTOMER` table is: 1×512B + 3×12B + 1×4B = 552B, which is sufficient to hold modified attributes of different versions without wasting memory. Note that even if all attributes of a value are modified at some versions (i.e., TotAttrSize = full-value size), the attribute bar can still store all these attributes, since in this case the calculated ABS is guaranteed to be larger than the full-value size.

Mitigating Contentions on Allocating Attribute Bars. When coordinators simultaneously allocate attribute bars, they will compete for the free space in delta area, leading to high contentions. To avoid this, Motor pre-assigns a small MB-scale delta space with proper size (based on ABS) in the delta area to each coordinator. In this way, the coordinator allocates attribute bars in its own delta space without competing with others. The `AttrBarPtr` is globally visible to all coordinators after completing the update operation, so that a coordinator is able to append attributes to the attribute bars created by other coordinators. In rare cases the delta space is exhausted, the coordinator informs remote CPU to allocate larger space.

One RTT for Reading/Writing Values. Though the full value and attributes are separated, Motor consumes only one round-trip time (RTT) to read/write a value at target version. (1) *Read*. A coordinator selects the target version (e.g., $V0$) in a CVT. The selection scheme is presented in § 5.1. If $V0$ is the newest version, the coordinator reads the full value using RDMA `READ` in one RTT. Otherwise, the coordinator calculates remote addresses of the required old attributes by using `AttrBarPtr` in CVT header and `StartOffset` as well as `Bitmap` in the Vcells whose `Version` is larger than $V0$. The coordinator then uses batched RDMA `READ`s to read the full value and old attributes together in one RTT and locally constructs an old version of value. (2) *Write*. The coordinator uses batched RDMA `WRITE`s to update the full value and appends old attributes to the attribute bar together in one RTT.

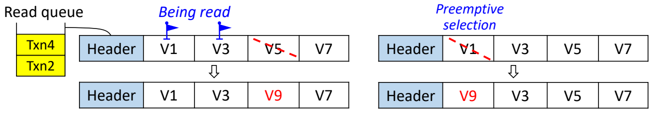
4.3 Coordinator-Active Garbage Collection

If there is no empty Vcell when updating data, we need a garbage collection (GC) mechanism to reclaim the obsolete versions. Legacy GC schemes track the oldest running transactions and delete the versions that are no longer used [16,64]. However, since the compute unit in the memory pool is not aware of transaction states, it is difficult to apply tracking in the memory pool. On the other hand, if the compute pool performs tracking, the coordinators need to confirm which versions are unused among all the in-flight transactions. This increases the network round trips for synchronizations and wastes the compute power.

In order to avoid the overhead of tracking, Motor proposes a *coordinator-active GC* scheme. The idea is that, if there is no empty Vcell, Motor allows the coordinator to actively select a victim version to be overwritten by the new version to complete GC. This scheme is lightweight due to eliminating the need of tracking the oldest running transaction.

To select the victim version, Fig. 4a shows a baseline scheme that skips the versions being read in a CVT, and selects the oldest version in the remaining versions. A read queue is reserved in each CVT to store the timestamps of transactions that are reading the CVT. Other coordinators check the read queue and skip the in-use versions. However, for read operations, since the coordinator does not know the current position of the queue's tail, it has to use RDMA `FetchAndAdd` to atomically move the tail, and then use RDMA `WRITE` to insert a timestamp to the read queue. Such extra RTTs in each read significantly increase the latency.

We observe that the oldest version in CVT has the smallest probability to be used, given that RDMA significantly accelerates transactions [26,78]. Hence, Motor enables coordinators to preemptively select the oldest version in CVT as the victim, as shown in Fig. 4b. This GC scheme avoids the RTT overhead in the baseline method. The tradeoff is that some long-running transactions would be aborted if their previously read data are quickly reclaimed. Nevertheless, the experimental results in § 7.2 show that reserving a proper number of versions in CVT efficiently mitigates such aborts. Overwriting old versions will make the versions in CVT unsorted, but the correctness is not affected, since the coordinator locally traverses all the versions in CVT to locate the target one.



(a) Skipping the versions being read (b) Overwriting the oldest version

Figure 4: Different garbage collection schemes for CVT.

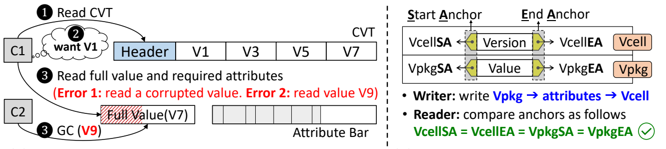
Note that if the attribute bar does not have enough space, the coordinator reclaims old attributes from the start of the attribute bar to write newly modified attributes. In this procedure, the coordinator checks which Vcells correspond to the reclaimed attributes, and sets the `Valid` in these Vcells to 0 to delete these versions. Since Motor appropriately configures the size of attribute bar to store attributes of multiple versions, reclaiming the old attributes does not invalidate many Vcells.

4.4 Anchor-Assisted Read

To obtain a data value, the coordinator reads a CVT to select the target version, and then reads the full value and necessary attributes. As shown in Fig. 5a, coordinator `C1` reads a CVT and needs the value at version $V1$ (Value$_{V1}$). `C1` reads the full value (Value$_{V7}$) and old attributes to reconstruct Value$_{V1}$. At this point, another coordinator `C2` is performing GC to reclaim version $V1$ and write Value$_{V9}$. As a result, there are two incorrect results for `C1`. (1) `C1` reads a corrupted full value due to being partially updated by `C2`. (2) `C1` reads Value$_{V9}$ but mistakenly regards it as Value$_{V7}$, thus reconstructing an incorrect Value$_{V1}$. The root cause of this issue is that the version and data value are separately stored, which prevents coordinators from "atomically" reading a value and its version.

To address the above challenge, Motor proposes an *anchor-assisted read* scheme to help coordinators identify the consistency between the version and value. As shown in Fig. 5b, this scheme uses two anchors at the start and end of a Vcell, called VcellSA (i.e., Vcell's **S**tart **A**nchor) and VcellEA (i.e., Vcell's **E**nd **A**nchor). Similarly, in a Vpkg, two anchors (VpkgSA and VpkgEA) are used to wrap the full value. An anchor is 1 byte. A pair of SA and EA and the content they wrap are implemented in a C++ `struct`, allowing a coordinator to access them together using a single RDMA `READ` or `WRITE`.

To make anchors efficiently work, coordinators follow two rules. (1) *Write*. A coordinator increases the anchor value by 1 for all the four anchors (i.e., VpkgSA, VpkgEA, VcellSA, and VcellEA) to make them **equal**. The coordinator writes the Vpkg first, then the modified attributes, and finally the Vcell. (2) *Read*. A coordinator reads a CVT and then fetches the Vpkg and necessary attributes. Since the full value region stores the newest value, the VpkgSA and VpkgEA are also


❶ Read CVT
❷ want V1
C1
CVT
Header
V1
V3
V5
V7
❸ Read full value and required attributes
(Error 1: read a corrupted value. Error 2: read value V9)
C2
Full Value(V7)
❸ GC (V9)
Attribute Bar
Start Anchor
End Anchor
VcellSA
Version
VcellEA
Vcell
VpkgSA
Value
VpkgEA
Vpkg
• Writer: write Vpkg → attributes → Vcell
• Reader: compare anchors as follows
VcellSA = VcellEA = VpkgSA = VpkgEA ✓

(a) Reading an incorrect value caused by concurrent GC

(b) Using anchors to detect incorrectness

Figure 5: The anchor-assisted read scheme.

the newest. Hence, the coordinator checks whether the newest VcellSA and VcellEA in CVT are equal to VpkgSA and VpkgEA. If the four anchors are equal, the full value and attributes are not modified since the last read. The coordinator then safely reconstructs the target-version value by copying the fetched old attributes into the newest full value. However, if any of the two anchors are not equal, the coordinator aborts the transaction due to detecting partial updates or a conflicting in-flight GC procedure. In essence, the four anchors assist the coordinator to read a version and the corresponding value in an "atomic" manner. Unlike Silo [71] that reads the version twice to confirm consistency, our scheme only needs to read once and compares the four anchors to identify consistency.

Guaranteeing Write Order. The correctness of the anchor-assisted read scheme is based on that all the written data are installed into the memory pool in the correct order, which has two requirements. **[R1]** Vpkg → modified attributes → Vcell. **[R2]** Inside a Vpkg (or Vcell): start anchor → content → end anchor. In practice, the two requirements are satisfied in network and at remote RDMA NIC (RNIC), because (1) the *reliable connection* mode for one-sided RDMA guarantees that the transmitted messages are not lost or reordered [6], and (2) when the request reaches the remote RNIC, the RNIC ensures that the RDMA `WRITE`s are totally ordered with regard to each other [61], i.e., these write requests are sent to the on-chip integrated memory controller (iMC) in order. However, the two requirements can be then violated due to DDIO (i.e., Data Direct I/O [8]). If DDIO is enabled, iMC sends the written data to the L3 CPU cache. Due to unpredictable cache behavior, the data in L3 cache could be evicted to memory out of order to break **R1** and **R2**. In fact, DDIO aims to improve the cache locality, which benefits the CPU execution in traditional monolithic servers, but becomes useless in the disaggregated memory, since the weak CPU in memory pool is not involved during transaction processing. Hence, Motor disables DDIO in the memory pool, so that iMC directly sends writes from its internal first-come-first-serve write pending queue to the main memory. In this way, the writes are installed into remote memory in the correct order to satisfy **R1** and **R2**.

5 Motor Transaction Protocol

We present the Motor transaction protocol. Our protocol works in a widely-recognized transaction processing framework, which includes reading data, handling conflicts, and writing data back. The main difference from existing studies [27, 39, 64, 77, 78, 84] is that our protocol fully exploits the CVT structure and pure one-sided RDMA to support MVCC based distributed transactions on the disaggregated memory.

Timestamp Generation. Motor leverages sequential numbers as transaction timestamps (i.e., 1, 2, 3 ...), which are also adopted as data versions. In fact, the timestamp generation is orthogonal to our designs. Existing studies propose scalable timestamp generation schemes [24, 38, 64, 76], which can be applied to the compute pool as the timestamp service to assign strictly and monotonically increasing timestamps. Our paper does not focus on optimizing the timestamp generation, and we assume that a scalable timestamp service is efficiently leveraged in the compute pool to serve for all coordinators.

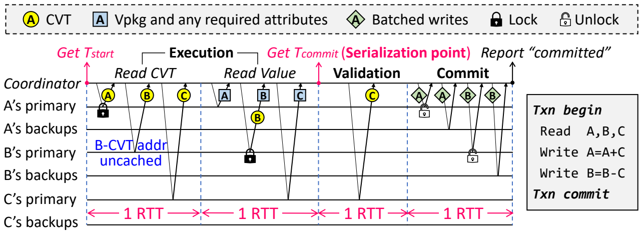
Overview. In the memory pool, each table is replicated to 1 primary and f backups, and the weak CPUs are not involved during transaction processing. In the compute pool, the coordinators leverage our protocol to execute transactions and access remote data through one-sided RDMA.

5.1 Processing Phases

Fig. 6 shows the procedure of handling a read-write transaction (e.g., `T0`) with serializability guarantee. All requests in the same RTT are issued in parallel. The read-write set is {`A`, `B`} and the read-only set is {`C`}. In Motor, the write set is included in the read set, since (1) for *Updates* and *Deletions*, the coordinator reads remote CVTs before writing data back, and (2) for *Insertions*, the coordinator reads remote buckets to obtain empty CVTs before inserting data. The detailed processing phases are presented below.

Phase 1. Execution. The coordinator obtains a start timestamp (T_{start}) from the timestamp service. For each read-only (RO) or read-write (RW) data, the coordinator looks up its local CVT address cache. (1) If the address has been cached (e.g., `A` and `C`), for the RO data (e.g., `C`), the coordinator uses RDMA `READ` to fetch their CVTs from the primaries; for the RW data (e.g., `A`), the coordinator uses doorbell-batched RDMA `CAS+READ` to respectively lock and read the CVTs from the primaries. The locking request prevents other conflicting transactions from modifying the same CVT at the same time. If the locking request fails, the coordinator aborts the transaction, instead of waiting, to avoid deadlocks. (2) If the address is not cached (e.g., `B`), the coordinator uses RDMA `READ` to fetch a hash bucket and then locally search for a `Key`-matched CVT. After obtaining the CVT, the coordinator selects a target version $V0$, which is the largest version among all the versions that are smaller than T_{start}.

Early Abort. If the coordinator observes a version (e.g., $V1$) larger than T_{start} in the CVT, it means that another transaction `T1`, has committed after `T0`'s T_{start}. In this case, the coordinator can *early abort* `T0` to guarantee serializability. The reason is that, even if using T_{start} to select $V0$ for execution, `T0` will be aborted in the next Validation phase, in which `T0` will obtain a larger commit timestamp than `T1`. That is, `T0` with a larger commit timestamp should have used `T1`'s update, i.e., $V1$, for execution, but `T0` used $V0$. Hence, the coordinator early aborts `T0`. Note that the early abort is unnecessary in the snapshot isolation, since it is sufficient for `T0` to read a snapshot at T_{start}, even if the snapshot becomes slightly stale [76].



```
Txn begin
 Read  A,B,C
 Write A=A+C
 Write B=B-C
Txn commit
```

Figure 6: The distributed transaction protocol of Motor.

After version selection, the coordinator uses batched RDMA `READ`s to read the Vpkgs and any required old attributes to construct the target-version value (§ 4.2). Note that for RW data that have not been locked (e.g., `B`), the coordinator additionally batches RDMA `CAS` with `READ`s to lock and re-read their CVTs when reading Vpkgs. After fetching all the data, the coordinator performs three checks for correctness: (1) if any locking fails, `T0` is aborted; (2) if a newer version larger than $V0$ occurs in the re-read CVT, `T0` is aborted, since another transaction has updated this data; (3) if the four anchors are not equal, `T0` is aborted, because the version and value are inconsistent. If passing all checks, the coordinator safely uses the data value inside the Vpkg to execute the transaction logic. Though Motor uses two RTTs to read the CVT and data value, the network payload is significantly reduced due to not transmitting unnecessary data values.

Phase 2. Validation. After all the remote CVTs of the RW data are successfully locked, the coordinator obtains a commit timestamp (T_{commit}) from the timestamp service. Note that if the read-write transaction does not contain any RO data, the following operations can be skipped to reduce latency, since all the RW data have been already locked. However, if the transaction contains RO data, the coordinator needs to validate that the versions of RO data are not changed from T_{start} to T_{commit} to provide serializability. To this end, the coordinator re-reads the CVT of each RO data from remote primaries and uses T_{commit} to select a version V', which is the largest version among all the versions that are smaller than T_{commit}. The coordinator checks whether any of the two cases occur: (1) the CVT is locked by another coordinator, or (2) $V' \neq V0$. In the first case, it is possible that another transaction with a lower T_{commit} is committing a new version. The second case means that another transaction with a lower T_{commit} has committed a new version. If either case occurs, the validation fails, because `T0` with a higher T_{commit} should read the new version but fails to do so in the Execution phase. As a result, `T0` is aborted to ensure serializability. In short, the validation succeeds only if the CVT is not locked and $V' = V0$.

Phase 3. Commit. When the validation succeeds, a coordinator commits the updates to all remote replicas together in a single RTT. The coordinator locally prepares the data to be written, which can be interpreted in three scenarios. **(1) Update.** If the record is updated for the first time, the coordinator allocates an attribute bar in its own pre-assigned delta space. The coordinator then finds an empty Vcell (i.e., `Valid` is 0) in the fetched CVT, sets the `Valid` to 1, fills the `Version` using T_{commit}, sets the `Bitmap` of the updated attributes, calculates the `StartOffset` inside the attribute bar, and configures both of VcellSA and VcellEA to be equal to a new number. If there is no empty Vcell or the `StartOffset` exceeds the length of attribute bar, the coordinator actively performs GC to reclaim old versions. Moreover, the coordinator collects the modified attributes that will be written to the attribute bar. The coordinator then prepares a new Vpkg by filling the new data value, and setting both of VpkgSA and VpkgEA to be equal to VcellSA. **(2) Insert.** Apart from preparing the Vpkg and Vcell like the Update operation, the coordinator prepares a new header and fills the `TableID`, `Key`, and `VpkgPtr`. The `TableID` and `Key` come from applications. The coordinator allocates the `VpkgPtr` in its delta space, i.e., Motor allows the newly inserted data to share the delta area with attribute bars to improve the space efficiency. **(3) Delete.** The coordinator sets the `Valid` of $V0$ to 0, so that subsequent transactions with larger timestamps cannot use the deleted version. The delete operation needs to set the full value in remote memory pool to an old-version value. To this end, the coordinator copies the old attributes fetched in Execution phase into the full value.

After these local preparations, the coordinator leverages doorbell-batched RDMA `WRITE`s to write the prepared data to all replicas and unlocks primaries in one RTT. When receiving *all* ACKs from *all* replicas, the coordinator reports “committed” to the application.

Processing Read-Only Transactions. A coordinator obtains a read timestamp (T_{start}) and reads the required CVTs from the primaries. The coordinator uses T_{start} to determine the target version, and then fetches the Vpkgs and any required old attributes from primaries to construct the value at the target version. If the four anchors are equal, the transaction commits, and otherwise aborts. Note that in single-versioning designs, the read-only transactions require validation [27, 39, 77, 84]. However, with multi-versioning, the read-only transactions do not require validation [57] due to obtaining a stable version snapshot at T_{start} (more details are discussed in § 5.2).

5.2 Flexible Support of Isolation Levels

By using our protocol, Motor supports two widely-used isolation levels, i.e., serializability (SR) [11] and snapshot isolation (SI) [12], to flexibly meet the requirements of different OLTP applications. With SR, the concurrent transactions appear to be executed one by one. Moreover, with SI, the transaction reads data from a snapshot at a time, which does not reflect changes made by other in-flight transactions.

Supporting SR. (1) For ***read-write*** transactions, they are serializable at the point of T_{commit} if guaranteeing that all the target versions selected at T_{start} are equal to those at T_{commit}. This property allows the transactions to be considered as executing at their T_{commit} one after another. Motor ensures this property by using locks and validations. i) If a transaction obtains all the locks of CVTs at T_{start}, the versions of read-write data cannot be changed by other transactions until T_{commit}. Hence, the versions of read-write data at T_{start} are equal to those at T_{commit}. ii) During validation, if a transaction detects that the remote CVT is locked or a new version appears at T_{commit}, the validation fails and the transaction aborts, since the previously fetched versions of read-only data become stale. If the validation succeeds, the versions of read-only data at T_{start} are equal to those at T_{commit}. (2) For ***read-only*** transactions, they do not have a commit timestamp due to not making data changes. In the multi-versioning design, since read-only transactions only observe a snapshot, the start time of read-only transactions can be considered to be "movable" in order to find a serializable execution order [57], i.e., the read-only transactions can be placed among other read-write transactions to make all the transactions appear to execute one by one. In summary, the write-write and read-write conflicts between transactions are respectively addressed by using locks and validations, which ensure that the precedence graphs [5] of all the transaction schedules do not contain cycles, thus guaranteeing serializability [68].

Supporting SI. To support SI, Motor disables the version validation for the read-only data in read-write transactions, i.e., these transactions are allowed to use a stale snapshot by using T_{start}. Note that the locking is still required to resolve the write-write conflicts. SI is weaker than SR, but achieves higher performance (as demonstrated in § 7.7) and has been adopted by multiple popular systems, e.g., MySQL [56], PostgreSQL [60], Oracle [59], and SQL Server [63].

ACID Guarantee. Motor guarantees ACID for transactions. (1) **A**tomicity. Motor maintains multiple versions of data, and the old versions act as "undo logs" to preserve the atomicity. (2) **C**onsistency. The data versions in memory pool are in a consistent state before a transaction starts and after it commits. (3) **I**solation. Motor supports serializability and snapshot isolation. (4) **D**urability. Motor stores $f+1$ replicas of each data against data loss, and can employ UPS-backed DRAM [27] or persistent memory [84] in the memory pool to durably store the committed updates even if a power failure occurs.

5.3 Fault Tolerance

Replica Failures in Memory Pool. By enabling data replication, Motor is able to tolerate replica failures in the memory pool. The replica failures can be quickly detected using RDMA [27]. If any replica fails before commit, the coordinator discards all the fetched data, unlocks remote locks, and aborts the transactions. If a primary fails during commit, Motor promotes a backup as the new primary to retain the committed updates, because the backups have the same updates as primary. The new primary is not visible to coordinators until the updates are installed into alive replicas. When the new primary becomes visible and subsequent coordinators can grab locks on the new primary, the updates of previous transactions have been already committed, thus guaranteeing serializability. Moreover, if a backup fails during commit, the coordinator selects another memory node to add a new backup. Adding a backup requires data migration, in which Motor enables memory nodes to use RDMA `WRITE` to quickly transmit application data. Subsequent transactions involving failed replicas hang up until the replicas are recovered. The $(f+1)$-way replication tolerates at most f replica failures.

Coordinator Failures in Compute Pool. In line with existing studies [27, 78], Motor supports to use leases [31] to detect coordinator failures. Motor enables the coordinators to write small-sized operation logs in local memory to record the operations (e.g., the keys that will be locked or committed) during execution. The operation logs are stored in UPS-backed memory and are not lost [27]. If a coordinator fails, Motor employs a new one to use the operation logs to resume the in-flight commit and unlock keys for recovery. For example, the new coordinator uses RDMA `CAS` to unlock the recorded keys, i.e., if the `CAS` succeeds, the previous lock is released to avoid starvation, and otherwise the key is actually not locked.

Network Failures. A network failure causes the network partition. In practice, it is hard to distinguish network failure from server failure. Like uKharon [34], we assume that the network partitions are discovered and resolved by datacenter administrators. If a network partition occurs, either availability or consistency cannot be fully guaranteed according to the CAP theorem [18, 30]. In the context of OLTP applications, offering consistency is more important to satisfy the ACID requirements. Hence, Motor weakens the availability by only allowing the major partition [17] to serve requests.

6 Implementations

We present some important implementation details including the transaction interfaces and execution framework.

Easy-to-Use Transaction Interfaces. Motor provides the following interfaces for applications to easily run MVCC based distributed transactions on the disaggregated memory.

- `TxnBegin()`: Start a transaction and record its ID.
- `GetTS()`: Get a timestamp from the timestamp service.
- `AddObject()`: Add a read-only (or read-write) object to the read-only (or read-write) set.
- `FetchAll()`: Obtain remote CVTs and target-version data values. The remote CVTs are simultaneously locked.
- `Validate()`: Validate the versions of read-only data.
- `TxnCommit()`: Commit the transaction by writing the updates back to remote replicas and unlocking the primaries.

Execution Framework. In the compute pool, Motor uses the CPU cores to spawn massive threads to execute transactions in parallel. However, if using a thread as a coordinator, the CPU core will become idle when waiting for RDMA ACKs, which decreases the throughput. To saturate the compute power of a CPU core, Motor generates multiple coroutines in a CPU thread to execute in a pipeline manner [39, 77, 84]. In a thread, one coroutine polls the RDMA ACKs, and each of the other coroutines acts as a transaction coordinator. Therefore, Motor enables substantial coordinators to concurrently execute transactions in the compute pool.

7 Performance Evaluation

7.1 Experimental Setup

Testbed. We configure four servers connected through a Mellanox SB7890 100Gbps InfiniBand (IB) Switch. Each server contains a 100Gbps Mellanox ConnectX-5 IB RNIC. One server containing Intel Xeon Gold 6330 CPUs is configured as the compute pool to run coordinators. Other three servers form the memory pool, and each server contains 192GB DRAM.

Benchmarks. We leverage a key-value store (`KVS`) as a microbenchmark. `KVS` stores 10M key-value pairs in one database (DB) table. The key is 8B and the value is 40B [39, 84]. In `KVS`, each transaction performs a read or an update operation to a 48B KV pair with skewed accesses following the Zipfian distribution [23]. We enable the skewness and the ratio of read-write transactions in the transaction mix of `KVS` to be configurable to facilitate comprehensive evaluation. Furthermore, we leverage three widely-used OLTP benchmarks, i.e., `TATP` [1], `SmallBank` [4], and `TPCC` [13], to evaluate the end-to-end transaction throughput and latency. Specifically, `TATP` shows a telecom application, which includes 4 DB tables and 80% of the transactions are read-only. `TATP` contains 2M subscribers and the record size is up to 48B. `SmallBank` models a banking application, which contains 2 DB tables and 85% of transactions are read-write. `SmallBank` has 10M accounts and the record size is 16B. `TPCC` models a complex ordering system, which contains 9 DB tables and 92% of transactions are read-write. `TPCC` contains 24 warehouses and the record size is up to 672B. Moreover, for all benchmarks, each DB table is replicated to three memory nodes to maintain a 3-way replication, i.e., 1 primary and 2 backups.

Comparisons. We compare our Motor with two state-of-the-art systems, i.e., FaRMv2 [64] and FORD [84]. FaRMv2 supports multi-versioning for transactions on monolithic servers, and uses the new-to-old chains to link versions [64]. To make FaRMv2 compatible with disaggregated memory (DM), we use one-sided RDMA to implement its transaction protocol, which is referred to as FaRMv2-DM in the rest of this paper. Moreover, FORD supports single-versioning for transactions on the disaggregated memory, and we run its open-source code. Though FORD leverages persistent memory, its one-sided RDMA designs on transaction protocol are also compatible with DRAM. Note that Motor targets on the disaggregated architecture, which is not comparable with the systems running on the monolithic architecture [39, 57, 76].

Performance Metrics. We report the transaction throughput by counting the number of committed transactions per second. Moreover, we report the 50th and 99th percentile latencies of committed transactions as the transaction latency.

7.2 Number of Versions in CVT

We explore how the number of versions (VNum) in CVT affects the performance of Motor. For each benchmark, we vary VNum from 2 to 15. The ratio of read-write transactions in `KVS` is 80%. Fig. 8 and 9 show that as VNum increases, the transaction throughput generally first increases and then decreases. The reason is that, when VNum gets larger, the abort rate of read-only transactions is reduced to increase the throughput. For example, in `TPCC`, the abort rate of a long-running read-only transaction `STOCK_LEVEL` decreases from 32.1% (VNum = 2) to 3.8% (VNum = 4). However, after reaching the peak transaction throughput, increasing VNum no longer significantly reduces aborts, but the CVT size continues to increase, which enlarges the payload size to increase RDMA read latency, as shown in Fig. 7. The increased read overhead overwhelms the benefit of reducing aborts, thus decreasing the performance. Besides, large VNums also consume more memory space, as presented in § 7.6. Fig. 8 shows that at skewness 0.7, `KVS` reaches the peak throughput earlier than 0.99, since a larger skewness incurs higher access contention and requires more versions to reduce aborts.

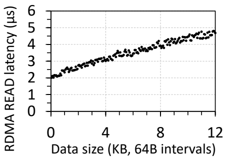
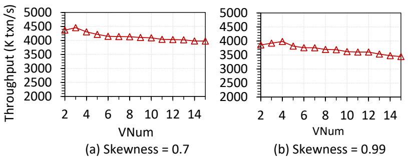


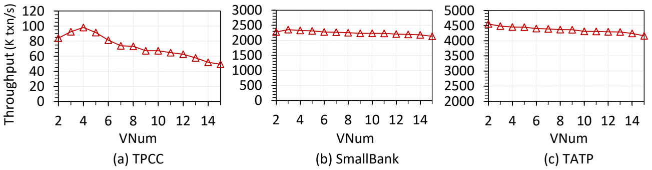
Figure 7: The latency of reading different sizes of data.



(a) Skewness = 0.7

(b) Skewness = 0.99

Figure 8: The transaction throughput on `KVS` benchmark when varying VNum with skewness 0.7 and 0.99.



(a) TPCC

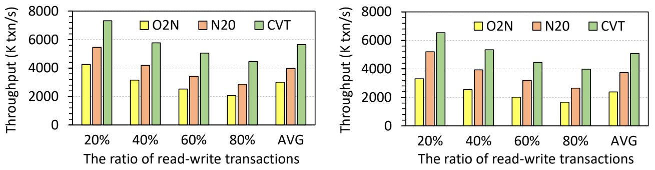
(b) SmallBank

(c) TATP

Figure 9: The transaction throughput on `TPCC`, `SmallBank`, and `TATP` benchmarks when varying VNum.

We observe that, as VNum increases after the peak throughput, the throughput degradation of `TPCC` (up to 49.6%) is heavier than other workloads. This is because one transaction in `TPCC` can access hundreds of records, which is much larger than other benchmarks, e.g., one transaction in `SmallBank` (or `TATP`) only accesses 1–3 (or 1–4) records. Therefore, the overall read overhead (considered as CVT size × number of records) of `TPCC` transactions is more sensitive to VNum, leading to sharper performance decrease. `SmallBank` is write-intensive, but its transactions are short, and maintaining 3 versions reaches the peak performance. `TATP` only requires 2 versions for a record to achieve the peak throughput, since 80% of transactions in `TATP` are read-only and short-running with low contentions. As VNum grows, the high read overhead leads to continuous throughput degradation in `TATP`.

In summary, determining a suitable VNum significantly depends on the characteristics of workloads, including the access contention and the number of accessed records in a transaction. When the contention is low (e.g., `TATP`), setting a small VNum is enough. If the contention is high, more versions are needed to allow higher concurrency, especially for the long-running transactions. We also need to consider



(a) VNum = 3, skewness = 0.7

(b) VNum = 4, skewness = 0.99

Figure 10: The transaction throughput of different version structures on KVS benchmark.

the number of records accessed per transaction to avoid large CVTs incurring high overall read overhead. According to these results, we respectively set the suitable VNum in TPCC, TATP, SmallBank, and KVS to 4, 2, 3, and 4.

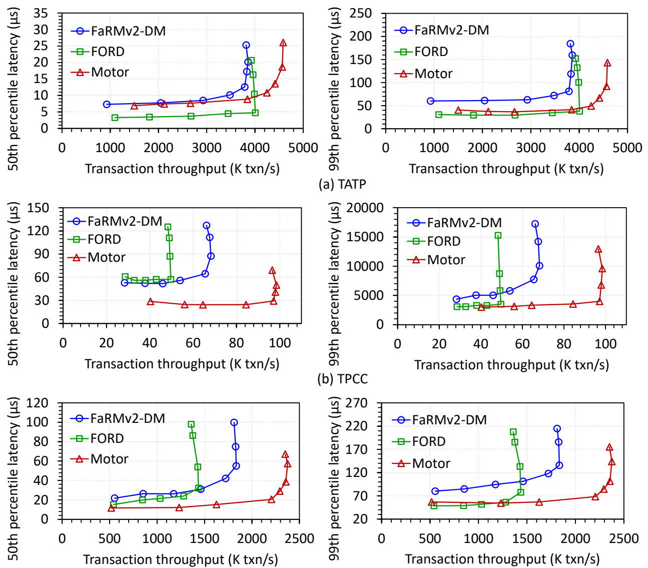
7.3 Performance of Version Structures

We compare the performance of our CVT and traditional linked-chain version structures, i.e., old-to-new (O2N) and new-to-old (N2O), upon the KVS benchmark. We configure the access skewness as 0.7 and 0.99, and vary the ratio of read-write transactions (RW-ratio) from 20% to 80% in the transaction mix of KVS. Based on the results in Fig. 8, we change the maximum number of versions to hold for all structures to 3 for skewness 0.7, and 4 for skewness 0.99.

Fig. 10 shows that CVT respectively improves the throughput by 1.7–2.4× and 1.3–1.6× compared with O2N and N2O. The reason is that, CVT enables the transaction to fetch the target version in a single round trip, while O2N and N2O require multiple round trips for chain walking. When increasing the RW-ratio, the throughputs of three structures decrease, since the write conflicts increase and read-write transactions require more round trips to commit. When the skewness is high (e.g., 0.99) and RW-ratio is low (e.g., 20%), the throughput gap between N2O and CVT becomes small, because the access is more concentrated and many read-only transactions quickly obtain new values from the chain head of N2O. However, such performance gap between O2N and CVT becomes larger at high skewness since the new versions in O2N are placed in the chain tail, which increases the read overhead. Moreover, CVT respectively reduces the 50th (and 99th) percentile latencies by 59.8%/30.8% (and 67.9%/47.7%) on average compared with O2N/N2O at skewness 0.99 due to the same reasons above. We have also examined that when further increasing the maximum number of versions to hold, CVT can deliver more performance benefits over O2N and N2O.

7.4 End-to-End Performance

We leverage TATP, TPCC, and SmallBank to evaluate the end-to-end performance of Motor, FORD, and FaRMv2-DM. All systems guarantee serializability. We configure the maximum number of versions in FaRMv2-DM's version chain to be the same as our CVT for fair comparisons. Fig. 11 illustrates the transaction throughput and latency. To plot a throughput-latency curve, we increase the request load by running 10–40 threads and 2–8 coroutines per thread, i.e., 10–280 concurrent coordinators. Each thread executes 1M transactions following the standard transaction mix of each benchmark [1,4,13].



(a) TATP

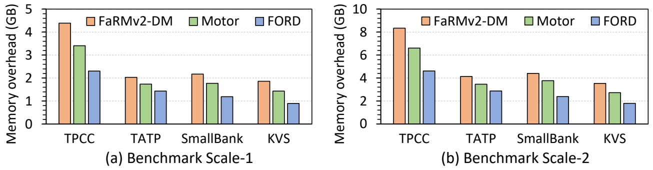
(b) TPCC

(c) SmallBank

Figure 11: The transaction throughput and latency of all the systems on TATP, TPCC, and SmallBank benchmarks.

Compared with FORD, Motor respectively improves the transaction throughput by 14.4% on TATP, 98.1% on TPCC, and 65.4% on SmallBank. FORD adopts the single-versioning design, which limits the throughput, since reads are blocked by writes during commit, and the undo logs consume network bandwidth. Unlike FORD, Motor allows to read existing versions in CVTs, and does not need to write undo logs to remote replicas by maintaining old versions of values. Hence, Motor improves the throughput over FORD. The improvements are higher in TPCC and SmallBank, because (1) they are write-intensive workloads in which Motor avoids many undo logs, and (2) Motor reserves multiple versions to reduces aborts for read-only transactions, especially long-running ones, e.g., STOCK_LEVEL in TPCC. FORD delivers the lowest 50th percentile latency in TATP, since the two transactions, i.e., GET_SUBSCRIBER_DATA and GET_ACCESS_DATA, occupy 70% of the transaction mix, and both of them only read one object. In this case, FORD only uses one RTT to read data, while Motor requires two RTTs to separately read the CVT and data value. However, the 99th percentile latency of Motor on TATP is close to FORD when the transaction becomes complex. Furthermore, Motor reduces the 50th percentile latency by 55.8%/26.2% on TPCC/SmallBank compared with FORD.

Compared with FaRMv2-DM, Motor respectively improves the transaction throughput by 18.9%/44.3%/29.5%, and reduces the 50th (99th) percentile latencies by 8.6% (39.1%) / 52.1% (35.6%) / 43.6% (34.5%), on TATP/TPCC/SmallBank. Motor achieves these improvements due to three reasons. (1) FaRMv2 uses the linked chain to store different versions, which increases network round trips to perform chain walking to obtain the target version. Unlike FaRMv2, Motor uses CVT to fetch the versions together in one round trip. Motor shows the highest improvement over FaRMv2-DM in TPCC, since TPCC requires more versions and the transactions read many



(a) Benchmark Scale-1 (b) Benchmark Scale-2

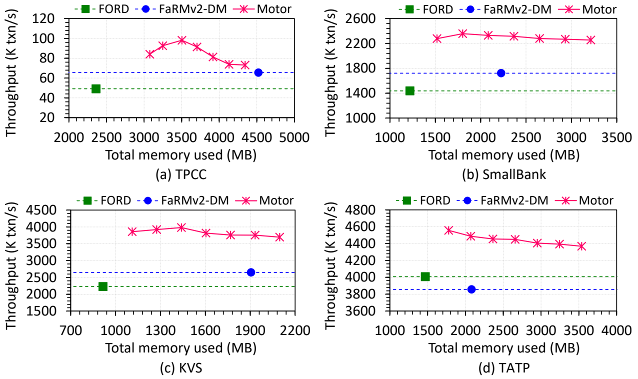
Figure 12: The space consumption in memory pool of all systems at two scales of benchmarks.

records, which exacerbates the chain walking in FaRMv2-DM to cause high overheads. (2) The design of FaRMv2 consumes a dedicated RTT to lock the read-write data, but Motor enables to batch the locking and CVT/value read requests to save RTTs. (3) The design of FaRMv2 uses two RTTs to commit the backups and primaries, while Motor updates all replicas together in one RTT. Moreover, FORD can also achieve lower latency than FaRMv2-DM by alleviating the read overhead, but FaRMv2-DM allows more concurrency in multi-versioning to improve the throughput.

7.5 Memory Overhead

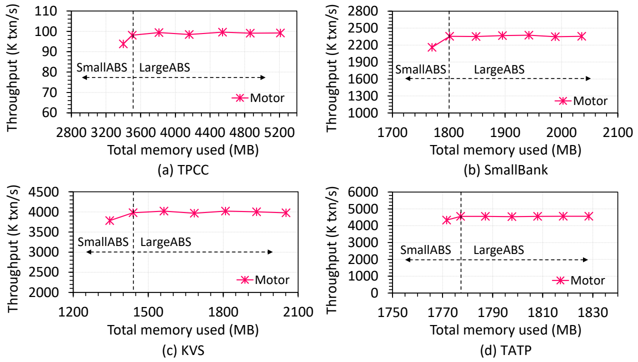
We present the memory overheads of all systems in the memory pool using two different scales of benchmarks. Scale-1 (or Scale-2): TPCC contains 24 (or 48) warehouses; TATP has 2M (or 4M) subscribers; SmallBank has 10M (or 20M) accounts; KVS stores 10M (or 20M) KV pairs with skewness 0.99 and RW-ratio 80%. Scale-1 is the default configuration in § 7.1.

As shown in Fig. 12, FORD exhibits the lowest memory overhead by storing only one version of data. Due to supporting multi-versioning, Motor and FaRMv2-DM consume larger memory space than FORD. Nevertheless, Motor saves memory space in three aspects: (1) maintaining the actually modified attributes rather than full values for different versions; (2) appropriately estimating the size of attribute bar without wasting space; and (3) configuring suitable VNums for different workloads without storing unnecessary versions. For example, Motor supports 4 versions of data in TPCC, but only consumes 1.45×, instead of 4×, of memory space over FORD. Such memory saving is also shown in other benchmarks. In TATP, Motor only incurs 17.3% higher memory overhead than FORD, since only 16% of transactions perform updates and the modified attributes are small. In SmallBank and KVS, Motor respectively consumes 32.7% and 37.7% higher memory space than FORD, since SmallBank and KVS are write-intensive and require more versions than TATP. FaRMv2-DM suffers from 14.6%-22.8% higher memory overhead than Motor due to two reasons. First, FaRMv2 stores a full-sized value for each version, while Motor only stores the modified attributes of values. Second, FaRMv2 requires pointers to link old versions in its version chain, while Motor does not need such pointers since our CVT structure consecutively stores all the versions. Moreover, Fig. 12b shows that when the benchmark scale increases, the gap of space consumption between Motor and FORD generally keeps stable in all benchmarks. This demonstrates that our reduction of memory overhead still works even if the workload scale becomes larger. In summary, Motor trades some extra memory space to achieve better performance than the single-versioning design, while also reducing the memory overhead as much as possible.



(a) TPCC (b) SmallBank (c) KVS (d) TATP

Figure 13: The comparisons of transaction throughput when varying Motor memory footprint by changing VNum.

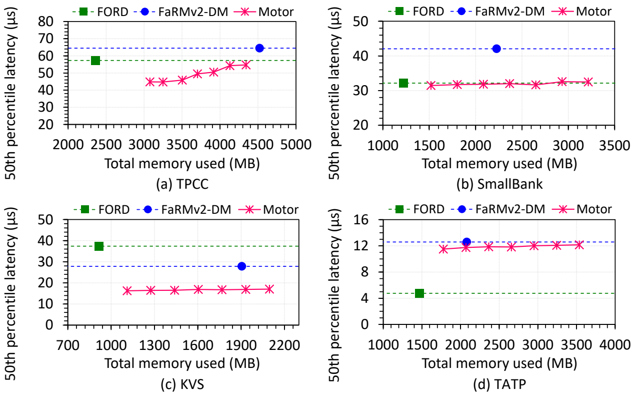


(a) TPCC (b) SmallBank (c) KVS (d) TATP

Figure 14: The transaction throughput of Motor when varying the memory footprint by changing ABS.

7.6 Varying Motor Memory Footprint

We study how Motor performs when varying the memory footprint based on the benchmark Scale-1 (§ 7.5). In the memory pool, since the full values always exist to provide complete user data, we vary Motor memory footprint by changing the number of versions (VNum) and the attribute bar size (ABS). As Motor has significantly reduced the memory overhead, the room to further decrease memory footprint is limited. For example, Motor only reserves 2 versions of data in TATP. This is the minimal number of versions for multi-versioning. Hence, in TATP, we increase VNum up to 8 to increase memory footprints. For other benchmarks, since their suitable VNums are larger than 2, we decrease (and increase) VNum from the suitable VNum to 2 (and 8) to vary memory footprints. When changing VNum (2–8), the corresponding ABS is estimated using the formula in § 4.2. Moreover, to vary ABS, we fix VNum to the suitable VNum in each benchmark, and (1) increase ABS to 2–6× of the estimated ABS using the suitable VNum, and (2) decrease ABS to 1× of the sum of different TotAttrSizes per transaction. Fig. 13–16 show the transaction throughput and latency of Motor when varying memory footprints. We also report the performance and memory footprints of FORD and FaRMv2-DM for comparisons.



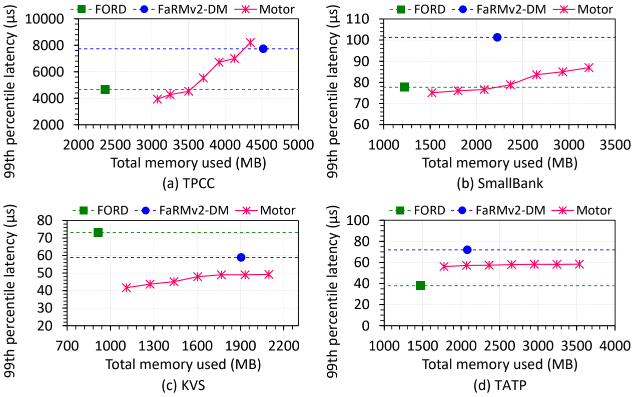
(a) TPCC

(b) SmallBank

(c) KVS

(d) TATP

Figure 15: The comparisons of the 50th percentile latency when varying Motor memory footprint by changing VNum.



(a) TPCC

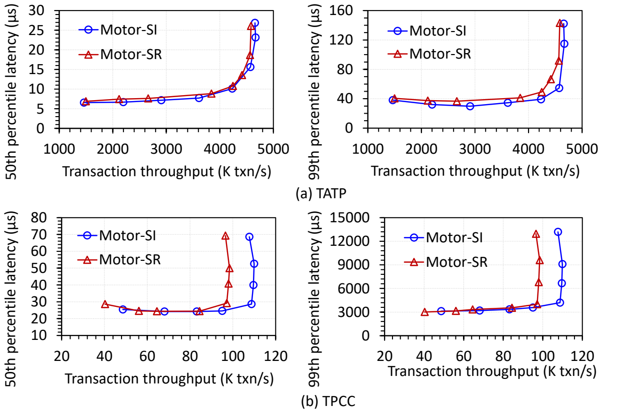
(b) SmallBank

(c) KVS

(d) TATP

Figure 16: The comparisons of the 99th percentile latency when varying Motor memory footprint by changing VNum.

As shown in Fig. 13, when decreasing VNum from the suitable value, the memory footprints of Motor are reduced by up to 22.8% and are close to FORD on many workloads. Through reducing the memory footprint to contain less versions, Motor still achieves higher throughput than FORD and FaRMv2-DM. The reason is that compared with FORD, (1) Motor reserves more than one version to avoid blocking reads and reduce transaction aborts; (2) Motor does not need to additionally write undo logs and the read-only transactions do not need to validate versions with multi-versioning. Moreover, compared with FaRMv2-DM, (1) our CVT structure avoids chain walking to reduce latency; (2) our MVCC protocol saves RTTs via efficient request batching (§ 7.4). When slightly increasing VNum (e.g., from 4 to 6 in KVS), Motor still consumes less memory than FaRMv2-DM thanks to only storing necessary modifications in the delta area. Hence, compared with FaRMv2-DM, Motor can store more versions using a smaller amount of memory. In fact, when VNum increases from 2 to 8 (4×), the Motor memory footprint only increases by 1.4×/2.1×/2×/1.9× on TPCC/SmallBank/TATP/KVS. Fig. 14 shows that when fixing VNum and reducing ABS from the suitable ABS, the throughput decreases, since a small-sized attribute bar would result in more than one Vcells being invalidated in garbage collection to increase aborts. However, when increasing ABS from the suitable ABS, the throughput generally keeps stable, since the transaction aborts are hardly reduced. This demonstrates the efficiency of our estimation on ABS, i.e., reserving an exact and sufficient size for the attribute bar without wasting memory. Fig. 15 and 16 show that the latency of Motor grows when increasing VNum to enlarge the memory footprint, since large-sized CVTs increase the transmission latency. Nevertheless, Motor still exhibits lower latency than FaRMv2-DM by using the CVT to obtain all versions in a single read. In TATP, FORD achieves the lowest latency due to consuming less RTTs to fetch data, as analyzed in § 7.4. But in other benchmarks, Motor shows lower latency than FORD at suitable VNums due to eliminating the overheads of writing logs for read-write transactions and validating versions for read-only transactions. In summary, these results demonstrate the benefits of Motor over state-of-the-art systems when varying Motor memory footprint.



(a) TATP

(b) TPCC

Figure 17: The transaction throughput and latency on TATP and TPCC benchmarks when using different isolation levels.

7.7 Performance of Different Isolation Levels

Motor supports two isolation levels, i.e., serializability (SR) and snapshot isolation (SI). Fig. 17 show that Motor-SI generally achieves lower latency and higher throughput than Motor-SR on both read-intensive (TATP) and write-intensive (TPCC) workloads by eliminating the validation phase for read-write transactions. Compared with TATP, Motor-SI shows higher throughput improvement in TPCC, since TPCC accesses more read-only data per transaction and features higher read-write contentions, thus allowing more throughput improvement when relaxing the isolation requirement.

7.8 Using PM in Memory Pool

Both DRAM and persistent memory (PM) can be used in a memory pool [69, 86]. We leverage six 128GB Intel Optane PM modules in each memory node to evaluate the performance of Motor on TPCC. We use RDMA READ-after-WRITE to flush the written data from remote RNIC to PM for remote data persistency [84]. Fig. 18 shows that the throughput only decreases by 13.1% on PM due to the limited PM bandwidth [80, 84]. The results demonstrate that Motor efficiently works on both DRAM and PM, thus offering good portability for applications to run on different types of memory devices.

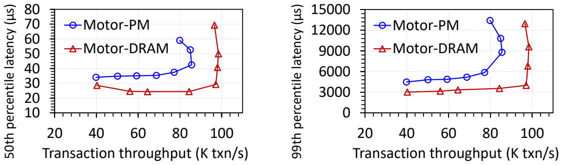


Figure 18: The transaction throughput and latency on TPCC benchmark when using DRAM and PM in the memory pool.

7.9 Fault Tolerance

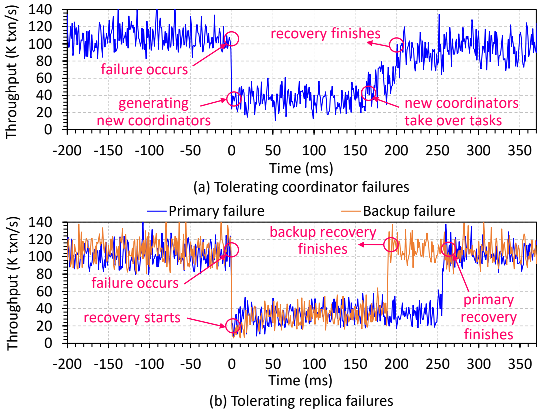
We leverage TPCC to show the resilience of Motor under coordinator failures in compute pool and replica failures in memory pool. We report the instantaneous transaction throughput in 1 ms interval over time (the crash occurs at time 0).

Fig. 19a shows the throughput timeline of recovering coordinators. We run 84 coordinators and 60 of them fail at the same time. Motor then generates 60 new coordinators and establishes network connections, which consumes about 170 ms. Afterwards, the new coordinators take over the remaining tasks. In Motor, each coordinator writes local operation logs to record the operations during execution. These operation logs consume very small space (up to 556B per transaction) and the log space can be reused across transactions. The new coordinators use the operation logs of failed ones to resume in-flight commits and unlock CVTs to avoid starvation. After recovery, Motor regains peak throughput.

Fig. 19b shows the results of recovering replicas. Considering that the CUSTOMER table is frequently used, we respectively allow the primary and one backup of CUSTOMER to fail, i.e., cannot be accessed. A small portion of transactions that do not access the failed replicas are normally executed, and hence the throughput does not become 0. Motor handles the primary failure by promoting a backup as the new primary and adding a backup. Motor tolerates the backup failure by adding a backup. Recovering the primary consumes more time, since Motor needs to change the view of primaries for coordinators, and the new primary is not visible until the updates are committed into alive replicas. Adding a backup requires data migration, during which Motor allows a memory node to use RDMA WRITE to transmit DB tables, CVTs, and attribute bars to another memory node. Write requests to the replicas involved in migration are blocked to guarantee the data consistency among replicas. Since the CUSTOMER table is large, the migration consumes nearly 200 ms. We also examine that if a small DISTRICT table fails, the migration consumes only 1.1 ms. Further optimization on migration is out of our scope. In practice, our ms-scale recovery is acceptable given that prior systems [27, 64, 66] also provide ms-scale recovery.

8 Related Work

Fast Distributed Transactions. Fast distributed transaction processing is a key pillar in distributed systems. Many systems use RDMA to process transactions [22, 26, 27, 39, 41, 58, 64, 77, 78]. Some studies transform a distributed transaction to a local one to reduce the communication overheads [19, 40, 52]. Some protocols on concurrency control [55, 74, 79, 82] and data replication [83] are proposed to improve the performance. The above systems work on the monolithic architecture, while our Motor targets on the disaggregated architecture.



(a) Tolerating coordinator failures

(b) Tolerating replica failures

Figure 19: The Motor's transaction throughput on TPCC over time under (a) coordinator failures and (b) replica failures.

Memory Disaggregation. Memory disaggregation improves the resource utilization. Existing studies explore memory disaggregation in many areas, such as hardware designs [35, 50, 51], operating systems [65], indexes [53, 75, 86], key-value stores [45, 49, 66, 69], networking [29, 67], erasure coding [47, 85], swapping [15, 20, 33, 62], and memory managements [14, 46, 48, 54, 70, 72, 73]. In fact, Motor focuses on transaction processing, which is orthogonal to the above systems. Though FORD [84] supports transactions on disaggregated memory, it adopts single-versioning, which limits the concurrency and incurs high logging overheads. Unlike FORD, Motor enables multi-versioning to address these limitations.

Multi-Versioning Schemes. Multi-versioning schemes have been adopted to support distributed transactions. They focus on high-performance MVCC protocols [28, 43, 57, 64], timestamp generations [38, 76, 81], garbage collections [16, 44], and verifications [21]. These systems are designed for traditional monolithic servers, which do not fit the disaggregated memory. Unlike these studies, our CVT structure and distributed transaction protocol efficiently support multi-versioning on the disaggregated memory.

9 Conclusion

This paper proposes Motor, an efficient distributed transaction processing system for multi-versioning in the context of disaggregated memory. Motor proposes a new consecutive version tuple structure to efficiently organize multiple versions of data in memory pool. On top of this, Motor designs a fully one-sided RDMA-oriented MVCC protocol to accelerate transactions. Extensive experimental results demonstrate that Motor significantly improves the transaction throughput and reduces the latency with moderate memory overhead.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to anonymous reviewers for their constructive suggestions and feedback.

References

[1] Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net, 2011.

[2] Intel® rack scale design architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf, 2018.

[3] Vmware Research: Remote memory. https://research.vmware.com/projects/remote-memory, 2021.

[4] Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank, 2022.

[5] Precedence graph. https://en.wikipedia.org/wiki/Precedence_graph, 2023.

[6] Rdma aware networks programming user manual v1.7. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes, 2023.

[7] Compute express link®. https://www.computeexpresslink.org, 2024.

[8] Intel® Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2024.

[9] MySQL: The world's most popular open source database. https://www.mysql.com, 2024.

[10] PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org, 2024.

[11] Serializability. https://en.wikipedia.org/wiki/Database_transaction_schedule#Serializable, 2024.

[12] Snapshot isolation. https://en.wikipedia.org/wiki/Snapshot_isolation, 2024.

[13] Tpc-c benchmark. http://www.tpc.org/tpcc, 2024.

[14] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[16] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proc. VLDB Endow.*, 13(2):128–141, 2019.

[17] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[18] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[19] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, 2018.

[20] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 79–92. ACM, 2021.

[21] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vmvcc, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 871–886. USENIX Association, 2023.

[22] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17. ACM, 2016.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, 2013.

[26] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[27] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[28] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and general datacenter transactions for on-disk databases. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 343–360. USENIX Association, 2023.

[29] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[30] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[31] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[32] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, 2010.

[33] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[34] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. ukharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 101–120. USENIX Association, 2022.

[35] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 417–433. ACM, 2022.

[36] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[37] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 175–188. USENIX Association, 2008.

[38] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming aborts in all phases for distributed In-Memory transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA,*

USA, November 2-4, 2016, pages 185–201. USENIX Association, 2016.

[40] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[41] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 297–312. ACM, 2018.

[42] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 312–313. ACM, 2009.

[43] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[44] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1307–1318. ACM, 2016.

[45] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[46] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[47] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 181–198. USENIX Association, 2022.

[48] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 574–587. ACM, 2023.

[49] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable rdma-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 99–114. USENIX Association, 2023.

[50] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.

[51] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[52] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[53] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou and. SMART: A high-performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 553–566. USENIX Association, 2023.

[54] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 757–773. ACM, 2020.

[55] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 479–494. USENIX Association, 2014.

[56] MySQL. Transaction isolation levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html, 2024.

[57] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM, 2015.

[58] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 97–108. ACM, 2019.

[59] Oracle. Transaction isolation levels. https://www.oreilly.com/library/view/java-programming-with/0596000871/0596000871_orasqlj-CHP-9-SECT-2.html, 2024.

[60] PostgreSQL. Transaction isolation. https://www.postgresql.org/docs/current/transaction-iso.html, 2024.

[61] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 71–85. USENIX Association, 2022.

[62] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[63] SQL Server. Set transaction isolation level. https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver16, 2023.

[64] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[65] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[66] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.

[67] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki-Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 255–270. USENIX Association, 2019.

[68] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 7th Edition*. McGraw-Hill Education, 2019.

[69] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[70] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[71] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13,*

Farmington, PA, USA, November 3-6, 2013, pages 18–32. ACM, 2013.

[72] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.

[73] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 35–53. USENIX Association, 2022.

[74] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 198–216. USENIX Association, 2021.

[75] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1033–1048, 2022.

[76] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 357–372. USENIX Association, 2021.

[77] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[78] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[79] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[80] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[81] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[82] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 511–526. ACM, 2020.

[83] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

[84] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 51–68. USENIX Association, 2022.

[85] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 55–71. USENIX Association, 2022.

[86] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation

Zu-Ming Jiang
Department of Computer Science
ETH Zurich

Zhendong Su
Department of Computer Science
ETH Zurich

Abstract

Database management systems (DBMSs) are crucial for storing and fetching data. To improve the reliability of such systems, approaches have been proposed to detect logic bugs that cause DBMSs to process data incorrectly. These approaches manipulate queries and check whether the query results produced by DBMSs follow the expectations. However, such *query-level manipulation* cannot handle complex query semantics and thus needs to limit the patterns of generated queries, degrading testing effectiveness.

In this paper, we tackle the problem using a fine-grained methodology—*expression-level manipulation*—which empowers the proposed approach to be applicable to arbitrary queries. To find logic bugs in DBMSs, we design a novel and general approach, *equivalent expression transformation (EET)*. Our core idea is that manipulating expressions of a query in a semantic-preserving manner also preserves the semantics of the entire query and is independent of query patterns. EET validates DBMSs by checking whether the transformed queries still produce the same results as the corresponding original queries. We realize our approach and evaluate it on 5 widely used and extensively tested DBMSs: MySQL, PostgreSQL, SQLite, ClickHouse, and TiDB. In total, EET found 66 unique bugs, 35 of which are logic bugs. We expect the generality and effectiveness of EET to inspire follow-up research and benefit the reliability of many DBMSs.

1 Introduction

Database management systems (DBMSs) are critical systems software and play important roles in modern data-driven applications and provide essential functionalities such as data storage and fetching [8, 12, 39]. Like other large-scale systems, DBMSs involve complicated code logic and various functionalities, and thus bugs are easily introduced during their development and maintenance [1, 13, 15]. One of the most critical kinds of bugs is *logic bugs*—the bugs silently cause DBMSs to produce incorrect query results [25–27]. To detect logic bugs, existing approaches generate SQL queries to test DBMSs and check whether the produced results follow the expectations [11, 25–27, 29, 35]. To do so, they either construct customized queries and validate the rows fetched by these queries [27], or transform the given queries and check whether the execution results of the transformed queries are consistent with the original ones [11, 25, 26, 29, 35].

However, all existing approaches have limited generality as they require the generated queries to follow specific patterns. Their generated queries cannot support SQL features that violate their designed query patterns, as shown in Figure 1. For example, PQS [27] requires that the results of the generated queries can be predicted by its manually implemented interpreter, and thus it is difficult for PQS to support advanced SQL features involving complicated calculations (*e.g.,* window functions). TLP [26] requires that the queries must contain predicates in **`WHERE`** or **`HAVING`** clauses for partitioning, while bug-triggering queries may not contain such clauses. In addition, TLP does not support advanced features like window functions and subqueries. DQE [29] limits its queries to only use common SQL features supported by **`SELECT`**, **`UPDATE`**, and **`DELETE`** statements, while any features (*e.g.*, **`JOIN`** operations and aggregate functions) supported by only one kind of statements are absent. Table 1 provides detailed information on whether existing approaches support specific SQL features. Except for our approach, none of the existing ones can encompass all the listed SQL features. Due to their limited support of general SQL queries, existing approaches miss many logic bugs (*e.g.*, the logic bug triggered by the query shown in Figure 2, which incorporate correlated subqueries and join operations).

The lack of generality in existing approaches is caused by the inherent limitations of their coarse-grained methodology, namely *query-level manipulation*. To construct customized queries or transform existing ones to other related queries, these approaches need to understand the semantics of the manipulated queries to guarantee the results produced by these queries follow their expectations. However, SQL queries are designed to be flexible [34], and can contain abundant and

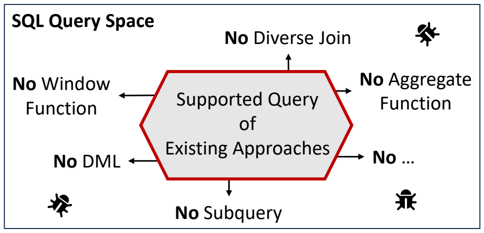


Figure 1: Limitations of existing approaches.

Table 1: Supported SQL features of existing approaches

Approach	Subquery	Join	Window	Group	DML
PQS	○	◐	○	●	○
NoREC	○	●	●	●	○
TLP	○	●	○	●	○
TQS	◐	●	○	●	○
Pinolo	◐	◐	○	○	○
DQE	○	○	○	○	●
EET	●	●	●	●	●

Window: window functions; Group: **GROUP BY** clauses; DML: data manipulation language (*e.g.*, **UPDATE** statements); ◐: TQS [35] and Pinolo [11] support simple but not correlated subqueries. PQS [27] and Pinolo support only parts of join operations.

complex semantics [13, 14, 16, 28, 37] supported by DBMSs. Existing approaches cannot be applied to complex queries because it is difficult to fully understand the complicated semantics contained by such queries (*e.g.*, the queries in Figure 2). Due to these challenges, existing approaches have to limit the patterns of their generated queries to constrain the query semantics. Therefore, these approaches cannot utilize general queries that fall outside their expected query patterns.

To address the inherent limitations of existing approaches, we propose to approach the logic-bug-detection problem using a new and fine-grained methodology—*expression-level manipulation*. Expressions are the essential units of SQL queries. They can be functions, operations, column variables, constant values, or subqueries, *etc*. By manipulating expressions, we can focus on the fine-grained semantics of queries, *i.e.*, expression semantics, and correspondingly manipulate queries without the need to understand their overall query-level semantics. For example, we can easily construct a new query for oracle checking by manipulating the expressions `t2.c2` and `t2.c3` of the original query in Figure 2, even though the query is complex. In this way, we do not need to limit query patterns to ones with simple semantics.

Based on this fine-grained methodology, we propose a novel and general approach, *equivalent expression transformation (EET)*, which applies to arbitrary queries and can effectively find logic bugs in DBMSs. Given an arbitrary query, EET traverses its abstract syntax tree (AST) to iterate over the expressions used in this query. For each expression, EET transforms it to another semantically equivalent one based on logical equivalences [5, 19] and SQL branch structures [34]. In the end, EET compares the execution results of the transformed query (*i.e.*, the query whose expressions have been transformed) and the original query, and any observed discrepancy indicates a logic bug. For example, EET transforms expressions `t2.c2` and `t2.c3` of the original query in Figure 2 into two semantically equivalent **CASE WHEN** expressions of the transformed query, and validate the tested DBMS by checking whether the results of these two queries are identical. The key intuition of what makes this approach effective is that the transformed expressions can lead DBMSs to exercise different code logic for the manipulated queries. Such different query executions cross-check each other as they are expected to produce the same results.

```
--- Statements for database generation
CREATE TABLE t0 (c0 TEXT);
CREATE TABLE t1 (c0 TEXT);
CREATE TABLE t2 (c0 INT4, c1 INT4, c2 TEXT,
                 c3 TEXT, c4 TEXT, c5 TEXT);
INSERT INTO t0 values ('');
INSERT INTO t1 values ('');
INSERT INTO t2 values (1, 2, 'a', 'a', 'a', 'a'),
                      (0, 1, '', '', 'a', 'L');

--- Original query , result set: 0 ✓
SELECT t2.c0 FROM t2
WHERE (t2.c1 >= t2.c0) <> (t2.c5 = (
   SELECT t2.c4 AS c_0
   FROM (t1 AS ref_0 INNER JOIN t0 AS ref_1
      ON (ref_0.c0 = ref_1.c0))
   WHERE t2.c3 = t2.c2
   ORDER BY c_0 DESC LIMIT 1));

--- Transformed query , result set: empty
SELECT t2.c0 FROM t2
WHERE (t2.c1 >= t2.c0) <> (t2.c5 = (
   SELECT t2.c4 AS c_0
   FROM (t1 AS ref_0 INNER JOIN t0 AS ref_1
      ON (ref_0.c0 = ref_1.c0))
   WHERE (CASE WHEN (((ref_0.c0 LIKE 'z~%')
             AND (NOT (ref_0.c0 LIKE 'z~%')))
             AND ((ref_0.c0 LIKE 'z~%') IS NOT NULL))
          THEN t2.c3 ELSE t2.c3 END) =
      (CASE WHEN (((ref_1.c0 NOT LIKE '_%%')
          AND (NOT (ref_1.c0 NOT LIKE '_%%')))
          AND ((ref_1.c0 NOT LIKE '_%%') IS NOT NULL))
      THEN t2.c4 ELSE t2.c2 END)
   ORDER BY c_0 DESC LIMIT 1));
```

Figure 2: Queries exposing an ancient logic bug (20 years old) in PostgreSQL.

We implemented our approach as a practical DBMS testing tool and evaluate it on 5 widely-used and extensively-tested DBMSs, MySQL [20], SQLite [32], PostgreSQL [24], ClickHouse [3], and TiDB [36]. In total, EET found 66 unique bugs, including 16 in MySQL, 10 in SQLite, 9 in PostgreSQL, 21 in ClickHouse, and 10 in TiDB. Among these bugs, 65 are confirmed and 37 are fixed. 35 bugs are logic bugs, and many are long latent. These results demonstrate the effectiveness of EET in finding logic bugs in DBMSs.

Overall, we make the following contributions:

- We propose a fine-grained methodology, *expression-level manipulation*, which can operate on arbitrary queries without limiting query patterns.
- We propose a novel and general approach, *equivalent expression transformation (EET)*, which can effectively find logic bugs in DBMSs using transformation rules based on logical equivalences and SQL branch structures.
- We implement the approach as an automatic DBMS testing tool and evaluate it on 5 widely used DBMSs. In total, we found 66 bugs, 35 of which are logic bugs. To further facilitate research on DBMS testing, we open-source the tool at `https://github.com/JZuming/EET`.

2 Motivation

In this section, we illustrate queries that trigger an ancient logic bug, analyze the limitations of existing approaches, and present our solution based on a fine-grained methodology.

2.1 Illustrative Example

Figure 2 shows the queries that trigger a very ancient logic bug caused by incorrect hash-join mechanisms. The bug existed for 20 years in PostgreSQL until EET found it. The bug-triggering queries consist of three parts. The first part contains several statements (*e.g.*, **`CREATE`** and **`INSERT`**) for setting up a database for later querying. The second part is a randomly generated query, termed as *original query*, and the third part is the query transformed from the original query by our approach, term as *transformed query*. The expressions `t2.c3` and `t2.c2` in the original query are transformed into two semantically equivalent **`CASE WHEN`** expressions in the transformed query. The transformed query preserves the semantics of the original query, and thus these two queries should produce the same results. However, their results differed, indicating a logic bug had been triggered.

The bug-triggering queries have been minimized but are still very complex. Specifically, the transformed query contains a subquery in its **`WHERE`** clause. The subquery is a correlated subquery that references value (*i.e.*, the columns of table `t2`) from the outer query. The subquery uses **`INNER JOIN`** tables in its **`FROM`** clause and uses **`ORDER BY`** and **`LIMIT`** to constrain its returned value. PostgreSQL is expected to return a row `{0}` for the transformed query. However, it returns an empty set because the predicate inside the subquery triggers a logic bug. The detailed root cause is discussed in Section 5.3.

2.2 Limitations of Existing Approaches

Several approaches have been proposed to detect logic bugs in DBMSs [11,25–27,29,35]. PQS [27] synthesizes a query in a way that the query is expected to fetch a specific row. If the row is not fetched, a logic bug is triggered. TLP [26] partitions a given query into three separated queries by decomposing the predicate in the **`WHERE`** or **`HAVING`** clause. The union of the results of these three queries should be consistent with the original one, otherwise, a logic bug is found. Pinolo [11] manipulates the query predicate in its **`WHERE`** clause to construct a new query whose results are the superset or subset of the results of the original query.

All these approaches are trapped in *query-level manipulation*, which is a coarse-grained methodology for logic-bug detection in DBMSs. To guarantee the correct manipulation, this methodology inherently requires the approaches to understand the semantics of the manipulated queries. For example, PQS needs to interpret its synthesized queries to predict their expected results. However, SQL is a flexible query language, providing various features (*e.g.*, subquery, join) to manipulate data [34]. Under specific demands, SQL queries (*e.g.*, analytical queries) can be very complex [16,28,37]. In these cases, query-level manipulation cannot work effectively because it cannot handle the complicated semantics of these queries.

For the example query in Figure 2, PQS cannot infer its expected fetched row because it cannot predict the results of the complex predicates involving correlated subquery with join tables. TLP cannot partition the predicates inside the subqueries, because predicate effects propagating from subqueries to the outer query are implicit and complicated. Partitioning predicates in subqueries cannot guarantee the consistency between the results of its unioned queries and its original query. Similarly, Pinolo cannot guarantee the superset or subset relationships between the manipulated queries because the logical effects inside the subqueries are difficult to predict.

To avoid such inapplicable cases from happening in their generated queries, existing approaches limit the query patterns to constrain the semantics of their generated query. As a result, many important SQL features cannot be supported by existing approaches, as shown in Table 1. For example, PQS and Pinolo support only parts of join operations, and none of the existing approaches supports correlated subqueries, because their semantics are complex. Therefore, these approaches miss many logic bugs that are not covered by their limited query patterns, such as the 20-year-old logic bug of PostgreSQL in Figure 2.

2.3 Our Solution

To propose a general approach that is applicable to arbitrary queries, we need to tackle the logic-bug-detection problem using a new methodology instead of query-level manipulation. In this paper, we propose a fine-grained methodology—

expression-level manipulation, which shifts our focus from the semantics of a whole query to the semantics of a single expression of the query, empowering the potential and flexibility of manipulation. We can manipulate queries by processing their fine-grained elements, expressions, without the necessity to analyze the semantics of the whole query, and thus do not need to limit the query patterns to simplify query semantics.

Based on this methodology, we propose *equivalent expression transformation (EET)*, which is applicable to arbitrary queries to find logic bugs in DBMSs. Given a SQL query, EET iterates all expressions of the query and transforms them into semantically equivalent expressions. EET validates the DBMSs by checking whether the query with transformed expressions produces the same results as the original query. In this paper, we use logical equivalences [5,19] and SQL branch structures [34] to perform semantic-preserving transformation. For example, in Figure 2, EET transform the expressions `t2.c3` and `t2.c2` in the original query to two `CASE WHEN` expressions in the transformed query. `CASE WHEN` expressions are SQL-style conditional branch structures. Depending on the results of the conditional expressions following `WHEN` keywords, the returned values of the `CASE WHEN` expressions are determined by the expression following either the `THEN` keywords (*i.e.*, `TRUE` branches) or `ELSE` keywords (*i.e.*, `FALSE` branches). Both the branch conditions of the two `CASE WHEN` are unsatisfiable and can only be evaluated to `FALSE`. Therefore, these two `CASE WHEN` are semantically equivalent to `t2.c3` and `t2.c2`, respectively, which are the expressions used in the original query. Therefore, the original query and the transformed query should produce the same results. However, the original query outputs 1 row `{0}`, while the transformed query outputs empty, exposing a logic bug.

EET is effective because the transformed expression can result in different execution logic of the tested DBMSs. For the example in Figure 2, the transformed expressions (*i.e.*, the two `CASE WHEN`) lead the PostgreSQL server to invoke its buggy hash-join mechanism, while the original one does not. Such execution differences make PostgreSQL produce different results for the two queries and indicate at least one of the queries triggers bugs.

3 Equivalent Expression Transformation

In this section, we present the formalization and overview of EET, the two kinds of expression transformations that we propose in this paper, and the properties of this approach.

3.1 Overview

We formalize the core idea of *equivalent expression transformation (EET)* as the following formula, where Q represents an arbitrary query, E represents expressions contained in Q, and $DB(Q)$ is the result the tested DBMS produces for Q:

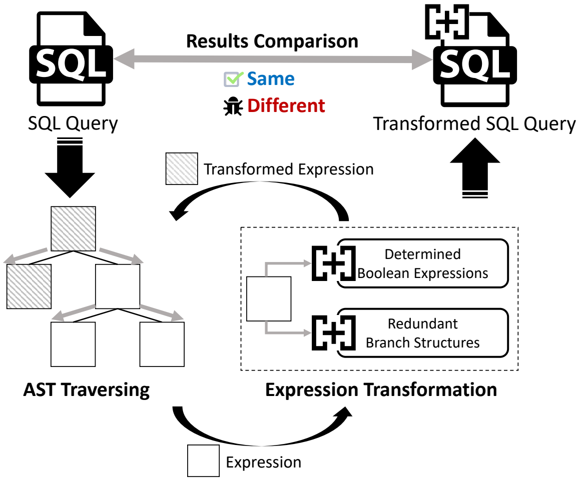


Figure 3: Approach overview of EET.

$$E \equiv E' \Rightarrow DB(Q) \equiv DB(Q'), \text{where } Q' = Q[\![E'/E]\!] \quad (1)$$

The idea is simple: given an arbitrary query Q with expressions E, construct a query $Q' = Q[\![E'/E]\!]$ by replacing all occurrences of E in Q with semantically equivalent expressions E'. Q' and Q are semantically equivalent by construction, and a DBMS should produce identical results on them.

Figure 3 shows the overview of EET. EET traverses the AST presentation of the query to iterate over expressions and transforms these expressions into semantically equivalent ones. After all the expressions have been transformed, the EET constructs a semantically equivalent query and validates the tested DBMS by comparing the results of the transformed query and the original query. In this paper, we propose *determined boolean expressions* (Section 3.2.1) and *redundant branch structures* (Section 3.2.2) to instantiate the expression transformations that satisfy $E \equiv E'$ in Eq. 1.

3.2 Expression Transformation

SQL queries contain various expressions, whose types can be categorized into two classes: boolean expressions and non-boolean expressions. For boolean expressions, we can transform them leveraging the logical equivalences [5,19] in mathematical logic, which have been well studied and generally recognized. For non-boolean expressions, it is difficult to propose general transformation, because these SQL expressions can be numeric (*e.g.*, integer, floating point), string, or timestamp, *etc*. Their execution rules are different. To generally support these types, we leverage SQL branch structures [34], which operate on expressions of various types and provide flexibility for transformation.

To this end, we propose two kinds of expression transformation, *determined boolean expressions* and *redundant branch* *structures*. Table 2 shows the details of each transformation, including its applied expressions and transformation rules. EET is extensible for additional expression transformations, and we expect that more kinds of effective transformation can be proposed in the future, as discussed in Section 3.3.

3.2.1 Determined Boolean Expressions

For each boolean expression, we can transform it using logical operations, such as **AND** and **OR**. We leverage 5 laws of logical equivalences [5, 19] for an arbitrary boolean expression p:

$$\top \vee p \equiv \top$$

$$\bot \wedge p \equiv \bot$$

$$\top \wedge p \equiv p$$

$$\bot \vee p \equiv p$$

$$p \vee q \equiv q \vee p, \quad p \wedge q \equiv q \wedge p$$

We interpret them to corresponding SQL equations:

$$\texttt{TRUE OR } p \equiv \texttt{TRUE} \tag{2}$$

$$\texttt{FALSE AND } p \equiv \texttt{FALSE} \tag{3}$$

$$\texttt{TRUE AND } p \equiv p \tag{4}$$

$$\texttt{FALSE OR } p \equiv p \tag{5}$$

$$p \texttt{ OR } q \equiv q \texttt{ OR } p, \quad p \texttt{ AND } q \equiv q \texttt{ AND } p \tag{6}$$

As the values of SQL boolean expressions can only be either **TRUE**, **FALSE**, or **NULL** [26], one of the expressions, p, **NOT** p, and p **IS NULL** must be **TRUE**, where p is an arbitrary boolean expression. Therefore, p **OR (NOT** p **) OR (** p **IS NULL)** must be **TRUE** according to Eq. 2. Similarly, one of the expressions, p, **NOT** p, and p **IS NOT NULL** must be **FALSE**, so p **AND (NOT** p **) AND (** p **IS NOT NULL)** must be **FALSE** according to Eq. 3. The equations are shown below:

$$p \texttt{ OR } (\texttt{NOT } p) \texttt{ OR } (p \texttt{ IS NULL}) \equiv \texttt{TRUE} \tag{7}$$

$$p \texttt{ AND } (\texttt{NOT } p) \texttt{ AND } (p \texttt{ IS NOT NULL}) \equiv \texttt{FALSE} \tag{8}$$

We use *true_expr* and *false_expr* to represent the expressions on the left-hand side of Eq. 7 and Eq. 8:

$$true\_expr(p) = p \texttt{ OR } (\texttt{NOT } p) \texttt{ OR } (p \texttt{ IS NULL}) \tag{9}$$

$$false\_expr(p) = p \texttt{ AND } (\texttt{NOT } p) \texttt{ AND } (p \texttt{ IS NOT NULL}) \tag{10}$$

Note that the operands of **OR**/**AND** can be randomly disordered according to Eq. 6 (*e.g.*, p and **NOT** p can switch their positions). Combining Eq. 4 with Eq. 7 and 9, Eq. 5 with Eq. 8 and 10, respectively, we get the following equations, where p and p' can be arbitrary boolean expressions:

$$true\_expr(p') \texttt{ AND } p \equiv p \tag{11}$$

$$false\_expr(p') \texttt{ OR } p \equiv p \tag{12}$$

Based on Eq. 11 and Eq. 12, we can transform an arbitrary boolean expression p by adding a structured expression containing a randomly generated boolean expression p'. We accordingly propose two transformation rules shown in rows No.1 and 2 in Table 2. Both of these transformation rules are guaranteed to preserve the semantics of original expressions. The queries shown in Figure 6 and Figure 7 (discussed in Section 5.3) are transformed by these rules.

3.2.2 Redundant Branch Structures

To transform non-boolean expressions, we propose to leverage **CASE WHEN** expressions, which are SQL-style conditional branch structures and support various SQL types. These expressions execute the following if-else logic:

$$C(p, expr_1, expr_2) = \texttt{CASE WHEN } p \texttt{ THEN } expr_1 \\ \texttt{ELSE } expr_2 \texttt{ END}$$

$$C(p, expr_1, expr_2) = \begin{cases} expr_1 & \text{if } p \text{ is } \texttt{TRUE} \\ expr_2 & \text{if } p \text{ is } \texttt{FALSE} \text{ or } \texttt{NULL} \end{cases} \tag{13}$$

We can determine the execution logic of **CASE WHEN** expressions by fixing the predicate p to be **TRUE** or **FALSE**. Furthermore, we can use Eq. 9 and Eq. 10 to replace the **TRUE** and **FALSE** values, making the transformed expression more complex. In the end, we get the following equations:

$$C(true\_expr(p), expr, expr') \equiv expr \tag{14}$$

$$C(false\_expr(p), expr', expr) \equiv expr \tag{15}$$

Based on Eq. 14 and Eq. 15, we can transform an expression to a designed **CASE WHEN** expression, which involves randomly generated expressions p and $expr'$ but still preserves the semantics of the original expression $expr$. We accordingly propose 2 transformation rules shown in rows No.3 and 4 of Table 2. Note that the type of $expr'$ should be the same as the type of $expr$, otherwise ambiguous behaviors may be triggered in some DBMSs [20, 24, 32]. The query shown in Figure 2 is transformed by these rules.

Besides fixing the predicate p to be **TRUE** or **FALSE** to determine the return value of a **CASE WHEN** expression, we can also manipulate the expression in **TRUE** or **FALSE** branch. If the expressions in the **TRUE** and **FALSE** branch are semantically equivalent to each other, the **CASE WHEN** expression is determined to be the expression in the **TRUE/FALSE** branch, no matter how the predicate p is evaluated. Formally, we get the following equation by making $expr_1 \equiv expr_2$ in Eq. 13:

$$expr_1 \equiv expr_2 \Rightarrow C(p, expr_1, expr_2) \equiv expr_1 \equiv expr_2 \tag{16}$$

Based on Eq. 16, we propose 2 transformation rules shown in rows No.5 and 6 of Table 2. In these rules, we deeply copy

Table 2: Transformation rules of EET

No.	Expression Transformation	Applied Expression	Transformation Rule
1	Determined Boolean Expressions	*bool_expr*: boolean	*bool_expr* → *false_expr*[3] **OR** *bool_expr*
2	Determined Boolean Expressions	*bool_expr*: boolean	*bool_expr* → *true_expr*[4] **AND** *bool_expr*

3	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *false_expr*
			THEN *rand_expr(type(expr))*[1, 2]
			ELSE *expr* **END**

4	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *true_expr*
			THEN *expr*
			ELSE *rand_expr(type(expr))* **END**

5	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *rand_expr(boolean)*
			THEN *copy_expr(expr)*[5]
			ELSE *expr* **END**

6	Redundant Branch Structures	*expr*: **CASE-WHEN** applicable	*expr* → **CASE WHEN** *rand_expr(boolean)*
			THEN *expr*
			ELSE *copy_expr(expr)* **END**

7	Origin	*expr*: non boolean **CASE-WHEN** inapplicable	*expr* → *expr*

[1] *type(e)*: the type of the return value of expression *e*.

[2] *rand_expr(t)*: randomly generated expression that returns a value with type *t*.

[3] *false_expr* → *p* **AND (NOT** *p* **) AND (** *p* **IS NOT NULL)** | *p* → *rand_expr(boolean)*

[4] *true_expr* → *p* **OR (NOT** *p* **) OR (** *p* **IS NULL)** | *p* → *rand_expr(boolean)*

[5] *copy_expr(e)*: an expression deeply copied from expression *e*

the original expression *expr* (*i.e.*, the expression itself and its subexpressions) to another expression, namely *copy_expr*. The expressions *expr* and *copy_expr* are semantically equivalent because they are the same. We distribute these expressions in **TRUE** and **FALSE** branches of a **CASE WHEN** expression, and randomly generate a predicate. Eq. 16 guarantees that such a **CASE WHEN** expression is semantically equivalent to the original expression. The query shown in Figure 8 (discussed in Section 5.3) is transformed by these rules.

3.2.3 Choosing Transformation Rules

CASE WHEN expressions can be applied for the majority types of SQL expressions, including numeric types, string types, timestamp types, *e.t.c.* EET randomly chooses one of the rules No.3 to 6 to transform the expression belonging to these types. Boolean types also support **CASE WHEN**, so EET randomly applies one of the rules No.1 to 6 for each boolean expression. However, some types of expressions are **CASE-WHEN** inapplicable, and thus none of the rules No.1 to 6 are available. The table expressions `t0` and `t1` in Figure 2 are the examples. When we replace them with **CASE WHEN** expression, the query will trigger syntactic errors. To address this problem, EET conservatively transforms these expressions to themselves, as shown in rule No.7 of Table 2.

3.3 Properties

Soundness. EET follows the formally-proved equations in Section 3.2, and thus is guaranteed to preserve the semantics of the original queries. If the execution results of the original queries are determined (*i.e.*, excluding SQL features involving randomness), the transformed queries must produce the same execution results as the original ones, otherwise logic bugs are triggered. Therefore, EET is sound and produces no false positives in logic-bug detection.

Generality. EET can be generally applied to various SQL queries because it works at the expression level instead of the query level. Existing approaches, which work at the query level, inherently require the generated queries to follow specified query patterns. Otherwise, these approaches cannot infer the oracle results to validate the execution of their generated queries. Such limitations make existing approaches not general because they cannot be applied to arbitrary queries (*e.g.*, the queries violated their patterns). In contrast, EET can be applied to validate arbitrary queries because it works at the expression level. Given an arbitrary query that is not limited to any query patterns, EET can generally transform it by transforming its expressions, and use the results of the transformed query to validate the results of the given query. In this sense, EET is general in validating arbitrary SQL queries.

While the high-level idea of EET works generally, its im-

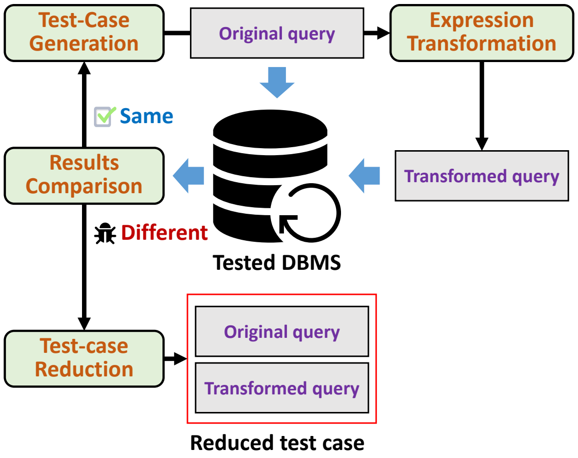


Figure 4: Implementation architecture of EET.

plementation for different SQL dialects can vary, depending on their branching structure syntax. For example, the function **DECODE** in Oracle [21] can return different values according to the comparison of its operands. This function can be implemented in the redundant branch structures of EET to transform queries written in Oracle-style SQL. In this paper, for redundant branch structures, we use only **CASE WHEN** expressions, which are supported by all the tested DBMSs.

Extensibility. In this paper, we propose two kinds of expression transformations for finding logic bugs to demonstrate the effectiveness of EET. Besides these two, we expect that more transformations are possible to enhance the approach. For example, a new transformation can be proposed to process table expressions (*e.g.*, **t0** and **t1** in Figure 2) by joining the original table with other tables while keeping the join results the same as the original one. EET can be easily extended to support new transformations as we only need to specify the transformation rules and the applied expression types. In our implementation, we used less than 200 lines of code for the proposed two kinds of expression transformations.

Black-Box Testing. EET is a pure black-box technique, which does not rely on the internal implementation of the tested DBMSs. Such a property makes our approach portable and can be easily deployed for testing various DBMSs, even the ones whose source code is unavailable.

4 Implementation

We implemented EET into a fully automatic tool on top of SQLsmith [33], which we mainly use for generating databases and complex queries. The overall codebase of the tool is 14k lines of C/C++ code, 2k of which is used to implement our approach. Figure 4 shows its architecture. The following describes the important implementation details.

Test-Case Generation. EET generates databases and queries randomly. To generate a query, the tool incrementally builds an AST tree according to the grammar of SQL [27, 34, 38]. When constructing a node of the AST, EET updates and records the available variables (*e.g.*, relations, columns), and fills the node with a randomly generated expression referencing the available variables. After the AST tree is completed, a new query is generated and can be fed to the tested DBMS.

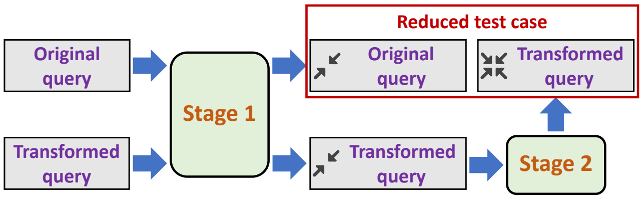


Figure 5: Two-stage reduction of EET.

Expression Transformation. EET leverages the AST representation of the query to iterate each expression. For each expression, EET checks its type and randomly chooses one of the suitable transformation rules, as discussed in Section 3.2.3. During transformation, EET may need to generate additional expressions (*e.g.*, an additional boolean expression is required in rules No.1 to 6 in Table 2). In this case, EET reuses the information (*e.g.*, available variables in corresponding AST nodes) used in test-case generation to randomly generate additional expressions with specific types.

Result Comparison. EET compares the execution results of the transformed query and the original query, including their query output and database changes caused by these queries. Any discrepancy indicates that logic bugs are triggered.

Test-case Reduction. To trigger a logic bug, EET may perform transformations on a large query (*e.g.*, hundreds of lines of SQL), and all its expressions are transformed. However, the same bug might be triggered using parts of the large query and couples of transformed expressions. To ease the burden of developers, we need to reduce the bug-triggering queries before reporting the bugs. We customize *two-stage reduction* for EET to reduce the bug-triggering queries automatically. Figure 5 shows the workflow of this reduction procedure.

In the first stage, EET reduces both the original query and the transformed query. Each time the parts (*e.g.*, an expression) of an original query are reduced, the corresponding parts (*e.g.*, the transformed expression) in the transformed query are also reduced to make these two queries consistent. For example, in Figure 2, when the expression **t2.c3** of the original query is reduced to a constant value **NULL**, the corresponding **CASE WHEN** expression in the transformed query should also be replaced by **NULL**. EET checks whether these two queries still produce different results. If so, the bug still exists and the reduction is effective. Otherwise, EET recovers the reduced parts and tries to reduce other parts of these two queries. When no part can be reduced, EET enters the second stage.

In the second stage, EET tries to incrementally disable the transformation for each expression in the transformed queries. After one transformation for an expression is disabled, if the execution results of the transformed queries are still different from the original, EET keeps this expression not transformed. Otherwise, EET recovers the expression to the transformed version. When no transformation can be disabled in the transformed query, this stage ends.

5 Evaluation

Our evaluation aims to answer the following questions to demonstrate the effectiveness of EET:

Q1 Can EET find real logic bugs in widely used and extensively tested DBMSs? (Section 5.2)

Q2 How diverse are the logic bugs found by EET? (Section 5.3)

Q3 Can EET find logic bugs that are missed by existing approaches? (Section 5.4)

5.1 Experimental Setup

We focused on testing open-source DBMSs for transparent and convenient bug reporting. We chose MySQL [20], PostgreSQL [24], SQLite [32], ClickHouse [3], and TiDB [36] because they are popular and extensively tested. At the time of paper writing, MySQL, PostgreSQL, and SQLite rank 1st, 2nd, and 6th, respectively, among the open-source DBMSs according to their popularity in DB-Engines Ranking [6]. ClickHouse and TiDB are relatively new DBMSs but very popular on GitHub [10]. They have gained over 31K and 35K stars, respectively, demonstrating their popularity. All of these DBMSs have been extensively tested by existing approaches for finding logic bugs [4, 11, 25–27, 29, 35] and crash bugs [13, 33, 38]. Finding new bugs in these DBMSs is very challenging and can demonstrate the effectiveness of EET.

We use EET to test the latest version of each DBMS. When the code of a DBMS is updated, we start a new test for the updated version. Specifically, we started to test MySQL from version 8.0.34, PostgreSQL from commit `3f1aaaa`, SQLite from commit `20e09ba`, ClickHouse from commit `30464b9`, and TiDB from commit `f5ca27e`. All the DBMS code is cloned from their official GitHub repositories. We intermittently deployed EET to test these DBMSs. We stopped and restarted the testing when we implemented new features in EET. The overall testing duration is three months. We evaluate EET on Ubuntu 20.04 with a 64-core AMD Epyc 7742 CPU at 2.25G Hz and 256GB RAM.

5.2 Bug Detection

As shown in Table 3, we reported 66 unique DBMS bugs found by EET, including 16 in MySQL, 9 in PostgreSQL, 10 in SQLite, 21 in ClickHouse, and 10 in TiDB. 65 of these bugs have been confirmed, and 37 have been fixed. None of these bugs are marked as duplicates by developers.

Table 3: Status of the bugs found by EET

DBMS	**Reported**	**Confirmed**	**Fixed**
MySQL	16	16	2
PostgreSQL	9	9	8
SQLite	10	10	10
ClickHouse	21	20	15
TiDB	10	10	2
Total	66	65	37

Table 4: Bug classification

DBMS	**Logic**	**Crash**	**Error**
MySQL	10	6	0
PostgreSQL	3	3	3
SQLite	9	0	1
ClickHouse	11	3	7
TiDB	2	7	1
Total	35	19	12

Bug Classification. We classify the bugs EET found into the three following types:

Logic bugs. The tested DBMSs incorrectly execute the SQL queries and produce wrong results (*e.g.*, select or update incorrect rows). These bugs were exposed because they incurred discrepancies between the results of the original queries and the transformed queries generated by EET.

Crash bugs. These bugs cause the tested DBMSs to crash or panic when specific queries are processed. Their root causes may be (1) memory corruption like null-pointer dereference; (2) assertion failures and (3) unexpected memory exhaustion.

Abnormal errors. The tested DBMSs report unexpected errors (*e.g.*, "database disk image is malformed" in SQLite) when processing syntactically and semantically valid queries.

As shown in Table 4, 35 bugs (52% of the bugs EET found) are logic bugs, which are the most interesting and hard-to-find bugs. The three logic bugs in PostgreSQL are exciting because PostgreSQL is a well-known hard nut for DBMS testing [13, 25, 27], where SQLancer [30], the DBMS testing tool integrating three logic bug detection techniques [25–27], found only one logic bug [31]. EET also found 19 crash bugs and 11 abnormal errors. These results demonstrate that EET is effective in finding bugs in DBMSs, especially logic bugs.

Bug Importance. We collect the severity information of the bugs we reported to MySQL and TiDB. PostgreSQL, SQLite, and ClickHouse do not provide the severity of each reported bug. All the crash bugs we reported to MySQL were identified as *confidential*, among which 2 have been assigned CVEs.

Among the 10 logic bugs in MySQL, 7 were recognized as *serious* bugs, and 3 were *non-critical*. Among the 10 bugs reported to TiDB, developers marked 6 as *major*, 1 as *minor*, and 3 as *moderate*.

The developers appreciated our effort in finding real bugs in their DBMSs. Particularly, PostgreSQL developers recognized our contribution to the reliability of PostgreSQL and sent us their commemorative coin [23]. ClickHouse and PostgreSQL developers provided their testimonials:

> ***ClickHouse:*** This tool has proven its value, and we want to integrate it into our CI and use it continuously. Thanks to @xxx for running it and reporting the findings!

> ***PostgreSQL:*** Thanks for your efforts! I thought about the generation of self-join tests for about a year, and it would be interesting to read about your approach. Could you send me a copy of the paper after release? Or the name of the conference to participate and see it offline.

Throughput. We count the number of tests (each one consists of one original query and one transformed query) performed by EET per second during testing the 5 DBMSs. On average, a single EET instance performs 3.39 tests per second (293k tests per day), which is lower than existing approaches. It is reasonable because EET supports complex queries, and DBMSs spend much more time executing complex queries than simple queries [13], making most CPU time spent on query execution (94.18% in our statistic results). We believe this throughput is practical considering (1) DBMS testing typically persists for several months [25–27, 29], and thus a sufficient number of tests can be executed and (2) setting up multiple testing instances can significantly improve the test efficiency of EET.

Table 5: SQL features of the 35 queries triggering logic bugs

ID	DBMS	Subquery	Join	Window	Group	DML
1	MySQL	●	●	○	○	○
2	MySQL	○	○	○	○	○
3	MySQL	○	○	○	○	○
4	MySQL	●	●	○	○	○
5	MySQL	●	○	●	○	○
6	MySQL	○	○	○	○	○
7	MySQL	○	○	○	○	○
8	MySQL	●	○	●	○	○
9	MySQL	○	○	○	○	○
10	MySQL	○	○	○	○	○
11	PostgreSQL	○	●	○	●	○
12	PostgreSQL	●	●	○	○	○
13	PostgreSQL	●	●	○	○	○
14	SQLite	●	●	○	○	○
15	SQLite	●	●	○	○	○
16	SQLite	●	○	○	○	●
17	SQLite	●	○	○	○	●
18	SQLite	●	●	●	○	○
19	SQLite	●	●	○	○	○
20	SQLite	○	●	○	○	○
21	SQLite	○	●	○	●	○
22	SQLite	○	●	○	○	○
23	ClickHouse	○	○	○	○	○
24	ClickHouse	●	●	○	○	○
25	ClickHouse	○	○	○	○	○
26	ClickHouse	●	●	○	○	○
27	ClickHouse	●	●	○	●	○
28	ClickHouse	●	○	●	○	○
29	ClickHouse	○	○	○	○	○
30	ClickHouse	●	○	○	○	○
31	ClickHouse	○	○	○	○	○
32	ClickHouse	○	●	○	○	○
33	ClickHouse	○	●	○	○	○
34	TiDB	●	○	○	○	○
35	TiDB	○	●	○	○	○

5.3 Bug Diversity

For the 35 logic bugs, we investigate their diversity from three aspects: (1) the diversity of bug-triggering queries involving multiple SQL features, (2) the diversity of the root causes of why DBMS produces incorrect results, and (3) the diversity of bug manifestation during testing DBMSs.

SQL Features. Table 5 lists the SQL features used in the 35 queries triggering logic bugs. The columns *Subquery*, *Join*, *Window*, and *Group* show whether the queries contain subqueries, join operations, window functions, and **`GROUP BY`** clauses, respectively. The column *DML* shows whether the bug-triggering queries are DML statements (*e.g.*, **`UPDATE`** and **`DELETE`**) instead of DQL (*e.g.*, **`SELECT`**).

Among the 35 bug-triggering queries, 18 contain subqueries (8 of them involve correlated subqueries), 18 use join operations (*e.g.*, inner join, outer join, and cross join), 4 invoke window functions (*e.g.*, **`DENSE_RANK`**, **`FIRST_VALUE`**), 3 involve **`GROUP BY`** clauses, and 2 are DML statements. We investigated the 10 bugs not involving these 5 features and found that all of them used SQL functions (*e.g.*, **`ACOS`**, **`HEX`**, and **`UNIX_TIMESTAMP`**) to perform complicated value calculations, string manipulation, and timestamp controlling. These results indicate that EET can find logic bugs triggered by various SQL queries.

The combined results in Table 1 and Table 5 demonstrate that EET can find many logic bugs missed by existing approaches because EET can support more various SQL features. For example, PQS [27], TLP [26], and NoREC [25] cannot find the 18 logic bugs related to subqueries, which are not supported by these approaches. Lacking support for join operations, DQE [29] cannot find the 18 join-related logic

```
--- Statements for database generation
CREATE TABLE t0 (c0 INT , c1 INT, c2 INT);
INSERT INTO t0 VALUES(2,1,-20);
INSERT INTO t0 VALUES(2,2,NULL);
INSERT INTO t0 VALUES(2,3,0);
INSERT INTO t0 VALUES(8,4,95);
--- Original query , delete 4 rows ✓
DELETE FROM t0 WHERE TRUE;
--- Transformed query , delete 3 rows 🐞
DELETE FROM t0 WHERE
((((t0.c0 <= t0.c2) AND
    (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
    OFFSET 2))) IS NULL) OR
((t0.c0 <= t0.c2) AND
    (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
    OFFSET 2))) OR
NOT ((t0.c0 <= t0.c2) AND
    (t0.c0 <> (SELECT c0 FROM t0 ORDER BY c0 LIMIT 1
    OFFSET 2))))AND TRUE;
```

Figure 6: Queries triggering a logic bug in the one-pass optimization of SQLite.

bugs. Notably, none of the existing approaches could find the 8 logic bugs related to correlated subqueries, as they cannot support semantically complex features. EET, benefiting from expression-level manipulation, is not limited to specific query patterns and can generally support all the listed features. Therefore, EET can find many bugs beyond the capabilities of existing approaches.

Root Cause Analysis. We investigated the 19 fixed logic bugs, whose patches and developer feedback are visible. They consist of 9 bugs in SQLite, 3 in PostgreSQL, and 7 in ClickHouse. We found that 12 bugs are caused by incorrect optimization. It is expected because EET supports logic bug detection for complex queries, which has huge potential to be optimized and thus can cover many optimization mechanisms in the tested DBMSs. 11 bugs are related to **JOIN** operations. Indeed, existing approaches could not systematically test the DBMS components related to **JOIN** operations until TQS was proposed [35], and thus many bugs remain unexposed. These results indicate that EET can also be used to effectively test the **JOIN** mechanism implemented in the tested DBMSs (*e.g.*, the hash-join bug shown in Figure 2). Different from TQS, EET can also detect bugs in other DBMS components, such as a bug in the JIT components used for expression compilation in ClickHouse. The following shows three interesting bug examples caused by different root causes.

Example 1: Optimization bug in SQLite. Figure 6 shows the queries triggering a logic bug in the one-pass optimization of SQLite. The original query is a simple **DELETE** statement with a predicate **TRUE**, which removes 4 rows in the table **t0**. EET transforms this query by applying rule No.2 in Table 2 to the predicate **TRUE**, which is semantic preserving. The transformed query should also remove 4 rows in **t0**, but only 3 rows are removed, indicating a logic bug triggered. The root cause is that the transformed query triggered the one-pass optimization in SQLite, which passes the target table only one time. For each row, SQLite evaluates whether it satisfies the predicate. If so, SQLite deletes the row. Because the subquery in the **WHERE** clause is behind a short-circuit operator (*i.e.*, **AND** operation), SQLite evaluates it after one or more rows have already been deleted, and SQLite thus produces a wrong result for the subquery, which at the end makes a row in table **t0** not deleted. SQLite developers fix this bug by disabling the one-pass optimization when the processed query contains a subquery in its **WHERE** clause.

```
--- Statements for database generation
CREATE TABLE t0 (c0 UInt32, c1 UInt32,
        PRIMARY KEY (c0)) ENGINE = MergeTree;
INSERT INTO t0 VALUES (2, 2);
--- Original query , result set: {FALSE} ✓
SELECT FALSE FROM t0;
--- Transformed query , result set: {TRUE} 🐞
SELECT (acos(c0) <> atan(c1)) AND
         (NOT (acos(c0) <> atan(c1))) AND
         ((acos(c0) <> atan(c1)) IS NOT NULL)
     OR FALSE from t0;
```

Figure 7: Queries triggering a logic bug in the JIT compilation of ClickHouse.

Example 2: JOIN bug in PostgreSQL. Figure 2 shows the query triggering an ancient logic bug in the hash-join implementation of PostgreSQL. Specifically, PostgreSQL built a hash table for the **INNER JOIN** tables (*i.e.*, **t1** and **t0**) in the **FROM** clause. Their hash table was affected by a PostgreSQL data structure, *Param*, a parameter used for passing values into and out of subqueries or from nested loop joins to their inner scans. PostgreSQL must rebuild the hash table when specific Param values are updated. However, in some cases, the inner hash-key expressions for the hash table reference some Params whose changes are unexpectedly missed by PostgreSQL. The transformed query in Figure 2 updated such Params while PostgreSQL did not perceive the changes of these Params, as a result of which PostgreSQL incorrectly reused the outdated hash table and produced wrong results. PostgreSQL developers fixed this bug by invoking specific functions to take the missed Params into account.

Example 3: JIT bug in ClickHouse. Figure 7 shows the query triggering a logic bug in the JIT compilation of ClickHouse. The original query is a simple **SELECT** statement. EET transforms the **FALSE** expression in the **SELECT** clause, applying rule No.1 in Table 2. The transformed query trigger the JIT compilation of ClickHouse because the query repeatedly uses the expression **acos(c0)<> atan(c1)**. To speed up the query execution, ClickHouse compiles this expression into machine code that can be executed by CPUs directly. However, the JIT compiler incorrectly compiled the non-equal operation (*i.e.*, **<>**), and thus the machine code produced wrong

```
--- Statements for database generation
CREATE TABLE t0 (c0 INT, c1 INT);
CREATE TABLE t1 (c0 INT, c1 INT, c2 REAL,
                 c3 REAL, c4 INT);
INSERT INTO t0 VALUES(14, 24000);
INSERT INTO t1 VALUES(85, 95000, 97.87, 0.0, 0);
--- Original query , result set: {1}  🐞
SELECT DISTINCT 1 AS c1
FROM ((t1 AS ref_0 RIGHT OUTER JOIN t0 AS ref_1
        ON ref_0.c4 = ref_1.c0)
   LEFT OUTER JOIN
      (t1 AS ref_2 LEFT OUTER JOIN t0 AS ref_3
        ON ref_2.c1 = ref_3.c0)
   ON (((SELECT c1 FROM t0 ORDER BY c1 LIMIT 1) IN (
     SELECT ref_4.c0 AS c0 FROM t1 AS ref_4)) IS TRUE))
WHERE ref_2.c3 <= ref_2.c2;
--- Transformed query , result set: empty ✓
SELECT DISTINCT 1 AS c1
FROM ((t1 AS ref_0 RIGHT OUTER JOIN t0 AS ref_1
        ON ref_0.c4 = ref_1.c0)
   LEFT OUTER JOIN
      (t1 AS ref_2 LEFT OUTER JOIN t0 AS ref_3
        ON ref_2.c1 = ref_3.c0)
   ON (((SELECT c1 FROM t0 ORDER BY c1 LIMIT 1) IN (
     SELECT ref_4.c0 AS c0 FROM t1 AS ref_4)) IS TRUE))
WHERE CASE WHEN TRUE THEN ref_2.c3 <= ref_2.c2
            ELSE ref_2.c3 <= ref_2.c2 END;
```

Figure 8: SQLite logic bug triggered by the original query.

results when comparing the **NaN** value returned from **acos (c0)**. Therefore, the transformed query produces an unreasonable result. The developers fix this bug by repairing the function responsible for compiling the non-equal operation.

Bug Manifestations. While analyzing the 19 fixed logic bugs, we found another interesting phenomenon: a logic bug can be triggered by either the transformed query, the original query, or both of them. Specifically, 10 of the 19 bugs are triggered by the transformed queries (*e.g.*, Example 1-3), while 8 bugs are triggered by the original queries (*e.g.*, Example 4). Interestingly, EET found a logic bug triggered by both the transformed query and the original query (*i.e.*, Example 5), whose results are different. These results demonstrate that EET can catch a logic bug if any discrepancy is incurred between their result, independently of which query is the culprit.

Example 4: Bug triggered in the original queries. Figure 8 shows a case where the original query triggers a bug in SQLite. The original query contains the **DISTINCT** keyword, and its **FROM** clause consists of multiple join tables with many join conditions specified in the corresponding **ON** clauses, while the predicate in the **WHERE** clause is a simple comparison expression. In this case, SQLite applies the omit-outer-join optimization, which can reduce the useless join tables (*e.g.*, **ref_3**) that are not referenced outside their **JOIN** expressions. However, this optimization did not work well when SQLite also flattened the subqueries in the **JOIN** expressions (*e.g.*, the two subqueries in the last **ON** clause). As a result, SQLite incorrectly reduced the tables consisting of the flattened subqueries and thus produced wrong results for the original query. EET transforms the predicate in the **WHERE** clause of the original query to a **CASE WHEN** expression. Such transformation makes the predicate complex and blocks the buggy omit-outer-join optimization, and thus SQLite produced correct results for the transformed query. SQLite developers fix this bug by adding restrictions for applying omit-outer-join optimization.

```
--- Statements for database generation
CREATE TABLE t0 (c0 TEXT);
CREATE TABLE t1 (c0 INT4, c1 INT4, c2 TEXT,
                 c3 INT4, c4 FLOAT8, c5 INT4);
CREATE TABLE t2 (c0 TEXT, c1 TIMESTAMP);
CREATE VIEW t3 AS
 SELECT '1' AS c_0
   FROM ((SELECT ref_0.c0 AS c_0 FROM t0 ref_0
          GROUP BY ref_0.c0) subq_0
     FULL JOIN t2 ref_1 ON (subq_0.c_0 = ref_1.c0))
  WHERE ref_1.c1 > ref_1.c1;
CREATE VIEW t4 AS
 SELECT ref_1.c5 AS c_2, ref_1.c4 AS c_3,
           ref_1.c1 AS c_4, 1 AS c_6, ref_1.c3 AS c_9
   FROM (t3 ref_0 RIGHT JOIN t1 ref_1
           ON (ref_0.c_0 = ref_1.c2));
INSERT INTO t1 VALUES (11000, 0, null, 0, 0.0, 15);
--- Original query , result set: {0}  🐞
SELECT COUNT(*) AS c_6
FROM (t1 AS ref_0 LEFT OUTER JOIN t4 AS ref_1
        ON (ref_0.c0 = ref_1.c_2))
WHERE ref_1.c_3 =
    DCBRT(CASE WHEN ref_0.c2 LIKE '7%z'
            THEN ref_1.c_6 ELSE ref_0.c4 END);
--- Transformed query , result set: {1}  🐞
SELECT COUNT(*) AS c_6
FROM (t1 AS ref_0 LEFT OUTER JOIN t4 AS ref_1
        ON (ref_0.c0 = ref_1.c_2))
WHERE (CASE WHEN ((ref_1.c_9 >= ref_1.c_4)
           OR (NOT (ref_1.c_9 >= ref_1.c_4))
           OR ((ref_1.c_9 >= ref_1.c_4) IS NULL))
      THEN ref_1.c_3 ELSE ref_1.c_3 END) =
    DCBRT(CASE WHEN ref_0.c2 LIKE '7%z'
            THEN ref_1.c_6 ELSE ref_0.c4 END);
```

Figure 9: PostgreSQL logic bug triggered by both the original query and the transformed query.

Example 5: Bug triggered in both two queries. Figure 9 shows the test case where both the original query and the transformed query trigger the same logic bug in PostgreSQL. EET transforms the expression **ref_1.c_3** in the **WHERE** clause of the original query to a **CASE WHEN** expression, which unexpectedly makes the transformed query return different results. We reported the test case to PostgreSQL developers, who confirmed that both the two queries in the test case triggered a bug according to their query plans. Figure 10 shows their query plans. Their query plans are nearly the same, and the only difference is that the original query used a hash join, while the transformed query used a hash right join. Both query plans are problematic because they lose join filters, which are responsible for filtering the rows that satisfy the

```
QUERY PLAN: Original Query / Transformed Query
Aggregate
|-- Hash Join / Hash Right Join
      Hash Cond: (ref_1.c5 = ref_0.c0)
MISSING: Join Filter: (ref_1.c4 / CASE... = dcbrt(...))
   |-- Nested Loop Left Join
         Join Filter: ('1'::text = ref_1.c2)
      |-- Seq Scan on t1 ref_1
      |-- Materialize
         |-- Seq Scan on t2 ref_1_1
               Filter: (c1 > c1)
   |-- Hash
      |-- Seq Scan on t1 ref_0
```

Figure 10: Query plans of the original query (using hash join) and the transformed query (using hash right join) in Figure 9.

predicate in the **WHERE** clauses. The root cause of this logic bug is that PostgreSQL removed some unnecessary **LEFT JOIN** tables that are underneath other **LEFT JOIN** but failed to clean the data structures referencing the removed tables, causing PostgreSQL to consider that the join filters affected by such data structures are unreasonable and drop them. After the bug is fixed, the query plans of both the origin query and the transformed query contain their corresponding join filters.

5.4 Comparative Study

To check whether EET can find logic bugs missed by existing approaches, we investigate the earliest bug-inducing versions of the 35 logic bugs EET found and check whether these versions are before the existing approaches got published. We would conclude that EET can find logic bugs missed by existing approaches if some long-latent bugs are found by EET. This comparison is reasonable and objective because: (1) all logic bugs found by EET are not marked as duplicates by developers, meaning that no approach found these bugs until EET found them; (2) all the 5 DBMSs in our evaluation have been extensively tested by existing approaches [4,11,25–27,29,35], meaning that the existing approaches did not find the long-latent bugs found by EET during their evaluation.

We classify the existing approaches according to the years they got published, resulting in 2 classes: 2020 (PQS [27], TLP [26], NoREC [25]) and 2023 (TQS [35], Pinolo [11], DQE [29]). Therefore, we check whether the versions inducing the logic bugs found by EET are before 2020 and 2023. Table 6 shows the results.

Among 35 logic bugs found by EET, 13 already existed before 2020 (*i.e.*, bugs were involved in 2019 or earlier), indicating that all the existing approaches miss these 13 logic bugs in their extensive evaluation, as all these approaches are proposed in or after 2020. In addition, 11 logic bugs can be triggered between 2020 and 2022, while none of the three approaches (*i.e.*, TQS, Pinolo, and DQE) published in 2023 found these bugs. These results indicate that existing approaches indeed miss many long-latent logic bugs. Due to the inherent limitations of query-level manipulation, these approaches have to limit the patterns of their generated queries and thus cannot be applied to complex queries (*e.g.*, the bug-triggering queries in Figure 2), while many DBMS bugs can be triggered only by complex queries [13]. Empowered by expression-level manipulation, EET can be easily applied to complex queries and thus successfully found many logic bugs missed by existing approaches.

Table 6: Latency of the logic bugs found by EET

DBMS	Found	Bug-involved year		Longest latency
		< 2023	< 2020	
MySQL	10	9	6	6 years
PostgreSQL	3	1	1	20 years
SQLite	9	5	4	8 years
ClickHouse	11	7	2	4 years
TiDB	2	2	0	3 years
Total	35	24	13	20 years

We investigate the longest latency of the logic bugs found by EET for each tested DBMS. The result is shown in Table 6. Interestingly, for each tested DBMS, EET can find at least one bug whose latency is longer than 3 years. The bug with the longest latency, *i.e.*, 20 years, is found in PostgreSQL, which is shown in Figure 2. These results demonstrate that EET can effectively find long-latent bugs.

While EET can find many logic bugs missed by existing approaches, it may inherently miss some bugs that can be found by existing approaches. For example, if a logic bug makes both the original query and the transformed query produce the same incorrect results, EET will miss this bug. However, approaches like PQS [27] can help detect such missed bugs by inferring the expected query results. One interesting future work is to integrate EET and existing approaches into a testing framework that can efficiently schedule these approaches during testing DBMSs.

6 Related Work

Logic Bug Detection in DBMSs. Several approaches have been proposed to detect logic bugs in DBMSs [11,25–27,29,35]. PQS [27] synthesizes a query that guarantees to return a specific row using its manually implemented interpreter. If the tested DBMS fails to fetch the row, PQS identifies a logic bug. NoREC [25] generates a query with a predicate and transforms this query by moving its predicates from its **WHERE** clause to its **SELECT** clause. NoREC identifies a logic bug if the moved predicate produces different results. TLP [26] partitions an original query into three separated queries by decomposing its predicate. The union of the results of these separated queries should be consistent with the result of the original one, otherwise, a logic bug is triggered. SQLancer [30] integrates the above three techniques and has been deployed to test various DBMSs. In addition, DQE [29] generates different types of queries (*e.g.*, **SELECT**, **UPDATE**, and **DELETE**) with the same predicates. These different queries should operate the same rows, otherwise, a logic bug is triggered. Pinolo [11] modifies the predicate of a query, making its constraints looser/stricter. Therefore, the modified query should return a superset/subset of the results of the original query, otherwise, Pinolo identifies a logic bug. TQS [35] applies schema normalization [7, 22] to slit a wide table into multiple tables and construct a customized query using join tables, whose ground truth results can be inferred from the original wide table. Based on these ground truth results, TQS finds many logic bugs related to **JOIN** operations.

As discussed in Section 1 and 2, all these approaches are based on *query-level manipulation*, which requires the approaches to understand the semantics of the manipulated queries. Therefore, existing approaches cannot be applied to complex queries whose semantics are complicated. Different from existing approaches, EET is based on *expression-level manipulation*, which operates on expressions and has no necessity to understand the query semantics. Therefore, EET does not need to limit the query patterns and thus can be applied to various queries.

DBMS Test-Case Generation. Without focusing on logic bugs, some approaches [9, 13, 17, 33, 38] are proposed to generate more diverse test cases for DBMSs. SQLsmith [33] is a grammar-based DBMS fuzzer, which embeds the AST rules of SQL language and can generate complex SQL statements. SQUIRREL [38] proposes a new intermediate representation to model SQL queries and infers the dependencies between statements. In this way, SQUIRREL can generate queries that contain multiple SQL statements. Griffin [9] performs grammar-free mutation to test DBMSs by summarizing the DBMS state information into its metadata graph and mutating SQL queries according to the graph to prevent semantic errors. DynSQL [13] performs dynamic query interaction to capture the latest DBMS state information and incrementally generate complex and valid queries. In addition, some approaches [2, 18] are proposed to improve the test-case generation for logic bug detection. SQLRight [18] enables code coverage feedback, which gives the supported test oracles more chance to find logic bugs in rarely executed DBMS code. QPG [2] records the covered query plans during testing DBMSs and prioritizes mutating the queries that trigger new query plans, which is more likely to expose a new logic bug.

These approaches and EET can complement each other. On one hand, the high-quality and diverse queries generated by these approaches can help EET find more logic bugs hidden in the corner logic of DBMSs. On the other hand, the general test oracle provided by EET can enable these approaches to catch more various bugs.

7 Conclusion

In this paper, we propose a new and fine-grained methodology, *expression-level manipulation*, for approaching logic-bug-detection problems in DBMSs without limiting the query patterns. Based on this methodology, we propose a novel and general approach, *equivalent expression transformation (EET)*, which can effectively find logic bugs using two transformations: determined boolean expressions and redundant branch structures. We evaluate EET on 5 widely used DBMSs. In total, EET found 66 bugs, 35 of which are logic bugs. Many of these logic bugs have long latency and are missed by existing approaches. We believe the generality and effectiveness of EET can inspire more follow-up research on DBMS testing.

Acknowledgments

We thank the anonymous OSDI reviewers for their valuable feedback on earlier versions of this paper. We also thank the DBMS developers for triaging and fixing our reported bugs.

References

[1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[2] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 2060–2071, 2023.

[3] ClickHouse. https://clickhouse.com/.

[4] Using SQLancer to test ClickHoouse and other DBMS. https://presentations.clickhouse.com/?path=heisenbug2021.

[5] Irving M Copi, Carl Cohen, and Victor Rodych. *Introduction to logic*. Routledge, 2018.

[6] DB-Engines Ranking, Accessed in October, 2023. https://db-engines.com/en/ranking.

[7] Jim Diederich and Jack Milton. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)*, 13(3):339–365, 1988.

[8] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. In *Proceedings of the 1998 International Conference on Management of Data (SIGMOD)*, pages 59–74, 1998.

[9] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th International Conference on Automated Software Engineering (ASE)*, pages 1–12, 2022.

[10] Github. https://github.com/.

[11] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In Julia Lawall and Dan Williams, editors, *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, pages 345–358, 2023.

[12] Jan L Harrington. *Relational database design and implementation*. Morgan Kaufmann, 2016.

[13] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[14] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. Detecting transactional bugs in database engines via graph-based oracle construction. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 397–417, 2023.

[15] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: automatic detection and diagnosis of performance regressions in database systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, pages 57–70, 2019.

[16] Raghav Kaushik and Ravi Ramamurthy. Efficient auditing for complex sql queries. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD)*, pages 697–708, 2011.

[17] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. Sequence-oriented dbms fuzzing. In *Proceedings of the 2023 International Conference on Data Engineering (ICDE)*, 2023.

[18] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of DBMS with coverage-based guidance. In *Proceedings of the 31st USENIX Security Symposium*, pages 4309–4326, 2022.

[19] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 2009.

[20] MySQL. https://www.mysql.com/.

[21] Oracle Database. https://www.oracle.com/database/.

[22] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß, editors, *Proceedings of the 20th Internationa Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017.

[23] PostgreSQL contributor gifts. https://wiki.postgresql.org/wiki/Contributor_Gifts.

[24] PostgreSQL. https://www.postgresql.org/.

[25] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE)*, pages 1140–1152, 2020.

[26] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. In *Proceedings of the 2020 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.

[27] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.

[28] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. Athena++ natural language querying for complex nested sql queries. *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, pages 2747–2759, 2020.

[29] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. Testing database systems via differential query execution. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 2072–2084, 2023.

[30] SQLancer. https://github.com/sqlancer/sqlancer.

[31] DBMS bugs found by SQLancer. https://www.manuelrigger.at/dbms-bugs/.

[32] SQLite. https://www.sqlite.org/index.html.

[33] SQLsmith: a random sql query generator. https://github.com/anse1/sqlsmith.

[34] SQL standard. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[35] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. Detecting logic bugs of join optimizations in dbms. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*, pages 1–26, 2023.

[36] TiDB. `https://www.pingcap.com/tidb/`.

[37] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the 2000 International Conference on Management of Data (SIGMOD)*, pages 105–116, 2000.

[38] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS)*, pages 955–970, 2020.

[39] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2020.

Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs

Tony Nuda Zhang
University of Michigan

Travis Hance
Carnegie Mellon University

Manos Kapritsos
University of Michigan

Tej Chajed
University of Wisconsin–Madison

Bryan Parno
Carnegie Mellon University

Abstract

Proving the correctness of a distributed protocol is a challenging endeavor. Central to this task is finding an *inductive invariant* for the protocol. Currently, automated invariant inference algorithms require developers to describe protocols using a restricted logic. If the developer wants to prove a protocol expressed without these restrictions, they must devise an inductive invariant manually.

We propose an approach that simplifies and partially automates finding the inductive invariant of a distributed protocol, as well as proving that it really is an invariant. The key insight is to identify an *invariant taxonomy* that divides invariants into Regular Invariants, which have one of a few simple low-level structures, and Protocol Invariants, which capture the higher-level host relationships that make the protocol work.

Building on the insight of this taxonomy, we describe the Kondo methodology for proving the correctness of a distributed protocol modeled as a state machine. The developer first manually devises the Protocol Invariants by proving a *synchronous* version of the protocol correct. In this simpler version, sends and receives are replaced with atomic variable assignments. The Kondo tool then automatically generates the asynchronous protocol description, Regular Invariants, and proofs that the Regular Invariants are inductive on their own. Finally, Kondo combines these with the synchronous proof into a *draft* proof of the asynchronous protocol, which may then require a small amount of user effort to complete. Our evaluation shows that Kondo reduces developer effort for a wide variety of distributed protocols.

1 Introduction

Distributed protocols are notoriously difficult to reason about. Because they underpin critical applications and infrastructure, any bugs can have severe consequences. Hence, in recent years, both researchers [14,20,25,37,38,41,45] and practitioners [2,30,46] have turned to formal verification to rigorously prove the correctness of distributed systems and protocols.

At the core of a formal distributed protocol safety proof is an *inductive invariant*, which implies that a desired safety property holds throughout a system's execution. As argued in previous work [13,26,43,44], manually deriving inductive invariants is a creative challenge. For example, the IronFleet [14,15] authors reported spending months to identify and prove an inductive invariant for Paxos [22,23].

Unsurprisingly, this challenge spurred a new category of algorithms and tools to automatically find the inductive invariants of distributed protocols [10,13,17–19,26,33,43,44]. For instance, DuoAI [43] finds an inductive invariant for Paxos in minutes, without any user guidance.

However, automated invariant inference has its own Achilles' heel—it limits how developers may express their protocols. State-of-the-art tools like DuoAI only work when the problem of checking the correctness of inductive invariants is a decidable one. Thus, they apply only to protocols that operate within the confines of a first-order logic fragment known as *effectively propositional reasoning* (EPR [34]). As an example of its limited expressivity, EPR does not permit arithmetic, requiring developers to use creative abstractions to encode common systems primitives such as epoch numbers and vote counting. As detailed by Padon et al. [31], expressing a protocol such as Paxos in EPR is quite challenging.

In summary, the current state of the art is a landscape of two extremes: the developer has to choose between 1) expressing the protocol naturally using standard programming primitives such as arithmetic, but manually find the inductive invariant, or 2) abstracting the protocol into the restrictive confines of EPR so as to apply automated invariant finding tools.

In this work, we present a new approach to bridge this gap. Our key insight is that there is an *invariant taxonomy* in the clauses within an inductive invariant. This taxonomy can be used to modularize invariants and proofs into strata of varying difficulty. Furthermore, we observe that all but the most difficult stratum can be derived almost fully automatically, even in a non-EPR setting that permits intuitive programming constructs such as arithmetic. Interestingly, proving the top-most stratum is often equivalent to proving the safety of a simpler, synchronous version of the protocol.

In this taxonomy, we identify a class of **Regular Invariants** with simple, regular structure that stem from recurring features and patterns in asynchronous message-passing systems. These invariants relate messages to their sending or receiving hosts (dubbed *Message Invariants*), assert the monotonic nature of common data structures (*Monotonicity Invariants*), and govern the ownership of unique resources (*Ownership Invariants*). As we will detail in subsequent sections, these invariants are not only easy to derive, but also easy to prove.

On the other hand, there is a separate class of **Protocol Invariants** that deal with the global relationships between hosts in the system. These capture the macro-level operation of the particular protocol, reflecting the structure of its design, and thus may require careful developer thought. Such invariants might state, for example, that a decision is made only when a majority of the nodes agree on it, or that all replicas of a log must have agreeing entries.

This taxonomy is not merely conceptual, but has significant practical implications. First, it enables a streamlined, systematic approach to finding and proving inductive invariants. The developer can easily dispatch Regular Invariants, before using them as building blocks for proving Protocol Invariants. Such an approach modularizes the derivation and proof of invariants into distinct components, with the most challenging part, Protocol Invariants, neatly contained. This is in contrast to monolithic proofs of the past, where developers have written invariants that intertwine complex protocol logic with simple local-level reasoning, thereby proliferating the difficulty of Protocol Invariants across the entire proof.

More importantly, Regular Invariants are sufficiently systematic that they can be derived from the protocol description with a few hints from the user, and then proven automatically, even in a general purpose verification tool such as Dafny [24]. Although our proposed taxonomy does not cover the space of all possible invariants, we observe that these derived Regular Invariants, in conjunction with a set of user crafted Protocol Invariants, often suffice to prove a wide variety of distributed protocols. In fact, these Protocol Invariants are typically (although not always) the same set needed to prove a *synchronous* version of the protocol; i.e., in a model without a network, where a sender sends a message and the receiver receives it in one atomic step.

Leveraging these insights, we design Kondo, a methodology and tool that lets developers harness the structure afforded by the invariant taxonomy. Using Kondo, the developer focuses their efforts on proving the correctness of a simpler, synchronous version of the protocol. Kondo then generates the asynchronous protocol from the synchronous description, and automatically devises and proves Regular Invariants for the asynchronous protocol with a few simple hints from the user. In addition, Kondo carries over Protocol Invariants from the synchronous proof, and combines them with Regular Invariants to create a *draft* proof of the asynchronous protocol. Sometimes, this draft proof is complete and constitutes the final proof; otherwise, the user helps complete the proof by adding some proof annotations to the draft.

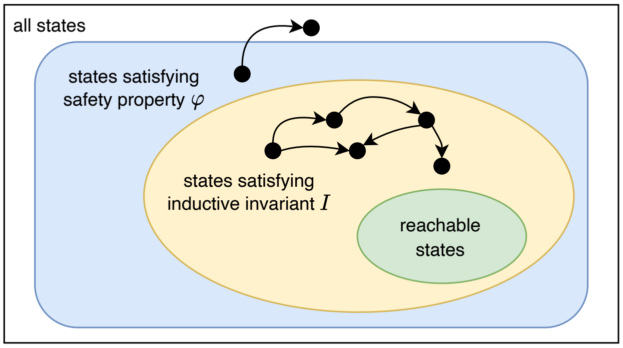

all states
states satisfying safety property φ
states satisfying inductive invariant I
reachable states

Figure 1: Venn diagram of states in a distributed system. The dots and arrows represent example states and transitions.

Overall, we make the following contributions.

1. We identify an invariant taxonomy that distinguishes between Regular Invariants and Protocol Invariants, and define three sub-classes of Regular Invariants (Section 3).
2. We propose the Kondo approach to help developers leverage the structure afforded by the invariant taxonomy, and implement the Kondo tool [1] as a new feature in the Dafny compiler. The user begins by proving a simpler, synchronous protocol. Kondo then aids in lifting that proof to a fully asynchronous setting (Sections 4 and 5).
3. We evaluate the effectiveness of Kondo on a range of distributed protocols that span different application domains, from consensus to mutual exclusion. We show that Kondo can simplify finding and proving the inductive invariants of these protocols, and identify areas in which Kondo may be less effective (Section 6).

2 The Challenge of Inductive Invariants

Manually proving the correctness of an asynchronous distributed protocol is widely regarded as a challenging task. Its difficulty is compounded by the lack of principled techniques for structuring the invariants that a developer must derive for the proof. This results in invariants that entangle complex protocol logic with otherwise standard network semantics. Such invariants are hard to reason about and can complicate the entire proof.

Background. A correctness proof of a protocol involves showing that a desired safety property φ is an *invariant* that is true in all reachable states of the system. However, φ itself is usually too weak to support an inductive argument; i.e., there exist states satisfying φ that can transition to an unsafe state (Figure 1). As a result, the developer must devise an *inductive*

```
datatype Vote = Yes | No
datatype Decision = Abort | Commit
datatype Message =
      VoteReqMsg
      | VoteMsg(v: Vote, src: nat)
      | DecideMsg(d: Decision)
datatype Option<T> = None | Some(v: T)
datatype Coordinator = Variables(
      numParticipants: nat,              // some constant N
      decision: Option<Decision>,        // initially None
      yesVotes: set<nat>,                // initially empty
      noVotes: set<nat>                  // initially empty
)
datatype Participant = Variables(
      hostId: nat,                // unique identifier ∈ [0,N)
      preference: Vote,           // non-deterministic constant
      decision: Option<Decision>,        // initially None
)
```

Figure 2: Hosts and message states of the Two-Phase Commit protocol, written in Dafny. Note that Vote, Decision and Message are sum types.

invariant $I = I_1 \wedge \cdots \wedge I_n$, composed of several individual invariants I_k. I needs to both imply φ and be *inductive*, i.e., it should hold in all initial states of the system, and be closed under system transitions—if I holds for a state s, it also holds for the next state s' after any transition from s. If an individual conjunct I_k is itself inductive (even if it does not imply φ) we refer to it as *self-inductive*.

Coming up with an inductive invariant is a creative challenge because it must be strong enough to be inductive, yet weak enough to encompass all the reachable states of the system. For distributed protocols, this challenge is exacerbated by the presence of an asynchronous network that may arbitrarily delay, drop, duplicate, or re-order messages. In addition to considering the local states of each host, the developer must also contend with the state of the network and its interaction with hosts. As a result, proofs of distributed protocols require complex inductive invariants involving many clauses that simultaneously juggle host and network states [14, 28, 31].

2.1 Case Study: Two-Phase Commit

To highlight the challenge in finding inductive invariants, we use the classic Two-Phase Commit protocol. It is parameterized by an arbitrary number of participants, and it has a single coordinator (Figure 2). Participants are initialized with some preference of Yes or No that they communicate to the coordinator, which then makes a decision, using the protocol listed in Figure 3. The safety specification we target in this example is that if any participant reaches a Commit decision, then every participant's local *preference* must be Yes:

1. Coordinator broadcasts VoteReqMsg.
2. Upon receiving VoteReqMsg, a participant p replies VoteMsg(p.preference, p.hostId).
3. Upon receiving VoteMsg(v, src), the coordinator adds src to its *yesVotes* or *noVotes* set based on v.
4. The coordinator waits to receive votes from every participant. Then, if the coordinator has $|noVotes| > 0$, it sets its local decision to Abort and broadcasts DecideMsg(Abort). Otherwise, if $|yesVotes| = numParticipants$, it sets its decision to Commit and broadcasts DecideMsg(Commit).
5. Upon receiving DecideMsg(d), a participant sets its local decision to d.

Figure 3: Two-Phase Commit protocol description.

$$\forall id : \; participants[id].\mathrm{decision} = \mathrm{Some(Commit)} \\ \implies \big(\forall id' : \; participants[id'].\mathrm{preference} = \mathrm{Yes}\big) \quad \text{(2PC-Safety)}$$

Unfortunately, outside the context of EPR, there is no established methodology one can follow in finding an inductive invariant for this specification. Instead, they must rely solely on wit and will, in a journey of intuition-guided trial and error. In particular, the developer devises a candidate list of other protocol properties that when taken in conjunction with 2PC-Safety, they suspect, will form an inductive invariant.

An example of an invariant one may come up with is:

> "*if there is a* DecideMsg(Commit) *in the network, then every* VoteMsg *from each participant must contain* Yes."

Despite its apparent correctness, proving this invariant requires non-trivial supporting invariants. The developer will find that the proof requires host-level reasoning about how the coordinator processes votes, how it decides based on its local vote tally, and how it avoids sending conflicting messages. Overall, this leads to a proof that tightly intertwines host and network reasoning.

In this work, we argue that such monolithic invariants need not be the norm. Rather, they can be structured into forms that are more tractable. The idea of using simple invariants as building blocks for proving more complex ones is not a new one [3, 4]. However, we propose a systematic way of applying this idea to distributed protocols and derive invariants in layers of increasing complexity.

3 The Invariant Taxonomy

We present an invariant taxonomy designed to help developers tease apart host-level invariants from low-level invariants (e.g.,

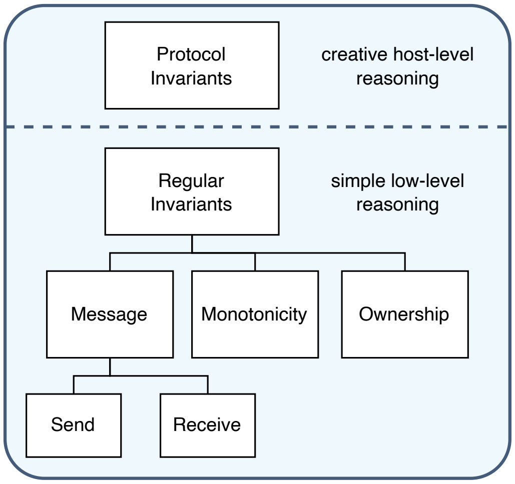


Figure 4: The invariant taxonomy.

invariants about the network). This taxonomy has two distinct categories—an upper stratum of Protocol Invariants, and a lower stratum of Regular Invariants, illustrated in Figure 4. Regular Invariants are easy to devise mechanically, and thus they can be assumed while discovering Protocol Invariants (which we show in Section 4 can even be discovered while completely ignoring the network). Using this taxonomy, the developer can contain host-level Protocol Invariants—the portion that demands user creativity—inside a well-demarcated portion of the proof. In turn, the Regular Invariants supporting those invariants are uncontaminated by host-level logic.

Note that these categories do not cover the space of all invariants. For a given protocol, there can exist invariants that are neither Regular Invariants nor Protocol Invariants. However, we find that our categorization is comprehensive enough to encompass all the invariants needed to prove a wide variety of distributed protocols (Section 6).

3.1 Regular Invariants

Regular Invariants are structurally simple and can be derived without requiring an understanding of why the protocol works. They concern low-level properties that follow from network semantics, the monotonicity of data like certain counters and sets, and the syntactic structure of protocol steps. They are also often self-inductive, which makes them easy to prove, and are even amenable to automation (Sections 4 and 5).

We identify three subcategories of Regular Invariants, namely Message Invariants, Monotonicity Invariants and Ownership Invariants, depicted by the hierarchy in Figure 4.

Message Invariants. These relate the state of the network to the state of hosts. In this way, they act as the logical bridge for proving invariants about relationships between host states when the hosts are separated by a network medium. Message Invariants come in two flavors:

1. **Send Invariants** assert that a message m is in the network only if it was sent by a host. They also describe, for each message variant, some relationship p between the message contents and the state of its sender:

$$\forall m \in \mathit{network} : p(m, \mathit{hosts}[m.\mathrm{src}])$$

2. **Receive Invariants** assert that if some condition q is met at some host h, then there must exist some message m in the network that was received by that host and has some relationship r with the host:

$$\forall h :\ q(\mathit{hosts}[h]) \implies \big(\exists m :\ m \in \mathit{network} \wedge h = m.\mathrm{dst} \wedge r(\mathit{hosts}[h], m)\big)$$

Monotonicity Invariants. Monotonic data types are a common primitive in distributed protocols [38]. Widely-used structures include grow-only epoch counters to filter stale messages, and add-only sets to collect votes from participant nodes. Monotonicity Invariants capture the monotonic properties of these data types, by asserting how the state of individual hosts may evolve as the system transitions:

$$\forall \sigma, \sigma' :\ \mathit{lteq}(\sigma, \sigma')$$

Here, σ and σ' are respectively the prior and current states of a host, and *lteq* represents some ordering relation on the values of local variables between the states.

Monotonicity Invariants require referencing the past states of hosts, which are typically not part of the distributed system model. In Section 4.2, we describe our history-preservation technique that enables us to systematically transform a protocol into one that preserves information about state histories. The protocol augmented with history information supports stating and proving Monotonicity Invariants, and using them in the proofs of higher-level Protocol Invariants.

Ownership Invariants. Many distributed protocols also require reasoning about uniquely owned resources. For example, at most one client can hold a unique lock in a distributed lock system, or at most one host can hold a unique key in a sharded key-value store.

Ownership Invariants capture the semantics of such resources. Specifically, they say that for each unique resource γ, at most one node can 'own' γ at any point in time, and if γ is in transit in the network, then no nodes can have ownership of γ. These properties are common to protocols that deal with resource ownership.

3.2 Protocol Invariants

Protocol Invariants describe a relationship ℓ among hosts, and do not mention the network. That is, they have the form

$$\forall h_1, \ldots, h_n :\ \ell(\mathit{hosts}[h_1], \ldots, \mathit{hosts}[h_n])$$

As such, Protocol Invariants are ignorant of the complications arising from network asynchrony. Instead, they focus on the higher-level reasoning of how host behaviors culminate in the overall correctness of the protocol, thus capturing properties that require insight into *why* the protocol is correct.

In addition, observe that given an asynchronous distributed protocol $\mathcal{P}$, any Protocol Invariant in $\mathcal{P}$ must also be an invariant in the *synchronous* version of the protocol $\mathcal{P}_{sync}$. This version, $\mathcal{P}_{sync}$, is one in which messages are delivered instantaneously between hosts—a sender sending a message and the receiver receiving it occurs as one atomic step, without a network delay. In practice, we observe that invariants used in proving the safety property in $\mathcal{P}_{sync}$ tend to be helpful Protocol Invariants in $\mathcal{P}$. Moreover, in all the protocols we evaluated, these Protocol Invariants, in conjunction with a set of automatically derived Regular Invariants, are all that is needed to prove $\mathcal{P}$ (Section 6.2).

3.3 Streamlining Proofs Using the Taxonomy

The invariant taxonomy mirrors the unique roles played by the network and the protocol logic. The network, while an inevitable part of reasoning about distributed systems, does not contribute to the underlying logic of the protocol—it simply serves as a medium to carry information between hosts. Instead, it is the interplay of host actions that affects the outcome of the protocol.

Respecting this natural division is key to writing a modular and efficient proof. By confining creativity-demanding invariants to exclusively describe hosts, Protocol Invariants ensure that user creativity is called upon only when needed, while the remaining mundane Regular Invariants are dispatched with minimal effort.

Two-Phase Commit Revisited. We revisit the Two-Phase Commit example from Section 2.1 to demonstrate how one can use the invariant taxonomy to bring order and simplicity to their proof. Figure 5 lists a set of invariants for Two-Phase Commit. The conjunction of these invariants and 2PC-Safety forms an inductive invariant that proves 2PC-Safety.

Of the six invariants, only *A*5 and *A*6 are Protocol Invariants. Discovering them involves protocol-level insight about how the states of different hosts are related. On the other hand, the remaining invariants are Regular Invariants. They arise from the individual steps where messages are sent or received, and describe what that particular step says about the network. For instance, *A*3 stems directly from protocol step 4. Likewise, *A*4 follows from step 5. They are also self-inductive—each invariant is preserved by the step that it directly relates to and is trivially preserved by other steps. For example, *A*1 relates directly to protocol step 2, which is the only step that adds a VoteMsg to the network.

Armed with this insight, the developer can employ the following proof strategy. They can first write down the Regular Invariants *A*1, *A*2, *A*3 and *A*4 without any thought about overall protocol correctness. This is easy because these invariants are apparent just from looking at individual, local protocol steps. Their self-inductive nature also allows the developer to quickly check if the invariants they wrote are correct. Finally, with these Regular Invariants effortlessly in place, the developer then devises the crowning jewels *A*5 and *A*6 as Protocol Invariants, the one part of the proof that requires creativity.

*A*1 For each VoteMsg(v, *src*) in the network, *src* is a valid participant identifier.

*A*2 For each VoteMsg(v, *src*) in the network, v reflects the *preference* of the participant identified by *src*.

*A*3 For each DecideMsg(d) in the network, d reflects the local *decision* at the coordinator.

*A*4 For each participant that decided Commit, DecideMsg(Commit) must be a message in the network.

*A*5 For each *id* in *yesVotes* at the coordinator, the participant identified by *id* must have the corresponding *preference*.

*A*6 If the coordinator decided Commit, then every participant's preference must be Yes.

Figure 5: Two-Phase Commit protocol invariants structured using the invariant taxonomy. The conjunction of these, together with 2PC-Safety, forms the inductive invariant of the protocol. *A*5 and *A*6 are Protocol Invariants, while the rest are Message Invariants.

4 Finding Invariants the Kondo Way

We now present the Kondo methodology and tool to help developers leverage the regularity afforded by the invariant taxonomy. In contrast to EPR-based approaches, Kondo targets general protocol models where determining the inductiveness of an invariant is an undecidable problem.

Kondo is based on two core observations. First, the systematic structure of Regular Invariants makes both *deriving and proving* these invariants amenable to automation, even in an undecidable setting that permits higher-order logic.

More surprisingly, we observe that Protocol Invariants can be devised and proven in a synchronous version of the protocol $\mathcal{P}_{sync}$ and used directly in the asynchronous protocol. Because messages are delivered instantaneously between hosts in $\mathcal{P}_{sync}$, it is much easier to iteratively devise an invariant and prove inductiveness. In all the distributed protocols in our evaluation (Section 6.2), the Protocol Invariants taken from $\mathcal{P}_{sync}$, in conjunction with the set of derived Regular Invariants for the asynchronous protocol, are sufficient to form an inductive invariant for each protocol's safety property (although this may not always be the case; see Section 6.5).

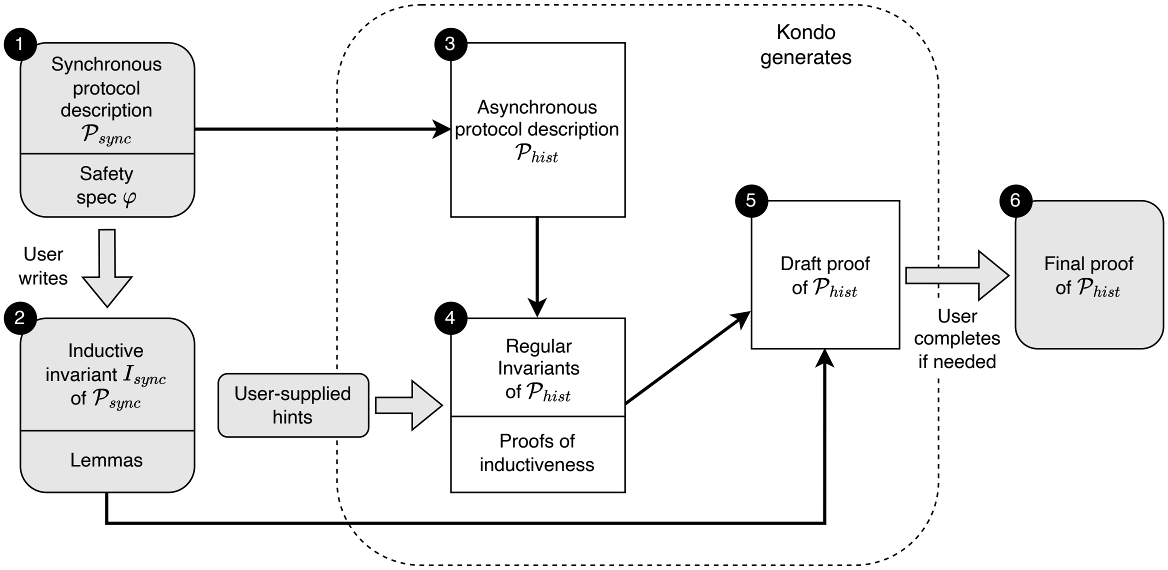

1
Synchronous protocol description $\mathcal{P}_{sync}$
Safety spec φ
User writes
2
Inductive invariant I_{sync} of $\mathcal{P}_{sync}$
Lemmas
User-supplied hints
Kondo generates
3
Asynchronous protocol description $\mathcal{P}_{hist}$
4
Regular Invariants of $\mathcal{P}_{hist}$
Proofs of inductiveness
5
Draft proof of $\mathcal{P}_{hist}$
User completes if needed
6
Final proof of $\mathcal{P}_{hist}$

Figure 6: Kondo workflow. Shaded bubbles represent artifacts that the user writes. Boxes represent artifacts that Kondo generates. Sometimes, the draft proof from ❺ may constitute the final proof, and the user need not perform step ❻ in such cases.

These observations inform the Kondo methodology. Using Kondo, the developer first writes and proves a synchronous version of the protocol. Then, Kondo automatically generates

1. a *history-preserving* asynchronous protocol $\mathcal{P}_{hist}$. In addition to modeling an asynchronous network, $\mathcal{P}_{hist}$ maintains a history of host states that aid in expressing Message and Monotonicity Invariants;
2. a set of Regular Invariants (sometimes with the help of user-supplied hints), along with their proofs of inductiveness; and
3. a *draft* proof of correctness of $\mathcal{P}_{hist}$. This draft may constitute the final proof, or may require modest effort from the developer to complete.

4.1 Overview

Figure 6 shows an overview of how a developer uses the Kondo methodology and tool to prove the safety of a protocol.

Step ❶: The user begins by writing a synchronous version of the protocol, denoted as $\mathcal{P}_{sync}$, together with a safety specification φ. This synchronous execution model does not have a network component. Instead, the overall system is simply a collection of host states that communicate atomically.

Step ❷: The user proves that $\mathcal{P}_{sync}$ satisfies φ by devising an *inductive invariant* I_{sync} that implies φ. Because we operate in a general setting in which checking the inductiveness of I_{sync} is undecidable, the user may also need to write a set of lemmas to convince the verifier that I_{sync} is indeed inductive.

Step ❸: Given $\mathcal{P}_{sync}$, the Kondo tool automatically generates a history-preserving asynchronous protocol $\mathcal{P}_{hist}$. It shares the same safety property φ as $\mathcal{P}_{sync}$.

Step ❹: Kondo generates a set of Regular Invariants for $\mathcal{P}_{hist}$, together with lemmas that prove their inductiveness. We denote the conjunction of all Regular Invariants as R. Note that to generate Receive, Monotonicity and Ownership Invariants, Kondo requires small hints from the user. The nature of these hints is detailed in Section 5.1.

Step ❺: From the user-written proof of $\mathcal{P}_{sync}$, Kondo generates a draft proof of $\mathcal{P}_{hist}$. This draft uses the conjunction $I_{\text{sync}} \wedge R$ as the inductive invariant. It lifts lemmas written for the $\mathcal{P}_{sync}$ proof to the new asynchronous context, while leaving gaps in places where the translation fails. Section 5.2 describes this process. Notably, code generated by Kondo is human-readable.

Step ❻: The user runs the verifier on the draft proof. In some cases, the draft suffices as the final proof. Otherwise, particular lemmas in the draft may be incomplete. The user then manually completes the proof of $\mathcal{P}_{hist}$ by filling in gaps in the bodies of these lemmas. Notably, *no new lemmas* need to be constructed, and the logical line of reasoning is identical to $\mathcal{P}_{sync}$. However, the user may need to write additional proof annotations to convince the verifier that the lemmas still hold in the asynchronous protocol.

4.2 Protocol Models

Like prior work [13, 14, 26, 43, 44], we use the temporal logic of actions (TLA [21]) to model a protocol as a state machine described by non-deterministic transition relations. This state machine in turn contains one or more sets of hosts. For example, the Paxos system has three sets of Proposer, Acceptor and Learner hosts respectively.

Hosts are themselves modeled as event-driven state machines that communicate via messages. A host can 1) take a spontaneous action that may or may not send a message, 2) take an action given some received message as input, or 3) crash for an indefinite amount of time.

Formally, a host is defined by a $\textsc{HostInit}(h\text{: Host})$ predicate and a $\textsc{HostNext}(h\text{: Host}, h'\text{: Host})$ relation. $\textsc{HostInit}$ circumscribes the initial states of the host, while $\textsc{HostNext}$ describes the host's state transition relation.

In the Kondo methodology, the developer works with two versions of a distributed protocol, $\mathcal{P}_{sync}$ and $\mathcal{P}_{hist}$, that differ by their network model. $\mathcal{P}_{sync}$ is initially written by the developer, whereas $\mathcal{P}_{hist}$ is derived automatically.

Synchronous Protocol $\mathcal{P}_{sync}$. The global state of $\mathcal{P}_{sync}$ is $S := (\sigma_1, \dots, \sigma_n)$, an n-tuple of host states. Its initial states are defined by the predicate

$$\textsc{SyncInit}(S) := (\textsc{HostInit}(\sigma_1), \dots, \textsc{HostInit}(\sigma_n))$$

asserting that hosts satisfy their respective initial conditions.

The system transitions are defined through the relation

$$\textsc{SyncNext}(S, S') := \textsc{Action}_1(S, S') \lor \cdots \lor \textsc{Action}_\textsc{k}(S, S')$$

where each action disjunct represents an atomic transition that the system may take. Each action also falls in one of two categories. First, one non-deterministically chosen host may take a local action, i.e., one that doesn't send or receive any data. Second, a non-deterministically chosen pair of sender and recipient hosts may take a corresponding send and receive action simultaneously; i.e., the sender transmits a message that is received instantaneously by the recipient.

Note that a consequence of tightly coupling sender and recipient pairs is that one host cannot both receive and send messages within a single action, as that would allow an arbitrary chain of hosts taking steps at once. This limitation does not sacrifice generality, as an action that receives and sends may always be modeled as two consecutive actions, with the first receiving the message and the second sending its response. It is, however, an additional restriction over the host model in prior work [14,15], and may increase proof complexity. Nevertheless, our evaluation shows that even with such a restriction, Kondo allows users to write simpler proofs than previous state of the art (see Paxos discussion in Section 6.3).

History-Preserving Asynchronous Protocol $\mathcal{P}_{hist}$. Given the synchronous protocol $\mathcal{P}_{sync}$, Kondo automatically generates a *history-preserving* asynchronous protocol $\mathcal{P}_{hist}$. This model adds to the synchronous model an asynchronous network that may arbitrarily delay, drop, duplicate, or re-order messages. Like prior work [14,15,45], we model this network as a monotonically increasing set of sent messages. When a host sends a message, the message is added to this set. When a host calls receive, it retrieves from this set some message addressed to it.

Unique to Kondo is the history-preserving aspect of $\mathcal{P}_{hist}$. It maintains a sequence *history* of host snapshots, enabling us to express monotonicity properties. Formally, let $history := [S_0, \dots, S_m]$ be a sequence of host state n-tuples. Each entry in *history* is a snapshot of all hosts in the system. Then the global state of $\mathcal{P}_{hist}$ is $S_{hist} := (history, N)$. The latest entry in *history* represents the current state of the hosts, while N is the set of messages representing the network's latest state.

The initial states of the system are defined by

$$\begin{aligned} \textsc{Init}(S_{hist}) := \; & len(S_{hist}.history) = 1 \\ & \land\ \textsc{SyncInit}(S_{hist}.history[0]) \ \land\ S_{hist}.N = \emptyset \end{aligned}$$

The system transitions are given by the relation

$$\begin{aligned} & \textsc{Next}(S_{hist}, S'_{hist}) := \\ & \quad \land\ len(S_{hist}.history) \geq 1 \\ & \quad \land\ S_{hist}.history = trunc(S'_{hist}.history) \\ & \quad \land\ \textsc{SyncNext}(curr(S_{hist}), curr(S'_{hist})) \end{aligned}$$

Here, $trunc(s)$ yields s with the last item removed. Meanwhile, $curr(S_{hist})$ gives the current state of the system, namely the tuple $(last(S_{hist}.history), S_{hist}.N)$, where $last(s)$ returns the last item in a sequence s.

4.3 Case Study: Echo Server

To illustrate the Kondo methodology, we apply it to a simple Echo Server protocol. It comprises an arbitrary number of clients and a single server. Each client maintains a constant set of *requests* that are defined by unique client and request identifiers.

Clients send their requests to the server. Upon receiving a request, the server responds by echoing the request back to the sender. When a client receives a response, it stores it in a local *responses* set. The safety specification is that clients do not receive rogue responses; i.e. for each client c, every element in c.responses matches a request in c.requests:

$$\forall\, client:\ client.\text{responses} \subseteq client.\text{requests} \qquad \text{(ES-Safety)}$$

We now show how the developer applies the Kondo methodology to proving this protocol.

Step ❶. The user starts by writing a synchronous version of the Echo Server protocol, $\mathcal{P}_{sync}$. They define the states of hosts (Figure 7), and the transitions that they can make (Figure 8).

They then define $\mathcal{P}_{sync}$ as the collection of a group of clients and a server. Here, $\textsc{SyncInit}$ asserts that every host satisfies their respective initialization predicates:

$$\begin{aligned} & \textsc{SyncInit}(S) := \\ & \quad \land \textsc{ServerInit}(S.\text{server}) \\ & \quad \land \forall\, 0 \leq id < |S.clients| : \textsc{ClientInit}(S.\text{clients}[id], id) \end{aligned}$$

```
datatype Request = Req(clientId: nat, reqId: nat)

datatype Message =
      SubmitReq(req: Request)
      | Response(req: Request)

datatype Client = Variables(
      clientId: nat,                          // unique identifier
      requests: MonotonicSet<Request>,
      responses: set<Request>
)

datatype Server =
      Variables(currentRequest: Option<Request>)

predicate ClientInit(v: Client, id: nat)
      ∧   v.clientId = id
      ∧   v.responses = ∅
      ∧   (∀ req ∈ v.requests : req.clientId = id)

predicate ServerInit(v: Server)
    currentRequest = None

```

Figure 7: Client and server states of the Echo Server protocol, written in Dafny. The MonotonicSet type is wrapper around Dafny's built-in set type and implemented in Kondo's monotonic type library. It indicates to Kondo that the sets are non-decreasing. Meanwhile, ClientInit does not constrain the size of a client's *requests* set, but ensures that every item in *requests* is marked with the client's unique *clientId*.

Meanwhile, SyncNext is a disjunction of two system-level atomic actions:

$$\text{SyncNext}(S, S') := \text{Action}_1(S, S') \vee \text{Action}_2(S, S')$$

In Action$_1$, a client-server pair performs ClientRequestStep and ServerReceiveStep respectively, where the client sends a request to the server, and the model ensures that the server instantaneously receives the request. In Action$_2$, a server-client pair performs ServerResponseStep and ClientReceiveStep, with the server sending its response and the client receiving it.

Step ❷. Next, the developer writes an inductive proof that $\mathcal{P}_{sync}$ satisfies its safety specification. Here, the inductive invariant is simple. It is the conjunction of ES-Safety with a single predicate asserting that the server's *currentRequest* belongs in the *requests* set of its sender:

$$\forall\, req : server.\text{currentRequest} = \text{Some}(req) \implies req \in clients[req.\text{clientId}].\text{requests} \quad (1)$$

Step ❸. Given $\mathcal{P}_{sync}$ written in step ❶, Kondo produces a history-preserving asynchronous version of the Echo Server protocol, $\mathcal{P}_{hist}$.

```
step ClientRequestStep(
  v: Client, v′: Client, send: Message)
   ∧ v′ = v                          // client state unchanged
   ∧ send.SubmitReq?
   ∧ send.req ∈ v.requests      // send SubmitReq(req)

step ClientReceiveStep(
  v: Client, v′: Client, recv: Message)
   // client receives Response

step ServerReceiveStep(
  v: Server, v′: Client, recv: Message)
   // server receives Request

step ServerResponseStep(
  v: Server, v′: Server, send: Message)
   ∧ v.currentRequest.Some?        // enabling condition
   ∧ v′.currentRequest = None
   ∧  send = Response(v.currentRequest)

```

Figure 8: Client and server transition relations from state v to v', with the bodies of ClientReceiveStep and ServerReceiveStep omitted for brevity. The argument *send* is a new message sent into the network, and *recv* is a message received from the network. Note that the '?' syntax is used to assert if a value is of a particular type or variant.

Step ❹. Together with $\mathcal{P}_{hist}$, Kondo also derives a set of Regular Invariants, along with their proof of correctness in $\mathcal{P}_{hist}$. In this example, the only hint required from the user is to label the client's *requests* set as a MonotonicSet type, shown in Figure 7. Note that the client's *responses* set does not need to be labeled as a MonotonicSet, because even though it is monotonic, such a property is not relevant to proving safety. The generated Regular Invariants are:

- A Message Invariant stating that every Response(req) is such that req was once processed by the server:

$$\forall\, \text{Response}(req) \in network : \exists\, i : \; history[i].\text{server.currentRequest} = \text{Some}(req) \quad (2)$$

- A Message Invariant stating that every SubmitReq(req) is such that req is in the *requests* set of the sender:

$$\forall\, \text{SubmitReq}(req) \in network : \exists\, i : \; req \in history[i].\text{clients}[req.\text{clientId}].\text{requests} \quad (3)$$

- A Monotonicity Invariant stating that the *requests* set at each client is non-decreasing:

$$\forall\, i \le j, clientId : history[i].\text{clients}[clientId].\text{requests} \subseteq history[j].\text{clients}[clientId].\text{requests} \quad (4)$$

Step ❺. The final step is to prove that $\mathcal{P}_{hist}$ satisfies the safety property. To this end, Kondo generates a draft proof of $\mathcal{P}_{hist}$ by combining the synchronous proof written in step ❶ with the generated Regular Invariants from step ❹. It derives from Equation (1) the history-preserving analogue

$$\forall\, i, req:\ history[i].\text{server.currentRequest} = \text{Some}(req) \implies req \in history[i].\text{clients}[req.\text{clientId}].\text{requests} \tag{5}$$

It then uses the conjunction of Equations (2) to (5) as the protocol's inductive invariant I.

Step ❻. In this example, the draft proof suffices as the final proof for the asynchronous Echo Server protocol, and no additional effort is required from the user.

4.4 Why History Preservation is Important

The Echo Server example also underscores the importance of our history-preservation technique. First, the history-preserving property of $\mathcal{P}_{hist}$ makes deriving and proving Regular Invariants amenable to automation. For instance, observe that for any $\textsc{Response}(req)$, its presence in the network implies that the sender of the message performed a $\textsc{ServerResponseStep}$ at some point in its execution history, during which req was its *currentRequest* value (Figure 8 line 19). This can be expressed as

$$\begin{aligned}\forall\, msg:\ & msg.\textsc{Response}?\ \wedge\ msg \in network \\ \implies & \exists\, i: \textsc{ServerResponseStep}(\\ & \quad history[i].\text{clients}[msg.\text{src}], \\ & \quad history[i+1].\text{clients}[msg.\text{src}], \\ & \quad msg)\end{aligned} \tag{6}$$

Equation (6) is easy to derive mechanically because it does not contain explicit references to internal host state, yet it implies all the properties that such a step entails, such as Equation (2).

Beyond making Message Invariants easy to derive, history preservation is what allows the invariant taxonomy to apply cleanly to a wide variety of protocols. Consider how the inductive invariant derived in step ❺ of Echo Server proves ES-Safety. First, Equation (2) allows us to relate $\textsc{Response}$ messages directly to the state of their sender (i.e., the server). Equation (5) then connects the server's state to a prior state of the respective client. Finally, Equation (4) relates that prior state to the current state to imply ES-Safety.

Without preserving history, Monotonicity Invariants such as Equation (4) would be impossible to express. Moreover, any previous values of $server.\text{currentRequest}$ are overwritten and erased from the system, hence preventing us from expressing simple Message Invariants such as Equation (2). Instead, the developer would have to resort to the alternative statement

$$\forall\, \textsc{Response}(req) \in network:\ \textsc{SubmitReq}(req) \in network$$

which mixes the protocol logic of how the server processes requests together with network reasoning, and is neither a Protocol Invariant nor a Regular Invariant. As explained in Section 2.1, this would result in proofs that are less tractable.

5 Automation in Kondo

Given a file describing a synchronous protocol $\mathcal{P}_{sync}$, Kondo generates the asynchronous protocol $\mathcal{P}_{hist}$, and human-readable files stating the derived Regular Invariants together with the proof that they are indeed invariants in $\mathcal{P}_{hist}$. Kondo also produces a draft proof of $\mathcal{P}_{hist}$ by combining the user-written synchronous proof with the derived Regular Invariants. In this section, we describe how Kondo derives and proves Regular Invariants with minimal user guidance, and how Kondo generates the draft proof of $\mathcal{P}_{hist}$.

We use the Dafny language and verifier [24] to specify and verify our protocols. We also implement Kondo as a new feature [1] inside the Dafny compiler.

5.1 Automating Regular Invariants

Message Invariants. In Kondo, we use the special sum type `Message` to define the messages of the system (e.g., Figure 7 line 2). We also require that messages be tagged with the unique identifier of their source host, accessed via $msg.\text{src}$. Without loss of generality, we assume that for each message variant α, there is exactly one host step T_α that sends that message. If there is more than one, α can simply be split into multiple variants, one for each host step that sends α.

Recall that Message Invariants relate the state of hosts to the state of the network via *Send Invariants* and *Receive Invariants* (Section 3.1). Kondo derives Send Invariants asserting that for each message in the network, its sender must have performed the action that sent that message. To do so, Kondo enumerates over `Message` variants. For each variant α, Kondo produces the statement that for each α message, m_α, in the network, there is some index i in the execution *history* when the source host of m_α performed the step T_α that sent m_α (exemplified by Equation (6)). Such statements yield two critical pieces of information—one, the *enabling condition* of step T_α (i.e., the preconditions required for step T_α to be taken) was satisfied at time i; and two, the state at time $i+1$ is in accordance with the transition. When combined with Monotonicity Invariants, they provide information on the current state of the system.

On the other hand, Receive Invariants derived by Kondo assert that if some witness condition q_α is met at a host state σ, then there must be a message of variant α in the network that made $q_\alpha(\sigma)$ true. More formally, if a host satisfies q_α at index j in its execution history, then there must be an earlier index $i < j$ and message m_α such that q_α is satisfied at index $i+1$ but not i, and this step involved the receipt of m_α. An example of a witness condition and the derived Receive Invariant for

```
// User-provided witness for PROMISE messages in Paxos
predicate PROMISEQ($v$: ProposerHost, $acc$: AcceptorId)
    $acc \in v.\mathit{promisesSet}$

// Generated Receive Invariant
predicate RECVPROMISEVALIDITY($s$: GlobalState)
    $\forall\, \ell,\ j,\ acc$ :
        PROMISEQ($s$. history[$j$]. proposers[$\ell$], acc)
        $\Longrightarrow$
        $(\exists\, i,\ msg :\ i < j$
          $\land\ \neg$ PROMISEQ($s$. history[$i$]. proposers[$\ell$], acc)
          $\land$ PROMISEQ($s$. history[$i{+}1$]. proposers[$\ell$], acc)
          $\land$ RECVPROMISESTEP(
                $s$. history[$i$]. proposers[$\ell$],
                $s$. history[$i{+}1$]. proposers[$\ell$],
                msg ) $)$

```

Figure 9: PROMISEQ is an example of a witness condition for the Paxos protocol. From this definition Kondo generates a corresponding Receive Invariant.

the Paxos protocol is listed in Figure 9. Receive Invariants inform the state of the network given the state of recipient hosts. When combined with Send Invariants, they bridge the relationship between senders and recipients.

Presently, Kondo requires the user to manually write the witness conditions. Kondo then generates one Receive Invariant for each condition. In practice, these are simple conditions that do not require much creativity. For instance, a representative condition is PROMISEQ in the Paxos protocol (Figure 9), which hints that any acceptor's ID in the *promises* set of a proposer host must have arrived via a message. The fact that specific fields in the host state are designed to track information received from messages must already be known by the developer at the time the protocol is conceived.

Monotonicity Invariants. These assert the monotonic policies of common data fields in local host state, such as round numbers and write-once variables used to store consensus decisions. In Kondo, we implement a library of common data types, each of which has a partial order relation, *less-than-or-equal-to* (*lteq*). The library includes write-once option types, grow-only numeric types, and append-only sets and sequences. Developers can easily expand this library with custom data types that implement the *lteq* interface.

Whenever the developer uses a monotonic type, Kondo produces an invariant stating that at any point in history, a value of that type must be *lteq* any future value. Equation (4) is one example, where *lteq* is the $\subseteq$ relation for sets. Importantly, the verifier enforces that these types are used correctly.

```
// User-provided label
predicate HOSTOWNSRESOURCE($v$: Host)
    $v$. hasLock                                  // boolean flag

// User-provided label
predicate INFLIGHTFORHOST($v$: Host, $msg$: Message)
    $msg$. dst $= v$. hostId $\land$ $v$. epoch $< msg$. epoch

// Generated Ownership Invariant
predicate ATMOSTONEOWNERPERRESOURCE(
  $s$: GlobalState)
   $\forall\, h_1, h_2 :\ ($
      $\land$ HOSTOWNSRESOURCE($s.\mathit{hosts}[h_1]$)
      $\land$ HOSTOWNSRESOURCE($s.\mathit{hosts}[h_2]$)
      $\Longrightarrow h_1 = h_2)$

// Generated Ownership Invariant
// RESOURCEINFLIGHTBYMSG is auto-generated wrapper
// around INFLIGHTFORHOST
predicate ATMOSTONEINFLIGHT($s$: GlobalState)
   $\forall\, m_1, m_2 :\ ($
      $\land$ RESOURCEINFLIGHTBYMSG($s, m_1$)
      $\land$ RESOURCEINFLIGHTBYMSG($s, m_2$)
      $\Longrightarrow m_1 = m_2)$

// Generated Ownership Invariant
// RESOURCEINFLIGHT is auto-generated wrapper
// around INFLIGHTFORHOST
predicate HOSTOWNSRESOURCEIMPLIESNOTINFLIGHT(
  $s$: GlobalState)
   RESOURCEINFLIGHT($s$)
        $\Longrightarrow$ NOHOSTOWNSRESOURCE($s$)

```

Figure 10: Example of user-provided ownership labels and the Ownership Invariants that Kondo generates for the Distributed Lock protocol.

Ownership Invariants. Many distributed protocols, such as lock services and sharded stores, revolve around ownership of exclusive resources. Though resources vary greatly in function and behavior, the semantics of what it means for the resource to be uniquely-owned is common. Figure 10 shows an example for how Ownership Invariants work in the Distributed Lock protocol [14], where hosts pass around a unique lock in a ring configuration.

For Kondo to generate Ownership Invariants, the user labels a data type as a uniquely-owned resource and describes its ownership semantics. This is done by defining two predicates under special names. First, HOSTOWNSRESOURCE states what it means for the host to own the lock. Second, INFLIGHTFORHOST describes the enabling condition for a host to receive a lock that is *in-flight*, meaning that the lock is in transit as a network message and can be received by the destination host. In Distributed Lock, HOSTOWNSRESOURCE says that a host owns the lock when its *hasLock* field is true, and INFLIGHTFORHOST evaluates to true when the message's *epoch* value is larger than that of the host.

We emphasize that these labeled conditions are not new concepts that a user must invent for Kondo. Instead, these are formulas that they must inevitably write for any ownership-based protocol. Kondo just requires them to be named in a standardized format.

Using the two predicates, Kondo generates invariants to cover the semantics of a uniquely owned resource. They assert that at most one host can own the resource, at most one copy of the resource can be in-flight, and that if the resource is in-flight then no one can own the resource in the meantime (respectively, ATMOSTONEOWNERPERRESOURCE, ATMOSTONEINFLIGHT, and HOSTOWNSRESOURCEIMPLIESNOTINFLIGHT).

Inductive Proofs of Regular Invariants. All of the Regular Invariants generated by Kondo come with verified Dafny lemmas proving that they are indeed invariants. The systematic structure of Regular Invariants ensures that these lemmas can be derived through simple syntax-driven rules. These invariants are such that every individual Message Invariant and Monotonicity Invariant is self-inductive, while the conjunction of Ownership Invariants is inductive.

5.2 Automating the Draft Proof

The final goal is to prove that the history-preserving asynchronous protocol $\mathcal{P}_{hist}$ satisfies its safety property. To aid the user in doing so, Kondo generates a draft proof based on the proof of the synchronous version $\mathcal{P}_{sync}$.

First, Kondo lifts the inductive invariants of $\mathcal{P}_{sync}$ to the history-preserving asynchronous world using the following mechanical transformation. Given a $\mathcal{P}_{sync}$ invariant $I(S)$, its history-preserving analogue is $I'(S_{hist}) := \forall i : I(S_{hist}.history[i])$, which simply states that I is true at all points in the history of S_{hist}. An example of this is the transformation of Equation (1) into Equation (5).

Second, Kondo also lifts lemmas for $\mathcal{P}_{sync}$ into lemmas for $\mathcal{P}_{hist}$. It converts inductive proofs of synchronous invariants to those of their history-preserving analogues by adding quantifiers over histories and choosing the appropriate elements in the histories as arguments to functions such as I.

One important detail is how Kondo handles *triggers* [29]. Triggers are syntactic patterns involving quantified variables that tell the Dafny verifier when to consider concrete instantiations of the quantifiers. For each quantified formula, the user may either manually supply its triggers or rely on Dafny to infer them automatically. In both these cases, Kondo lifts the triggers used in the synchronous invariants into their asynchronous counterparts. This ensures that the Dafny verifier triggers on the same terms in the draft proof as in the synchronous proof, such that lemmas that pass the verifier for $\mathcal{P}_{sync}$ are also likely to pass for $\mathcal{P}_{hist}$.

Overall, Kondo's generated draft proof is a best-effort syntax-guided translation. As such, it may contain lemmas that do not pass the Dafny verifier. Failing lemmas fall into one of two categories. First, the lemma's body contains $\mathcal{P}_{sync}$-specific constructs, namely the concrete instantiation of synchronous actions that cannot be easily translated into the $\mathcal{P}_{hist}$ context. In these cases, Kondo removes these lines from the $\mathcal{P}_{hist}$ proof, and requires the user to complete the translation. Second, Dafny simply needs more proof annotations to guide the verifier in exploring $\mathcal{P}_{hist}$'s more complex state space. In both these cases, the user embellishes the lemma body with more proof code, and may additionally call upon the generated Regular Invariants. With the candidate inductive invariant already in place, however, the effort demanded in this step is mechanical, and the bulk of user creativity is confined to the initial $\mathcal{P}_{sync}$ proof. We quantify this effort in Section 6.4.

6 Evaluation

We evaluate the Kondo methodology and tool by applying it to a wide range of distributed protocols. Informed by our experience in building and verifying distributed systems, this selection seeks to cover the space of common protocols as much as possible. The result is the list of protocols in Table 1, which concern a variety of application domains, ranging from consensus to mutual exclusion and concurrency control. We used Two-Phase Commit, Ring Leader Election, and Lock Server to develop and refine the Kondo approach. We then applied the Kondo approach to the remaining protocols.

Our evaluation determines whether Kondo is effective in helping developers prove the correctness of their distributed protocols. In doing so, we answer the following questions.

1. How applicable is the Kondo methodology in finding the inductive invariants of various distributed protocols? (Section 6.2)
2. How effective is Kondo in reducing the number of invariants a user must derive manually? (Section 6.3)
3. How burdensome is writing proof annotations when using Kondo? (Section 6.4)
4. Are there cases in which the Kondo methodology fails? (Section 6.5)

6.1 Evaluation Methodology

Each protocol in Table 1 is described as a state machine in Dafny following the IronFleet style [14], together with an associated safety property. For each protocol, the desired output is a theorem checked by Dafny which shows that the

	$\mathcal{P}_{sync}$ LoC	Invariant Clauses					Lines of Proof Code		
		No Kondo	Sync	Owner	Mono	Msg	No Kondo	Sync	Mods
Echo Server	260	5	1	0	1	3	93	40	0
Ring Leader Election [5]	183	6	1	0	0	2	191	56	0
Simplified Leader Election [39]	255	7	3	0	1	2	136	94	0
Two-Phase Commit	400	8	4	0	1	3	184	133	19
Paxos [23]	631	27	20	0	5	6	856	557	220
Flexible Paxos [16]	633	27	20	0	5	6	856	554	226
Distributed Lock [14]	194	2	0	3	0	0	64	31	0
ShardedKV	213	2	0	3	0	0	172	61	7
ShardedKV-Batched	225	2	0	3	0	0	172	31	0
Lock Server [26]	287	7	1	6	0	0	267	44	15

Table 1: Summary of proof effort using Kondo. Column '$\mathcal{P}_{sync}$ LoC' is the lines of code of the $\mathcal{P}_{sync}$ protocol description. Under 'Invariant Clauses', 'No Kondo' is the number of invariants a user writes to prove the asynchronous protocol when not using Kondo, while 'Sync' is the number of Protocol Invariants the user writes in completing the synchronous proof when using Kondo. The columns 'Owner', 'Mono' and 'Msg' count the Ownership, Monotonicity and Message Invariants Kondo generates. Under 'Lines of Proof Code', 'No Kondo' is the amount of code a user writes to prove the asynchronous protocol when not using Kondo. Meanwhile, 'Sync' is the amount of user-written code to prove $\mathcal{P}_{sync}$ in Kondo, and 'Mods' represent the total size of lemmas that the user had to modify in completing the draft proof generated by Kondo.

protocol's safety property is an invariant. In particular, we note that every protocol and its respective inductive invariant is outside of EPR.

To obtain the proof of each protocol, we apply the Kondo methodology by first finding and proving the inductive invariant for a synchronous version of the protocol $\mathcal{P}_{sync}$. From $\mathcal{P}_{sync}$ and its proof, the Kondo tool generates the asynchronous protocol $\mathcal{P}_{hist}$, a set of Regular Invariants, and a draft proof of $\mathcal{P}_{hist}$. Finally, we manually fill in any gaps in the draft proof to complete the final output.

6.2 Applicability of Kondo and the Invariant Taxonomy

We report that the Kondo methodology succeeds in producing inductive invariants for all 10 protocols. Table 1 tallies how the invariant clauses in each inductive invariant are classified in the invariant taxonomy. We note that the Regular Invariant columns in Table 1 only counts those that are useful in the final inductive invariant; i.e., removing such an invariant causes the proof to fail. In all of the examples we tested, the number of extraneous Regular Invariants generated by Kondo is small (at most 2). Hence, there is little threat of the developer being overwhelmed by a deluge of unneeded invariants.

Overall, this result demonstrates the applicability of Kondo along three fronts. First, it shows that the taxonomy is comprehensive in the properties that it covers. Using only invariants that fall within the taxonomic categories, we are able to express all the properties needed to form the inductive invariants of an extensive set of distributed protocols.

Second, the result shows that the inductive invariant for each of these protocols can be formulated in way that respects the classification between Protocol Invariants and Regular Invariants, where host-level reasoning is cleanly confined to Protocol Invariants. This supports the main hypothesis of the invariant taxonomy—it is the well-delineated core of Protocol Invariants that capture the deeper intuitions of the system design. Beyond this core, a set of easily derivable Regular Invariants completes the rest of the proof.

Third, the result supports the conclusion that Kondo is a viable strategy and tool for proof developers. By using only Protocol Invariants derived from $\mathcal{P}_{sync}$ in conjunction with the Regular Invariants generated using Kondo, we are able to find inductive invariants for a wide selection of protocols.

6.3 Reducing the Invariant-Finding Burden

In this study, we investigate the degree to which Kondo alleviates the developer's burden through reducing the number of invariant clauses they need to find manually for a protocol. In the case of Kondo, manually-derived invariants are the Protocol Invariants listed under 'Sync' in Table 1. We compare this to the number of invariants one must devise using the conventional IronFleet approach, listed under 'No Kondo'—these numbers are obtained from performing fully manual proofs of the asynchronous but non-history-preserving versions of the protocols, which represent the conventional way to write these protocols [14, 15].

In most cases, the number of manual invariants is drastically reduced when using Kondo, e.g., by up to 6x for Ring Leader

Election. Surprisingly, for Distributed Lock, ShardedKV and ShardedKV-Batched, the user need not write any invariants at all when using Kondo. This is because these protocols are about managing unique resource ownership, a problem that is trivial in the synchronous model where resources are passed atomically between hosts and there is no need to guard against duplication that may occur in an asynchronous network. In the asynchronous model, Ownership reasoning is in turn handled automatically by Kondo's Ownership Invariants.

For Paxos and Flexible Paxos, we observe that the reduction is less drastic, from 27 to 20 in both cases. An experimental factor contributes to the smaller difference. In the non-history-preserving version of these protocols (used to obtain the 'No Kondo' numbers), we used a simpler state machine for the proposer hosts. In particular, they include a step that received a message and sent the response in the same step, a simplification that made the proofs more tractable. Kondo, however, does not allow steps that perform both a send and a receive (see "Synchronous Protocol" in Section 4.2). Hence, we applied Kondo to a modified proposer state machine where that step was decomposed into two separate steps. Indeed, the fact that Kondo required fewer manual invariants despite this added complication highlights the usefulness of Kondo.

Furthermore, these numbers do not fully reflect the qualitative relief Kondo gives to the developer. Because Protocol Invariants are derived from the synchronous protocol model $\mathcal{P}_{sync}$, the developer can ignore the complications caused by network asynchrony when conceiving them. This luxury makes deriving Protocol Invariants qualitatively easier than the invariants in the traditional setting that is tarnished by the asynchronous network.

Overall, we find that Kondo allows the developer to be responsible for both fewer and simpler invariants across a variety of protocols.

6.4 Proof Experience

Because our protocols are not expressed in a decidable logic such as EPR, the user is inevitably tasked with writing proof annotations to convince the verifier of the correctness of the inductive invariant. In the Dafny language, these proof annotations come in the form of lemmas that resemble a hand-written inductiveness proof. Using the Kondo methodology, the user is responsible for writing proof annotations in two steps of the process (steps ❷ and ❻ in Figure 6, respectively):

1. Prove that the conjunction of Protocol Invariants is an inductive invariant of the synchronous protocol $\mathcal{P}_{sync}$.
2. Complete the draft proof of the asynchronous protocol, if it is not already complete. This involves adding proof annotations to the *bodies* of lemmas that fail to verify. Notably, the user need not introduce new lemmas, or modify the pre- and post-conditions of existing lemmas.

The columns 'Sync' and 'Mods' in Table 1 under the heading 'Lines of Proof Code' quantify the above efforts respectively, using lines of code as a proxy. They are in contrast to the 'No Kondo' column, which represents the amount of proof a user writes when not using Kondo.

Note that in the 'Mods' column, the numbers represent the total lines of lemmas that required additional proof annotations. In other words, even if just one line had to be added to the lemma, the lines of the entire lemma definition are counted. This is to include conservatively the effort the user may spend reading the lemma in order to complete the proof.

Finally, we emphasize that completing the asynchronous draft proof is a mechanical process once the developer has the synchronous proof in place. In particular, in the asynchronous proof, the developer uses exactly those lemmas already defined in the sync-proof, and the logical reasoning behind why the lemmas are true remains identical. The developer simply adds more assertions in the proof to guide the verifier in exploring a larger state space.

6.5 Limitations

Kondo targets safety proofs of crash fault tolerant distributed protocols. As such, liveness proofs are beyond its scope. Moreover, Kondo is not applicable in a Byzantine fault model, as it is not safe to relate a message to the state of a Byzantine-faulty sender or receiver.

Next, Kondo is not guaranteed to be complete, in that it may not work for every protocol or safety property. First, Protocol Invariants derived based on the synchronous protocol may not be correct invariants in the asynchronous model. For instance, the property "there is at least one node holding the lock" is an invariant in the synchronous version of Distributed Lock, but not the asynchronous one where the lock can be in-flight and not held by any node. In our evaluation, however, we have not encountered any safety properties where such invariants were required.

Second, even when the Protocol Invariants derived in the synchronous protocol are correct invariants in the asynchronous version, it is not theoretically guaranteed that their conjunction with Kondo-generated Regular Invariants must be an inductive invariant of the asynchronous protocol. However, we did not come across any such examples.

Ultimately, unlike EPR-based techniques, Kondo is intended to augment, rather than replace, general verification frameworks. In cases where Kondo fails to apply, it is always possible to fall back to the general verification framework, albeit giving up on the full benefits of using Kondo's structured invariants. It is also possible that new invariant categories and techniques may be needed as we apply Kondo to more diverse protocols. In this sense, Kondo comprises a toolkit of techniques that is general enough to cover a useful set of protocols, and it may grow to accommodate new classes of problems as they arise.

7 Related Work

Verification of distributed systems has received substantial attention, with proposed solutions falling along a spectrum of automation with differing trade-offs.

Manual Proofs. IronFleet's proof of Paxos [14] and Verdi's proof of Raft [42] both use general-purpose theorem provers to tackle their respective correctness proofs. They require entirely handwritten invariants and proofs, cumulating in 4,581 lines of Dafny for IronFleet and 50,000 lines of Coq for Verdi. However, they both accomplish the feat of verifying not just the protocol in the abstract, but an entire executable implementation, whereas Kondo is concerned with the protocol.

Automated Invariant Inference. To reduce the substantial effort needed for these proofs, considerable work has focused on automating the process. Ivy [32] makes use of the effectively propositional logic fragment (EPR [34]), which makes inductiveness-checking decidable and efficient in practice. Building on this, several algorithms aim to automate the construction of invariants wholesale; these include I4 [26], SWISS [13], DistAI [44], IC3PO [11], DuoAI [43], Primal-Dual Houdini [33], and P-FOL-IC3 [19].

However, EPR restricts how developers can express their protocols. In invariant inference, this restriction applies to the protocol description and its inductive invariant. For example, to handle Paxos, Padon et al. [31] develop abstractions that transform the verification conditions into EPR, a creative process aided by knowing the invariants in advance. Meanwhile, most approaches that can automatically infer invariants for Paxos (e.g., SWISS) require the protocol to already be transformed. Kondo, however, is not limited to EPR, so it does not share these restrictions. This is possible because it is not a *fully* automatic approach, instead allowing some human intervention. As a result, our solution to Paxos uses a natural, non-transformed protocol description.

Other works that infer invariants outside of EPR include a paper [12] targeting Paxos and the endive tool [36], which verifies a Raft-based protocol using the expressive language of TLA+. Being outside EPR, they cannot check invariants automatically in the unbounded domain, similar to Kondo. The endive authors report providing human guidance but did not quantify such effort.

Leveraging the Structure of Distributed Systems. Like Kondo, some work has used the observation that synchronous systems are easier to reason about than asynchronous ones. Pretend Synchrony [40], for example, rewrites asynchronous protocols into synchronous ones with much simpler invariants. However, it requires protocols to obey a restriction called "round non-interference" which precludes certain optimizations that save state between rounds, as in Multi-Paxos.

Some work introduces reasoning principles for the round-based *Heard-Of model* [6], including PSync [9], the $\mathbb{CL}$ logic [8], and *ConsL* [27]. Of course, these frameworks are only applicable to protocols that operate in rounds. Other work in this area [7] makes use of the communication-closure property of some protocols, but may not apply to protocols that do not have this property, such as the Echo Server.

Like Kondo, the work on *message chains* [28] identifies a class of useful invariants based on an insight into the structure of distributed systems. It proposes *message-chain invariants* that accumulate history inside network messages, explicitly mixing host and network state. In contrast, Kondo aims to isolate these two concerns.

Leveraging Ownership. Frameworks such as Aneris [20] and Grove [38] use separation logic [35] to reason about distributed systems. Separation logic is particularly good at reasoning about resource ownership, a concept also captured by Kondo's Ownership Invariants. Separation logic is very expressive, but it also requires the developer to come up with invariants (a process that is hard to automate), and it often requires significant technical expertise to use effectively.

8 Conclusion

This paper presents an invariant taxonomy that identifies structure in the inductive invariants of distributed protocols. The taxonomy classifies invariants into Regular Invariants (with regular structure that follows from the protocol description) and Protocol Invariants (which capture protocol-specific reasons why the protocol is correct). Building on this insight, the Kondo methodology gives developers a workflow and tool for coming up with an inductive invariant and proving inductiveness. They identify the Protocol Invariants on a synchronous version of the protocol; then use the Kondo tool to get an asynchronous protocol description and Regular Invariants; and finally, prove the inductiveness of the conjunction of Protocol and Regular Invariants, by completing the draft proof generated by Kondo.

Acknowledgment

We thank Andrea Lattuada and Jon Howell for the fruitful early discussions about this project. We also thank our shepherd, Alexey Gotsman, and the anonymous OSDI reviewers for their great feedback. This work was supported by a gift from VMware and by National Science Foundation grants CCF-2318953 and CCF-2318954.

References

[1] The Kondo tool. Available online at: https://github.com/GLaDOS-Michigan/Kondo.

[2] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2021, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Aaron R. Bradley. Understanding IC3. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, page 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Form. Asp. Comput.*, 20(4–5):379–405, Jul 2008.

[5] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[6] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22, 04 2009.

[7] Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 344–363, Cham, 2019. Springer International Publishing.

[8] Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2014.

[9] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[10] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[11] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, page 131–150, Berlin, Heidelberg, 2021. Springer-Verlag.

[12] Aman Goel and Karem A. Sakallah. Towards an automatic proof of Lamport's Paxos. *CoRR*, abs/2108.08796, 2021.

[13] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[14] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2015, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, Jun 2017.

[16] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1), Mar 2017.

[18] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 338–356, Cham, 2022. Springer International Publishing.

[20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars

Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In Peter Müller, editor, *Programming Languages and Systems*, pages 336–365, Cham, 2020. Springer International Publishing.

[21] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[22] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, Dec 2001.

[24] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 151–166, Carlsbad, CA, July 2022. USENIX Association.

[26] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2019, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2017.

[28] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), Oct 2023.

[29] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 20–29, New York, NY, USA, 2009. Association for Computing Machinery.

[30] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015.

[31] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[32] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL), Jan 2022.

[34] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.

[35] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.

[36] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and simple inductive invariant inference for distributed protocols in TLA+. *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 273–283, 2022.

[37] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL), Dec 2017.

[38] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2023, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 662–677, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2019.

[41] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[42] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)*, Jan 2016.

[43] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, Carlsbad, CA, July 2022. USENIX Association.

[44] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[45] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal verification of latency properties for distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI), Jun 2023.

[46] Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Automated verification of an in-production DNS authoritative engine. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2023, page 80–95, New York, NY, USA, 2023. Association for Computing Machinery.

Performance Interfaces for Hardware Accelerators

Jiacheng Ma, Rishabh Iyer, Sahand Kashani, Mahyar Emami, Thomas Bourgeat, George Candea
EPFL, Switzerland

Abstract

Designing and building a system that reaps the performance benefits of hardware accelerators is challenging, because accelerators provide little concrete visibility into their expected performance. Developers must invest many person-months into benchmarking to determine if their system would indeed benefit from using a particular accelerator. This must be done carefully, because accelerators can actually hurt performance for some classes of inputs, even if they help for others [53].

We demonstrate that it is possible for hardware accelerators to ship with *performance interfaces* that provide actionable visibility into their performance, just like semantic interfaces do for functionality. We propose an *intermediate representation* (IR) for accelerator performance that precisely captures all performance-relevant details of the accelerator while abstracting away all other information, including functionality. We develop *a toolchain* (ltc) that, based on the proposed IR, automatically produces human-readable performance interfaces that help developers make informed design decisions. ltc can also automatically produce formal *proofs of performance properties* of the accelerator, and can act as a *fast performance simulator* for concrete workloads.

We evaluate our approach on accelerators used for deep learning, serialization of RPC messages, JPEG image decoding, genome sequence alignment, and on an RMT pipeline used in programmable network switches. We demonstrate that the performance IR provides an accurate and complete representation of performance behavior, and we describe a variety of use cases for ltc and the resulting performance interfaces.

The code for ltc is open-source and freely available at [68].

1 Introduction

From datacenters to hand-held devices, modern systems increasingly rely on hardware accelerators to speed up a variety of tasks, such as machine learning [4, 48, 49, 64], video processing [28, 73], compression, encryption [17, 40], communication [29, 53], and even system infrastructure tasks [5, 32].

However, building a system that uses accelerators correctly—i.e., that fully extracts their performance benefits—remains a challenging task, because software engineers have little to no visibility into an accelerator's expected performance behavior. Every accelerator bakes design choices into silicon, such as specific throughput-vs-latency trade-offs [70] or assumptions about the workload [53], and if the software is a poor fit for these choices, acceleration will offer few benefits or even make performance worse [55, 59, 60].

This lack of visibility into expected performance hampers system developers in all three stages of system development: design, implementation, and deployment.

First, during the design stage, what functionality (if any) to offload, and which accelerators to use, is not obvious. Consider the offloading of (parts of) an RPC stack to an accelerator, where the candidates are RPC serializers/deserializers like ProtoAcc [53] and Optimus Prime [70], or one of several SmartNICs. Software engineers need to know what latency and throughput they can expect from each candidate accelerator, given their code and workload. Then they can decide which one offers the best price–performance ratio, before investing in thousands of new chips and refactoring the RPC stack. To answer these questions today, one needs to purchase every candidate accelerator, port the code, and benchmark them together—performance depends not only on the accelerator but also on the code and workload. For example, Optimus Prime is best suited for small data objects (≤300B), while ProtoAcc is best suited for larger data objects (≥4KB) [53], but this does not transpire at all from vendors' datasheets. Blindly offloading to *any* accelerator is not an option either, because this can end up degrading system performance. For instance, for workloads comprising long strings, ProtoAcc can perform worse than a regular Xeon server, because the accelerator is bottlenecked by memory-intensive operations [53].

Second, in the implementation stage, software engineers want to know how they can best optimize their code for the chosen accelerator. Ideally, tools like compilers should answer such questions quickly and automatically, but compilers too are hampered by the lack of visibility into accelerator performance. For instance, the TVM compiler [15]—a widely used compiler for deep learning models—takes several hours to optimize code for a target accelerator [16, 58]. This is because the compiler cannot figure out quickly and accurately what latency can be expected when running a specific sequence of instructions on the accelerator. So it generates multiple variants of the code and profiles them on the accelerator itself (or on slower cycle-accurate simulators [7] when the accelerator is not available) to pick the optimal one. This makes optimizing code for accelerators challenging [16, 58], given the large space of candidate code sequences, the fact that providing an accelerator for each compilation run is costly, and that engineering teams often optimize for the next generation of accelerators even before the hardware is available.

Third, when deploying a system, engineers often need *guar-*

antees on performance properties. Consider an autonomous-vehicle driving system that integrates accelerators for real-time image decoding, object detection, and object recognition. To guarantee safe navigation in all operating conditions, engineers must be able to precisely know, for example, the upper bound on image decoding latency. There exists no good way to "verify" the performance of third-party accelerators today. The state of the art is blackbox testing, which is rarely sufficient, so system designers typically rely on heuristics and accumulated wisdom [35]. Given that accelerators are expected to become ubiquitous [63, 79], this status quo must change.

We argue that hardware accelerators should come with standardized *performance interfaces* [42, 43, 45] that summarize performance behavior just like semantic interfaces summarize functionality. Software engineers routinely use semantic interfaces such as code documentation or header files to quickly find answers to questions like what a system call does, or which library is best suited for their requirements, or how incorporating a library will affect their system's functionality as a whole. Since an accelerator's raison d'être is performance (after all, its functionality could come just as well from software running on a general-purpose processor), performance interfaces are as integral to the correct use of accelerators as are semantic interfaces. As explained above, using an accelerator without a performance interface can fail to deliver on the acceleration promise, or even make performance worse.

We propose a new abstraction for representing accelerator performance that makes performance interfaces possible for hardware accelerators; we call this abstraction a *Latency Petri Net* (LPN). An LPN distills only the performance-relevant details of a circuit and excludes all other information, such as functionality. This distillation enables LPNs to serve as a high-fidelity intermediate representation (IR) of a circuit that is *performance-equivalent*: it takes the same inputs as the original circuit, and its performance behavior matches that of the original circuit. The semantics of the LPN circuit's outputs, however, are different. We envision accelerator developers manually producing the LPN as part of their regular design process, and shipping it with the accelerator. We show that doing so is both straightforward for accelerator developers (takes a few hours) and enables them to better understand and debug their own designs. We also show that the LPN of an accelerator need not disclose proprietary intellectual property.

We develop *a toolchain* (ltc) that, based on an accelerator's LPN, automatically produces performance interfaces in the form of simple, human-readable Python programs. Software engineers can use these interfaces to make informed decisions at the system design stage without needing to write code or to purchase the accelerator. ltc also provides a performance simulator that helps engineers understand how to optimize their code. Since the LPN distills only performance-relevant details, ltc's performance-only simulator is orders of magnitude faster than its state-of-the-art cycle-accurate counterparts that also simulate functionality. Finally, ltc also provides a formal verification tool that enables software engineers to prove key performance properties before deploying their systems (e.g., latency bounds for a specific but potentially infinite class of workloads). Our toolchain prototype works well for fixed-function ASICs (e.g., TPU [49] or the accelerators on SoC-based SmartNICs [4, 9, 53]) and simple programmable accelerators. General-purpose programmable accelerators (e.g., GPGPUs) are left for future work.

We demonstrate ltc's effectiveness on accelerators used for deep learning, serialization of RPC messages, JPEG image decoding, genome sequence alignment, and on a Reconfigurable Match Tables (RMT) pipeline used in programmable network switches. We show that the LPN intermediate representation can precisely capture the latency and throughput of various accelerators. Even after LPN simplifications that trade accuracy for simulation performance, we show that the IR still has an average performance-prediction error of only 1.7% across all accelerators. We present a variety of use cases for the resulting performance interfaces and LPNs, including: enabling informed decision-making during the system design stage without requiring elaborate benchmarking; cycle-level performance simulation that is up to 7821× faster than state-of-the-art cycle-accurate simulators, enabling ML compilers to generate code optimized for the accelerator in seconds instead of hours; and using formal verification to gain confidence in an accelerator's performance before deploying it in production.

The rest of this paper is organized as follows: We provide an overview of our proposed solution (§2), then define the new LPN abstraction (§3) and describe the ltc toolchain (§4). We then evaluate ltc experimentally (§5), discuss further ideas (§6), present related work (§7), and conclude (§8).

2 Design Overview

To help software engineers reason precisely about accelerator performance, we introduce the Latency Petri Net (LPN) intermediate representation: an abstraction of the accelerator's implementation that is *performance-equivalent*, i.e., its performance behavior (but not functional output) matches that of the original circuit. We then propose a workflow that uses the LPN to answer key questions about accelerator performance at the system design, implementation, and deployment stage via an extensible toolchain that we call ltc.

The LPN is inspired by classic Petri nets [69], a class of graphs used for the description and analysis of concurrent systems and processes. They are used in various domains, including the design and verification of digital asynchronous circuits. We define the LPN in §3, and Fig. 2 shows an example.

Petri nets are a good starting point for the LPN abstraction, because the key challenge in reasoning about hardware performance is not reasoning about the individual components but rather about the end-to-end performance that emerges when these components (e.g., multiple pipeline stages) operate together, in parallel. Petri nets were designed to model concur-

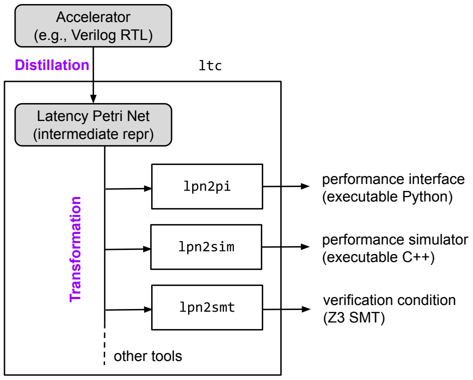


Figure 1: Proposed two-phase workflow: Hardware engineers distill their accelerator design into an LPN IR, and tools transform automatically this IR into the forms desired by accelerator users.

rent systems, and they can precisely capture hardware's inherent parallel and asynchronous execution.

Using a Petri net-like representation also ensures that the LPN is easy for accelerator developers to produce. This is because, when generating an LPN, accelerator developers do not need to reason about the impact of parallel execution on performance, rather they only need to (abstractly) represent the individual components and their local interactions. The ltc toolchain takes the final step to fill in the gaps and turn the LPN into forms that can be consumed by humans.

Fig. 1 illustrates our proposed workflow, consisting of two stages that produce and consume the LPN IR, respectively. The first stage (*distillation*) involves manually translating the accelerator's design into its corresponding LPN (we describe this in §3.3). We propose that distillation be performed by accelerator developers as part of their regular design process, but one could also imagine tools that translate Register-Transfer Level (RTL) designs into LPNs. Since the definitive clock frequency of the circuit is decided in the post-RTL synthesis stage, the LPN abstraction represents execution latency in terms of cycles (i.e., an RTL-level metric), not wall-clock time. The latter is easily calculated once the frequency is known.

The second stage of the workflow (*transformation*) automatically processes an accelerator's LPN into actionable information about accelerator performance. The ltc toolchain consists of several tools: lpn2pi summarizes the performance of the accelerator into human-readable, executable Python programs that enable software engineers to make informed development decisions without purchasing the accelerator or porting their code to it. lpn2sim produces an executable simulator of the LPN that developers and tools can use for fast performance simulation while optimizing their code. lpn2smt translates the LPN together with a user-provided performance property into a verification condition and passes it to the Z3 constraint solver [22] for a proof or refutation of the property.

This two-staged workflow—distilling the accelerator design into a performance IR and then transforming the IR into answers to specific questions about performance—provides flexibility and customizability. Since the LPN is a universal and accurate representation of the accelerator's performance, it can be transformed into answers to arbitrary questions about the accelerator's performance. We envision the set of tools in ltc expanding over time, to address other questions one might ask about an accelerator's performance.

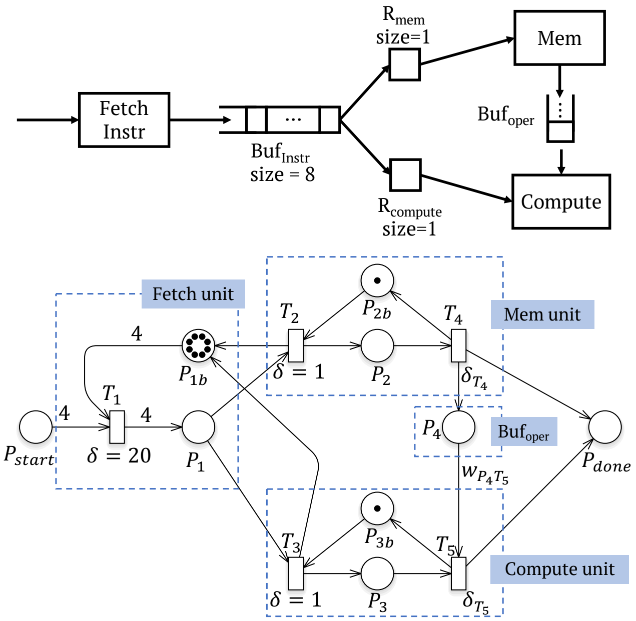


Figure 2: Example hardware pipeline (top) and its LPN (bottom). This is a simplified version of the deep-learning accelerator in §5.

3 The Latency Petri Net Abstraction

We now define the LPN abstraction, first at a high level (§3.1) and then more precisely (§3.2); a complete formal definition is beyond the scope of this paper. We then describe step-by-step how to distill an accelerator design into its corresponding LPN (§3.3). Finally, we discuss the use of LPNs to model components surrounding accelerators, such as memory and interconnects (§3.4).

3.1 LPN Overview

To illustrate the LPN concept, we use the simple hardware pipeline shown in Fig. 2. It consists of a fetch unit that brings instructions into an instruction buffer, followed by an in-order dispatch to a memory and a compute unit. The memory unit fetches the operands for compute instructions from memory into an operands buffer. The memory and compute units operate in parallel, i.e., the memory unit can fetch operands for a future compute instruction while the compute unit is still processing the current instruction. Each unit operates on one instruction at a time, stored in each unit's local register (R_{mem} respectively $R_{compute}$) until the unit finishes processing it.

The memory and compute units have variable latencies d_{mem} and d_{comp}, respectively, that take into account the instruction type and when the operand was last accessed. For simplicity of presentation, we fix the fetch unit's latency to 20 cycles (fetches 4 instructions at a time), set the instruction buffer's maximum size to 8, and let the operands buffer have infinite capacity, unlike in a real accelerator.

Reasoning about the latency of a sequence of instructions is challenging, even for such a simple pipeline, due to the fetch, memory, and compute units operating in parallel. Parallel execution can both hide latencies (e.g., loads that bring in the operands for future compute instructions may complete before the current compute instruction) and introduce stalls (e.g., in the fetch unit due to back pressure when the memory and compute units drain the instruction buffer too slowly).

The bottom half of Fig. 2 illustrates the LPN for this simple pipeline. The LPN is a directed graph with two kinds of vertices: *places* (circles) and *transitions* (rectangles). Adjacent vertices in the LPN must be of different kinds, i.e., edges in the graph can only connect places to transitions and vice-versa. Each place in the LPN contains *tokens* (solid black dots) that collectively represent the state of the circuit, and tokens are stored and consumed in FIFO order.

An LPN models how data flows through a circuit by *enabling* transitions. Each transition has a *guard* (not shown) that determines whether the transition is enabled or not, a *delay* (δ) that specifies the duration of the transition in cycles, and a *producer* function (not shown) that generates new tokens. Once a transition is enabled, after the number of cycles indicated by the delay, it *commits*, i.e., atomically consumes input tokens and produces output tokens. We define each of these operations precisely in the next section.

In Fig. 2, we show the correspondence between the circuit blocks and the subgraphs of the LPN. For some of the LPN details, such as the transition delays, one needs to consult the RTL of the accelerator (not shown). The LPN at the bottom is an abstract representation of a circuit that is performance-equivalent to the one at the top: (1) it operates on the same inputs, using a function (not shown) that converts instructions to tokens in the special place P_{start}; and (2) given any input, the number of cycles it takes the LPN to deposit the last token in P_{done} corresponds to the number of cycles the upper circuit takes to produce its output. However, the LPN's output tokens are meaningless other than indicating completion.

3.2 LPN Definition

In essence, an LPN models a system of queues connected by logic units that consume tokens originating from multiple input queues and generate tokens for designated output queues. The LPN is a directed dataflow graph in which places P_i represent the queues, and transitions T_j represent the logic units. Edges directed from places to a transition are the transition's input edges, while those directed from the transition to places are its output edges. We equip the LPN with a timestamping machinery CLK to denote when each token in the system was produced—this is a key ingredient for modeling performance.

An LPN state $S = ((s_1,\dots,s_n),t)$ is a tuple consisting of a collection $s_1,\dots,s_n$ of sequences s_i representing the in-flight tokens corresponding to places $P_1,\dots,P_n$, and one global non-negative number t, the current value of CLK. A token $k = (p,ts)$ is composed of a map $k.p$ of key-value pairs and a timestamp. Each key in $k.p$ is the name of a property of k. Each token has a type, determined by the set of properties (but not values) that tokens of that type have. All tokens in a particular place have the same type. The timestamp $k.ts$ denotes the CLK value when token k was produced. By construction, the timestamp $k.ts$ of any token in a reachable state S is $k.ts \leq S.ts$. The tokens in a place are always consumed in FIFO order, which is why $s_1,\dots,s_n$ are sequences and not mere sets.

A transition $T = (\gamma,\delta,\pi)$ is a tuple of three functions: a guard γ, a delay δ, and a producer function π. The guard decides when T is ready to execute: $T.\gamma$ reads (without consuming) a subset of the tokens present in T's input places and returns *NotReady* if the transition cannot execute at this time. If it can, then the guard returns $Enabled(w_1,...,w_k)$ with weights w_i. To execute, the transition locks the first w_1 free tokens from its 1st input place, the first w_2 free tokens from its 2nd input place, and so on. The guard must guarantee that $\forall i, w_i$ is less than or equal to the number of free (not locked) tokens already present in the transition's ith input place.

When a guard $T.\gamma$ switches from *NotReady* to *Enabled*, thus enabling T, the transition does not immediately consume the tokens but rather locks them for $T.\delta$ cycles. The lock means that no other transition is allowed to consume those tokens. At the end of the delay $T.\delta$, the transition commits: the locked tokens are atomically removed from T's input places, and the tokens produced by $T.\pi$ are pushed to the output places, with the current CLK (commit time) as their timestamp. Both the delay $T.\delta$ and the producer $T.\pi$ are arbitrary functions of all the input tokens that the transition promises to consume.

To avoid race conditions when two transitions share an input or output place, we require that the two transitions never be simultaneously enabled. The value of a guard $T.\gamma$ is not allowed to change between the moment it switches to *Enabled* and the moment when $T.\delta$ has elapsed (and T commits).

LPN Semantics. Given an initial LPN state $S_0 = ((s_1,\dots,s_n),0)$ with all the tokens in $s_1,\dots,s_n$ having a timestamp equal to 0, we define the semantics of the LPN starting from S_0 as the potentially infinite sequence of states inductively defined by $((s_1,\dots,s_n),t) \rightarrow ((s'_1,\dots,s'_n),t')$.

The next state of an LPN is obtained by applying the effects of all the transitions that are enabled at CLK $= t$ and known to be ready to commit at CLK $= t'$. For a transition T_i to belong to this group, it must be that the guard $T_i.\gamma$ returned *Enabled* at time $\leq t$, its input tokens locked in the corresponding places were produced before T_i started (i.e., the highest timestamp of those tokens is $ts_{\max} = t' - T_i.\delta$), and the earliest time when T_i can commit is t'. All transitions known to be ready to commit at t' commit as a group, and they advance the LPN from $((s_1,\dots,s_n),t)$ to $((s'_1,\dots,s'_n),t')$. When no more transitions can ever commit, the LPN has reached its terminal state.

In an LPN, it is possible that $T_i.\delta = 0$. If a commit at t' enables such a 0-delay transition, T_i will also commit at t', even if this was not previously apparent, i.e., T_i was not previously "known" to be ready to commit at t'. Thus, after the first commit, the LPN transitions $((s_1,\ldots,s_n),t) \rightarrow ((s'_1,\ldots,s'_n),t')$, and subsequently it transitions $((s'_1,\ldots,s'_n),t') \rightarrow ((s''_1,\ldots,s''_n),t')$, i.e., there are multiple states with the same timestamp. The process repeats until no more transitions can commit at t'.

LPNs are reminiscent of several extensions of Petri nets [46, 69, 86], mixing the notion of timestamp and information-carrying tokens with enforced FIFO ordering between tokens. With an LPN, we can accommodate the different modeling needs of hardware accelerators, while keeping the underlying models formal and machine-analyzable. We designed the LPN to provide a favorable trade-off between compactness, analyzability, expressivity, and ease of manipulation for our different uses and tools. We chose the name "latency Petri net" to acknowledge the inspiration we drew from Petri nets, without implying a theoretical equivalence.

3.3 Distillation: From RTL to LPN

We now describe how a hardware engineer can represent the performance of an accelerator using an LPN.

Distilling an accelerator's register-transfer level (RTL) representation into its corresponding LPN is an element-wise, structural conversion of the RTL: FIFO buffers in the RTL become LPN places, and RTL compute elements become LPN transitions. Transitions can operate in parallel (if enabled at the same time), so the engineer can produce the performance-equivalent representation by analyzing in isolation the latency of each stage of the accelerator pipeline. The LPN then glues back together this stage-by-stage performance decomposition.

RTL-to-LPN distillation is a five-step process; we describe each step in reference to the example in Fig. 2.

Step 1 involves listing the places and transitions that map directly to elements in the RTL: places P_1, P_2, P_3, P_4 correspond to the four buffers/registers, transitions T_1 to T_5 correspond to the three units that consume/produce from/to those buffers plus the two copy actions of instructions to the registers $\mathrm{R}_{\mathrm{mem}}$ and $\mathrm{R}_{\mathrm{compute}}$. The latter two are not explicit computations in the block diagram but are units in the RTL source code.

Step 2 involves defining the guard functions and the corresponding weights. For many transitions, becoming *Enabled* simply requires the presence of a specific number of tokens in an input place; their guards do not look at the properties of those tokens (e.g., $T_4.\gamma$ and $T_5.\gamma$). Occasionally, guards may depend on the values of token properties: $T_2.\gamma$ (respectively $T_3.\gamma$) will be *Enabled* if and only if the first free token in P_1 has a value corresponding to a memory (respectively compute) instruction, because instructions are dispatched in order.

Most weights returned by the guards are constants (e.g., $w_{T_1P_1} = 4$ because the fetch unit fetches 4 instructions at a time, and $w_{P_1T_2} = w_{P_1T_3} = 1$ because both registers store 1 instruction at a time). Default weights of 1 are not shown in Fig. 2. Occasionally, weights may depend on the values of token properties: $w_{P_4T_5}$ determines the number of operands transition T_5 reads, and it is a function of the value of the property of the token in P_3 that specifies the type of instruction. In both cases, the weights are intuitive for accelerator developers to define, because they directly correspond to an architectural quantity: the rate of consumption of tokens in the dataflow. This also illustrates why weights need to be computable based on tokens from all of a transition's input places.

Step 3 involves defining the delay and producer functions for each transition. The delay typically comes straight from the RTL. The producer function produces tokens with just those property values that are strictly necessary for the LPN to accurately model performance—performance-irrelevant should be discarded.

Step 4 involves modeling backpressure by adding capacity constraints to each place in the LPN. Take for example the *Mem* unit: we add an extra "capacity place" (P_{2b}) with a fixed initial number of "capacity tokens", corresponding to the capacity C of the buffer in question (1 token for P_2); this is a classic Petri net pattern [46]. The capacity place is connected to the transitions incident on the original place (T_2 and T_4), but in reverse, to form a loop. We adjust T_4's producer function to also produce 1 capacity token into the capacity place P_{2b}, and T_2's guard to require that there be at least 1 capacity token in P_{2b} to enable T_2. This way, when T_2 first commits and consumes the initial token in P_{2b}, it cannot commit again until T_4 has committed and deposited a capacity token in P_{2b}. This models the *Mem* unit backpressure: no new instruction will be copied into $\mathrm{R}_{\mathrm{mem}}$ until the previous memory instruction has finished processing. The same pattern is applied, for instance, to the P_1 place representing the instruction buffer ($\mathrm{Buf}_{\mathrm{instr}}$), except that there are two consumers for $\mathrm{Buf}_{\mathrm{instr}}$ and the capacity is $C = 8$, thus 8 initial tokens in P_{1b}.

Finally, step 5 involves adding *start* and *done* places (P_{start} and P_{done}), and placing the initial tokens. The hardware engineer then provides a "tokens from input" function Ψ to translate the accelerator's input to the LPN tokens placed in P_{start}. A stream of input data (e.g., an image) can be split into task units (e.g., individual blocks), and each task becomes a token that is placed inside P_{start}. Depending on the accelerator's semantics, a task token could also be an instruction, a short DNA sequence, etc. When the processing of a task completes, a "done token" k_{done} should be produced into P_{done}.

Constructing LPNs is a natural fit for accelerator development workflows and a materialization of what hardware engineers already have in mind, i.e., a more detailed architectural diagram annotated with latency expressions. Compared, for instance, to building a simulator, producing an LPN is easier, because the accelerator functionality is abstracted away. ltc provides a Python library with built-in types for places, transitions, edges, etc. that engineers can use to write the LPN.

We asked a hardware engineer to produce an LPN for the Menshen RMT pipeline used in programmable network switches [82]. After taking 3 days to understand the RTL design, he produced the corresponding LPN (which we evaluate in §5) in less than 3 hours. This suggests that the manual distillation step is indeed straightforward for someone who understands the accelerator's design. The same engineer also mentioned that writing the LPN actually helped to better understand the performance behavior of the circuit.

Finally, in most cases, LPNs do not leak much proprietary information about the accelerators. Except for the latency details, an LPN reveals no more information than the high-level architectural diagrams, which are often made public anyway. No implementation details appear in the LPN.

3.4 Memory, Caches, and Interconnects

Accelerators are often part of a larger system, and their performance is influenced by the components surrounding them, such as memory, caches, and interconnects. LPNs can be used to model these components as well. However, they provide fewer benefits over other kinds of performance models than they do for accelerators.

First, LPNs can be constructed even without a reference RTL implementation, by speculatively modeling the internals of a hardware component based, for instance, on documentation and online posts. We built an LPN for a sophisticated PCIe interconnect based on documentation alone, and we describe this example in §5.

Second, modeling complex memory hierarchies is challenging, because semantics are tightly intertwined with performance: the latency of a cache access depends on which entries are present in the cache or not, and knowing this requires tracking the specific contents of the cache, which in turn requires modeling the semantics of the cache in more detail than for most accelerators. This is an example where the ability of an LPN to abstract away functionality is limited, and thus its advantage over, say, a cycle-accurate simulator is reduced. One could model the state of the entire cache with a single token, and each cache line would be an individual property of that token. This LPN, though, would likely be more complex than what the ltc toolchain was designed for.

Nevertheless, an LPN can still abstract away some semantic details of the memory hierarchy and be productively used, for instance, to model and reason about the parallelism within the memory subsystem. If we took the RTL of a cache and distilled it into an LPN by following the steps discussed in §3.3, we could model the cache's internal logic (without taking into account cache state) and simulate it with lpn2sim. This could help reveal that a particular cache design can only handle 1 cache hit every 2 cycles, whereas a better design could handle a cache hit every cycle, through pipelining. The pipeline design does influence cache performance, even if not as much as replacement strategy and associativity configuration do. An LPN can help fine-tune the pipeline design.

4 Transforming the LPN

The LPN is an abstraction that is *performance-equivalent* to the accelerator; nevertheless, it is not easy to read for those unfamiliar with the accelerator's implementation details. As a result, it is not directly useful to software developers who want to use the accelerator in their systems. We now describe how the ltc toolchain bridges this hardware–software gap by transforming the LPN into representations that software developers can use in the different stages of system development.

The ltc toolchain currently consists of three main tools: (1) lpn2pi, which transforms the LPN into human-readable performance interfaces in the form of executable Python programs, which are meant to be read as much as executed; (2) lpn2sim, which merges the LPN with a simulator skeleton to produce an executable simulator that both developers and tools can use for fast performance simulation; and (3) lpn2smt, which translates the LPN together with a user-provided performance property into verification conditions that can be proven or refuted using an SMT solver, to provide performance guarantees before the system is deployed. ltc also provides other, simpler tools that we do not describe here, such as lpnviz, which produces a visualization of the LPN that hardware developers can use to better understand and debug the accelerator. We envision both hardware and software engineers contributing more such tools to ltc over time.

While the lpn2sim simulator (just like the RTL) operates on *concrete* inputs, both lpn2pi and lpn2smt produce outputs—performance interfaces and verification conditions, respectively—that describe performance for an *abstract, symbolic* input. Since the space of all possible inputs to an accelerator is large, often infinite, producing complete performance interfaces or verification conditions is intractable for most LPNs, due to the path explosion problem [11]. We circumvent this challenge by introducing the notion of *input classes* for LPNs, which partition a given input space into input sets for which, individually, it is feasible to produce complete performance interfaces and verification conditions. We now describe how input classes partition an input space (§4.1), and then describe lpn2pi (§4.2), lpn2sim (§4.3), and lpn2smt (§4.4).

4.1 Input Classes for lpn2pi and lpn2smt

To use the lpn2pi and lpn2smt tools, one must first constrain the input space to the one of interest. For example, for the JPEG Decoder, the user might include all images up to a given maximum size (number of pixels × pixel depth in bits) and exclude all others. This input space is then partitioned by an ltc tool into *input classes*, with lpn2pi and lpn2smt then solving the problem for each class independently. This ltc tool employs symbolic execution [13] to partition the input space.

Intuitively, an *input class* is a group of inputs for which simulating the LPN will cause (1) each transition in the LPN to commit exactly the same number of times for all executions corresponding to inputs in that class; and (2) the n^{th} commit of each transition will consume and produce the same number of tokens in all executions, for all values of n. (An "execution" is a complete simulation of the LPN from CLK=0 until it deposits the last k_{done} token into P_{done}.) For example, if executing the LPN with an input from a class causes transition T to commit twice during the execution, consuming 3 tokens for the 1st commit and 4 tokens for the 2nd commit, then executing the LPN with any other input from that class must also cause T to commit twice and to consume 3 tokens for the 1st commit and 4 tokens for the 2nd commit. The tokens consumed and produced in different executions must have the same type (§3.2), but can have different property values. Commits of different transitions can be interleaved arbitrarily in different executions for inputs in a class; the only thing that matters is the commit and token counts. §A.1 in the Appendix contains a formal definition of input classes.

Input classes are defined such that all inputs in any given class impose the same pattern on the trace resulting from the simulation of the LPN. The tools leverage this commonality to do their analysis once per pattern (which could subsume many inputs). Even though input classes are not defined based on human-understandable semantics of the accelerator's input, they often do correspond to input types that are intuitive for users. For example, for the Protoacc LPN (§5), all messages of a given format constitute one input class. For the JPEG decoder LPN (§5), all images of the same size form a separate input class.

As mentioned, ltc includes a preprocessing tool for automatically partitioning the user-specified input space into input classes. It symbolically executes the LPN in a special way and partitions the input space into sets. One input class can possibly span multiple sets, but a set never contains inputs from more than one input class. Then, by operating on each set in isolation, lpn2pi and lpn2smt can avoid path explosion and are trivially parallelizable by input set. Please see A.2 in the Appendix for details of how input classes are generated.

4.2 lpn2pi

The lpn2pi tool transforms the LPN into human-readable performance interfaces represented as executable Python programs, in the spirit of [42, 43]. The performance interface takes the same inputs as the accelerator (e.g., a stream of network packets, multiple RPC messages, a long DNA sequence) and returns the start-to-end latency (i.e., total execution cycles) it would take the accelerator to process that input. The performance interface describes the start-to-end latency not with concrete numbers but with *formulae*, as introduced in [44] but expressed in terms of properties $k.p$ of initial tokens k in the accelerator's LPN. The performance interface returns a single formula per input class. We show examples of lpn2pi-extracted performance interfaces in Figs. 4 - 7.

lpn2pi does not aim for fully precise performance interfaces—while the LPN has suitable constructs to precisely represent the accelerator's asynchrony and parallelism, reflecting these in a precise, closed-form, human-readable formula is typically intractable. Instead, lpn2pi *approximates* the accelerator's start-to-end latency, trading precision for human readability and closed-form expressions. Approximation turns out to be sufficient, because we expect performance interfaces to be used mostly during the system design stage, when software engineers are interested in coarser-grained descriptions of performance. In §5, we show that, while approximate, lpn2pi-extracted interfaces nevertheless enable informed decisions at the design stage, such as choosing the accelerator configuration that fits best a particular workload profile.

The following three assumptions underlie the approximation made by lpn2pi: (1) The size of the input is large enough so that the accelerator's pipeline is almost always full, i.e., the time spent filling and draining the pipeline is a negligible fraction of the overall start-to-end latency; (2) For all inputs in any given input class, the accelerator has the same bottleneck, i.e., the stage in the accelerator's pipeline (or transition in the LPN) that incurs the longest delay is the same for all inputs in that class; and (3) The bottleneck in the accelerator pipeline is stable, i.e., it does not shift from one pipeline stage to another during the processing of an input.

We define the effective delay ε_T of a transition T in an execution of an LPN as the product $N \times \overline{g}_T$ of the number of times N the transition commits in that execution and the average duration between consecutive commits of that transition (called average commit gap $\overline{g}_T$). The transition with the largest effective delay is deemed to be the bottleneck. Based on the three assumptions above, lpn2pi approximates the start-to-end latency to be the effective delay ε_T of the bottleneck transition (i.e., of the bottleneck stage of the pipeline). Recall that the definition of an input class (§4.1) requires the number of times each transition commits in an execution to be the same for all inputs in that class. So, given an input class and a transition T, N is a constant for all inputs in that class. However, $\overline{g}_T$ is a symbolic expression parameterized by properties of the initial tokens, so ε_T is also a symbolic expression.

Determining ε_T for each transition comes down to determining the respective $\overline{g}_T$. Note that $\overline{g}_T$ does not necessarily equal $T.\delta$, because a transition may be stalled for an arbitrary number of cycles (due to its input places not having enough tokens) or it could be non-blocking and become enabled again before it commits (i.e., commit multiple times in parallel).

Accurately estimating the average commit gap $\overline{g}_T$ is challenging in LPNs with loops. In a loop-free LPN, $\overline{g}_T$ for all transitions is just the maximum of all transition delays. But with loops, this simple method is no longer accurate. Consider a simple loop $P_0 \rightarrow T_1 \rightarrow P_1 \rightarrow T_2 \rightarrow P_0$, with initially a single token in P_0 and none in P_1. Every time T_1 commits, it subsequently has to wait for T_2 to commit before it can be enabled again (and T_2 also has to wait for T_1), so the average commit gap for both T_1 and T_2 is the sum of the delays of T_1 and T_2.

To approximate the $\overline{g}_{T_i}$ of all transitions T_i in a loop, we define the loop delay Δ and a parallel factor F_{T_i} for each transition T_i in the loop—we estimate $\overline{g}_{T_i}$ as Δ/F_{T_i}. Here is why: Consider a simple loop that has only one place with M initial tokens. Δ is the time it takes for the M initial tokens to complete a full iteration around the loop, with all transitions committing at least once. Recall that transitions are non-blocking, in the sense that a transition T_i, once enabled, could become enabled again before it commits, if the configuration of free tokens in its input places changes such that $T_i.\gamma$ is satisfied once again. As a result, there can be multiple instances of the same transition enabled at the same time. By F_{T_i} we denote the maximum number of instances of T_i that can be simultaneously enabled. Increasing levels of concurrency proportionally reduce the gap between successive commits, so we estimate the average commit gap for T_i as $\overline{g}_{T_i} = \Delta/F_{T_i}$.

lpn2pi starts out by setting $\overline{g}_T$ $=T.\delta$ for each transition T in the LPN; if $T.\delta$ is a general function on input tokens, $\overline{g}_T$ is the corresponding symbolic expression. Then, for each loop in the LPN, lpn2pi computes Δ and the F_{T_i} factors, and then it recomputes $\overline{g}_{T_i}$ = $\max(\overline{g}_{T_i}, \Delta/F_{T_i})$ for each T_i in that loop. Once lpn2pi has treated each loop once, we say that it has completed one iteration of the process. Dependencies among loops and the order in which the loops are treated can influence the estimated $\overline{g}_{T_i}$ values, so lpn2pi continues iterating in order to improve the estimates. The current version of lpn2pi stops after a fixed number of iterations that is configurable (default 10). In the next version, lpn2pi will automatically stop when the $\overline{g}_{T_i}$ values stabilize, i.e., do not change from one iteration to the next by more than a configurable threshold. Comparing changes at the level of symbolic expressions (which is what some $\overline{g}_T$ values are) is fundamentally hard, so lpn2pi will instead compare concrete values of these expressions, obtained by randomly sampling the input class and computing the corresponding concrete values of the expressions. Once iterative estimation is complete, lpn2pi derives the start-to-end latency formula as the maximum of the ε_T expressions (i.e., $max_{T_i}(N \times \overline{g}_{T_i})$. lpn2pi then uses `sympy` [61] to simplify the formula to obtain an expression that is easier for humans to read.

lpn2pi repeats the process described in the previous paragraph for each input class, after which it emits the corresponding interface program in Python, in the form seen in Figs. 4 - 7.

Please refer to A.3 in the Appendix for more details on the computation of ε_T, Δ, F_{T_i} and on the underlying assumptions.

4.3 lpn2sim

For a given LPN, the lpn2sim tool produces a bespoke cycle-level performance simulator that can be used by both engineers and tools. As we show in §5.3, the LPN's power of abstraction enables lpn2sim-generated simulators to simulate performance orders-of-magnitude faster than state-of-the-art cycle-accurate simulators.

lpn2sim's simulator is event-driven and works as follows: In step ⓪, it sets the value of CLK to zero, and then repeatedly performs the following two steps to make forward progress. In step ①, the simulator finds all transitions that can commit at the current CLK value. If more than one can commit, the one with the smallest ID is committed first. The simulator repeats this step until no transition can commit at the current CLK value. In step ②, the simulator finds the next earliest timestamp at which a transition can commit. If no transition can commit, the simulation terminates. Else, the simulator updates the CLK to that timestamp and goes back to step ①.

lpn2sim automatically translates the LPN (described by hardware engineers using our Python API) to an equivalent C++ program. It first emits equivalent place and transition objects in C++, and then translates individual delay, guard, and output functions. To make such automatic translation of delay, guard, and output functions feasible, our Python API only allows arithmetic operations and conditionals within these functions; this proved sufficient for all the accelerators we evaluated. lpn2sim then combines the translated LPN code with a simulator skeleton we wrote in C++ and compiles to an executable.

4.4 lpn2smt

The lpn2smt tool is used to formally reason about performance properties of an accelerator based on its LPN. It has three inputs: the target LPN, the input space Υ, and a query Φ. The Υ parameter is the subspace of inputs that are of interest to the user. lpn2smt currently supports queries related to latency bounds of two kinds: what is the upper (lower) bound on latency, or can you prove/disprove expression $\phi(x)$ involving latency x. An example of the latter, which we use in our evaluation, is $\phi(x) : |(E-x)/x| < 0.2$, where E is an expression taken from an (approximate) performance interface. This asks for a formal proof that E is within 20% of the true latency for all possible inputs and, if not, asks for a counter-example. lpn2smt can be extended to support other kinds of queries too.

lpn2smt first partitions Υ into input classes (§4.1), then derives based on the LPN a precise SMT expression Λ_i for the start-to-end latency for each input class i. Then it pieces together a global latency expression for the entire input subspace Υ as $\Lambda_\Upsilon = \mathsf{ite}(C_1, \Lambda_1, \mathsf{ite}(C_2, \Lambda_2, \mathsf{ite}(...)))$ using the if-then-else ite operator supported by SMT solvers like Z3 [22] and the inputs constraints C_i that define the corresponding input classes. Note that this is a precise expression, not like lpn2pi's approximations meant to be human-readable.

For the first kind of query, lpn2smt passes Λ_Υ to the SMT solver's optimizer and asks for a formally verified upper (lower) bound on Λ_Υ. For the second kind of query, lpn2smt passes $\phi(\Lambda_\Upsilon)$ to the SMT solver and returns either a formal confirmation that it is true or a counter-example.

The SMT expression Λ_i for input class i is constructed as follows: Let N_i be the number of transition commits in an execution from this class; by the definition in §4.1, there is a unique N_i for each input class i. lpn2smt instantiates N_i symbolic timestamps CLK_1, CLK_2, ... corresponding to when the transitions committed—the start-to-end latency will be $max_{j=1..N_i}\{\text{CLK}_j\}$. To compute this expression, lpn2smt instantiates for each commit j the consumed tokens, and places them in the corresponding input places. For each such token k, lpn2smt sets $k.p$ and $k.ts$ to symbolic values constrained to reflect the relationship to CLK_j. Recall that the number of tokens produced/consumed by a transition is the same for all inputs in a class (§4.1). Then lpn2smt uses the "tokens from input" function Ψ (applied to a suitably constrained symbolic input from this class) to obtain the initial tokens, and places them in P_{start}. It then uses the transitions' producer functions to instantiate the transition-produced tokens into the corresponding places. For each initial and produced token k, lpn2smt constrains $k.p$ and $k.ts$ according to the function that produced it and the respective $T.\delta$ and CLK_j. Then, lpn2smt captures the constraints resulting from the fact that every consumed token must either be an initial token or one resulting from a commit. The constraints that result are propagated to CLK_1, CLK_2, ..., and finally lpn2smt computes $\Lambda_i = max_{j=1..N_i}\{\text{CLK}_j\}$.

In summary, the LPN is a generic IR that is performance-equivalent to the accelerator circuit and can be transformed by tools into higher level representations useful to software engineers. In this section, we presented three of the tools in the ltc toolchain: lpn2pi, lpn2sim, and lpn2smt. We envision both hardware and software engineers contributing more such tools to ltc, increasing its usefulness over time.

5 Evaluation

In this section, we evaluate ltc on several accelerators and show that it answers the questions mentioned in §1. We first describe our experimental setup (§5.1), then present fine-grained results that shed light on detailed aspects of LPNs (§5.2), and conclude with higher-level results (§5.3).

5.1 Experimental setup

We evaluate ltc on 5 accelerators (Table 1), each representative of a particular class of accelerators. We require access to the RTL, so the evaluation is limited to open-source accelerators.

Apache VTA (Versatile Tensor Architecture) [2] is a deep-learning accelerator with a compiler stack based on TVM [15]. The accelerator incorporates tensor cores that perform vector or matrix operations. The design includes parallel units for compute, load and store operations, which decouples memory accesses from the compute, to hide memory latencies [74]. VTA can be used to program arbitrary dataflows when executing the deep-learning model. Certain high-level machine learning operations can be implemented with different VTA instruction sequences. Each instruction sequence exhibits different performance, and so TVM (VTA's compiler) generates multiple instruction sequences and selects the best performing one. This process is called auto-tuning. Our evaluation uses a workload consisting of 1,500 instruction sequences generated from auto-tuning ten 2d convolution tasks from ResNet-18 [34], an 18-layer deep convolutional neural network commonly used to measure auto-tune latency and inference speed.

Accelerator	Domain	Workload	LOC
VTA [2]	Deep learning	Autotune ResNet-18 [34]	6,628 Chisel
Protoacc [53]	RPC message serialization	Hyperprotobench [31] and microbenchmarks	3,197 Chisel
JPEG [80]	Image decoding	30K Flickr [51] and 30K Div2k [50]	7,003 Verilog
Darwin [20]	Bioinformatics	10 DNA test sequences [21]	1,535 Verilog
Menshen [82]	Programmable P4 switch	3 Verilog testbenches (with up to 100 packets)	11,169 Verilog + 4,318 VHDL

Table 1: Open-source accelerators used for evaluating ltc.

Protoacc [53] is a hardware accelerator developed by Google for protocol buffers [71] and integrated into a RISC-V SoC. We only consider Protoacc's serializer, which is the most interesting part of Protoacc: multiple fields within a message are serialized in parallel within the accelerator. Deserialization is sequential and thus less interesting. As in the evaluation of the ProtoAcc paper [53], we use the Hyperprotobench benchmark [31] and their microbenchmarks to measure serialization performance of both large messages (>1MB) and small messages (<1KB). While Protoacc's standard testbench includes a complex memory subsystem (with caches, DRAM, and TLB), we are only interested here in the performance of the accelerator itself, i.e., what a vendor would provide an LPN for. Therefore, in the empirical measurements, we warm up and overprovision the caches and TLB to prevent them from disturbing the performance of the accelerator.

JPEG [80] is an image decoder core for FPGAs written in Verilog. It supports various chroma, fixed and dynamic Huffman tables, DQT tables for JPEG input streams, etc. Our workload consists of the Flickr [51] and Div2k [50] datasets. Each has 30K diverse images, and all images in the Div2k dataset are high-resolution.

Darwin [20] is a GACT (DNA sequence) alignment accelerator. The accelerator has two main stages. The first stage uses a systolic array to fill scores in a 2D score matrix, and the second stage computes alignment actions at each step: insertion, deletion, and match. For the workload, we use ten pairs of test DNA sequences used by the Darwin authors [21].

Menshen [82] is a Reconfigurable Match Tables (RMT) pipeline used in a programmable P4 network switch [12]: incoming packets are processed by flowing through a programmable packet filter, 2 packet header parsers, 5 header processing stages, and 4 header de-parsers. Menshen extends the RMT architecture with isolation mechanisms to ensure that multiple P4 programs running on the same switch do not suffer from performance interference. It spatially partitions its stateful resources (match-action table entries and stateful memories) and uses per-packet configuration overlays for its stateless resources (packet filter, header parsers, header processing stages, and header de-parsers). As workloads, we use Menshen's two original device-level testbenches, plus an additional testbench based on the original but extended to 100 packets. Menshen contains several closed-sourced IP blocks, which restricts some of our experiments.

We ran all experiments on a 2-socket 48-core Intel Xeon Gold 6248R processor with 376 GiB of memory, 1 thread per core, running Ubuntu 20.04.4 LTS with the 5.15 Linux kernel. For the speedup and accuracy baselines, we compare to Verilator [81], the fastest open-source cycle-accurate RTL simulator available today—it generates optimized C++ code from Verilog that is 200–1000× faster than interpreted simulators [81]. We use Verilator v5.010 for all accelerators except for VTA, where we use v4.022, for compatibility reasons. All speedup comparisons are single-threaded. Verilator v4.022 and v5.010 have negligible performance differences on a single thread. We use the Clang-11.1.0 compiler. For the PCIe experiments, we use an AMD Alveo U200 accelerator card connected with a gen3 x16 PCIe interconnect to a host without DDIO.

We build the LPNs for the above accelerators by manually inspecting the RTL source code. The LPNs use tokens to abstractly represent the data of various formats and units that flow through the real hardware. For example, input packets in Menshen are turned into tokens with a property representing the type and length of a packet, each 8×8 image block in JPEG is turned into a token with a property representing the number of non-zero pixels after quantization, each instruction in VTA is turned into a token with properties representing different parts of the decoded instruction, and each field in a message in Protoacc is turned into a token with properties representing the type of the field and field length. LPN transitions represent the different hardware components that operate in parallel, and LPN places represent the buffers.

5.2 Understanding LPNs in detail

We now provide a quantitative deep-dive into the LPN abstraction, and we also describe how hardware engineers can themselves use LPNs to better understand and debug their designs.

5.2.1 Accuracy and completeness of the LPN

As explained in §2, the LPN representation enables accelerator developers to describe performance in terms that are familiar to them, and then rely on the ltc toolchain to translate the LPN to representations palatable to software engineers.

Fig. 3 shows that using the LPN as an IR is justified: across all benchmarks and all accelerators, the average latency prediction error of the simulator generated by lpn2sim based on the LPN is 1.7%. The maximum error never exceeds 10%. For the LPN to be 100% accurate, it would need to retain almost all the RTL-level details, which is unnecessary in practice.

This means that the LPN provides a performance IR that is highly accurate and complete, i.e., it contains all the necessary details to provide predictions that are close to reality. Tools based on the LPN IR can therefore achieve high accuracy.

5.2.2 Representation efficiency

Besides accuracy and completeness, the utility of an LPN also depends on its conciseness, ease of update, understandability by non-technical staff, and so on. As we will show in Fig. 8, by incorporating only performance-related details and nothing else, the LPN brings about orders-of-magnitude improvements in simulation time. This is one measure of representation efficiency. In Table 2 we show the complexity of the LPNs along different dimensions, which serves as another measure of representation efficiency.

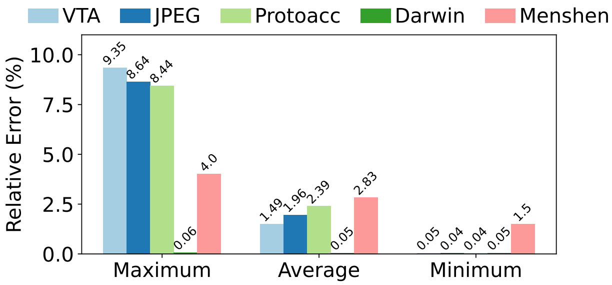


Figure 3: Relative latency prediction errors of the LPN-based simulation vs. Verilator cycle-accurate simulation.

Accelerator	LOC		Number of ...		
	RTL	LPN	transitions	places	edges
VTA	6628 Chisel	506	12	22	41
JPEG	7003 Verilog	109	6	16	33
Protoacc	3197 Chisel	758	97	112	365
Darwin	1535 Verilog	214	2	4	6
Menshen	11169 Verilog	544	29	44	85

Table 2: Comparative complexity of LPN and RTL representations.

5.2.3 Hardware engineer effort to write LPNs

As already mentioned, we asked an accelerator developer with several years of mixed academic and industry experience to read §3 and write an LPN for Menshen, whose design he had not seen before. He wrote the LPN without assistance, and then tested its accuracy with the Menshen testbenches. After understanding the RTL design, it took him less than 3 hours to write an accurate LPN. He estimated that a developer who knew the design and did not need to go back and forth between the RTL and the LPN would take less time.

The Menshen code base is quite substantial. This result therefore strongly suggests that hardware engineers would find it acceptable and practical to write LPNs for their accelerators, especially if they stand to gain (as we argue below).

5.2.4 Utility to SoC and accelerator developers

Besides being easy to write, we believe LPNs, accompanied by the ltc toolchain, can improve the productivity of accelerator designers. For example, finding the right configuration (e.g., sizing the buffers in a programmable switch) is today labor-intensive and error-prone. The wrong choices for buffer sizes can affect the delicate internal balance of an accelerator and lead to performance degradation due to unnecessary stalls.

We discovered the utility of lpn2smt in optimizing buffer sizes while trying to prove an upper bound on the stall-to-cycles ratio (≤ 0.4) for JPEG. In the default configuration, due to an under-sized buffer in the output unit, the previous unit was backpressured early. We had lpn2smt find a buffer size that respects the desired stall-to-cycle ratio: we made the buffer size symbolic and queried lpn2smt to optimize the stall-to-cycles ratio. lpn2smt took 3 minutes to return 0.276 as the optimal ratio and a concrete buffer size that satisfies that ratio. Changing the buffer size (1 line of RTL) led to a 37% performance improvement on the Div2k dataset [50]. Since only some of the images have a stall-to-cycle ratio > 0.4, such a nuanced performance bottleneck would be hard to find.

5.2.5 LPNs beyond accelerators

The performance of the interconnect and external memory accesses affects overall performance when running accelerators, so engineers may want to connect LPNs for accelerators to models for the interconnect and memory, to understand the overall system performance. LPNs are a natural fit for modeling interconnects. We inferred hardware details from PCIe documentations [66], then created an LPN for a reconfigurable PCIe topology, including root complex and switches, and connected it to the LPN for JPEG. With a fixed memory-access latency model, the LPN-based system model achieves on average 1.9% (maximum 5.1%) relative error compared to the end-to-end latency measured with the real hardware system (i.e., a JPEG decoder on an FPGA connected to the host CPU via PCIe). The image set we evaluated on includes 40 images of varying sizes, and the per-image latency ranges from 15 microseconds to 100 milliseconds.

5.3 Key results

In this section, we present high-level results that illustrate the value of LPNs and ltc to accelerator developers and users.

5.3.1 Performance interfaces are human-friendly

This set of results illustrate how LPNs and ltc can answer questions like "What latency/throughput can I expect from this accelerator for my code?" and "Which of accelerators X or Y will best accelerate my workload?". For the latter, we were unable to find two open-source accelerators that provide identical functionality, and so we demonstrate this use case using two configurations of the same accelerator.

Consider the JPEG performance interface in Fig. 4, produced by lpn2pi (as explained in §4.2, the variable names come from the token property names). A quick read conveys that the latency of decoding an image grows with the number of blocks in the image; the compression ratio, which is inversely related to the number of non-zero elements in the block, affects the latency as well. Developers can visually infer the bounds on accelerator latency. To derive a latency in seconds, one multiplies the cycles by the clock period. The `@perf_interface` decorator adapts the input, based on the "tokens from input" function Ψ (§3.3), to make the token properties (e.g., `num_blocks` and `avg_num_nonzero_perblock`) available to the interface at the right level of abstraction.

```python
freq = 75*10**6 # 75MHz
clk_period = 1/freq
@perf_interface
def latency_jpeg_decode(img):
    x = 6*(img.avg_num_nonzero_perblock*3+6)
    cycles = img.num_blocks*max(x,509)/4
    return cycles*clk_period

@perf_interface
def tput_jpeg_decode(img):
    # Images are processed one-by-one
    # We provide throughput for RGB blocks instead
    return img.num_blocks / latency_jpeg_decode(img)
```

Figure 4: Latency and throughput interfaces for the JPEG decoder. Comments are manually added. The throughput interface is manually constructed based on the latency interface.

If developers understand the parameters of their workloads, they can directly look at the performance interface to reason about the latency distribution for those workloads. Otherwise, they can generate test cases and quickly run them with the performance interface, which is executable Python code.

Next, consider the performance interface for Protoacc (Fig. 5), which directly conveys the cost of serializing different message types. The latency for serializing a series of messages is just the sum of the latency of serializing individual messages. As mentioned earlier, lpn2pi extracts the performance interface for each input class—in this case, input classes correspond to Protoacc message types—and assembles them together. The performance interface raises an error if the input message is not part of the input classes for which the performance interface was extracted. Due to space limitations, we do not show throughput interfaces, as they are straightforward to derive from the latency interface.

We use two configurations of Protoacc to demonstrate how performance interfaces can help developers choose between accelerators, or between different configurations of the same accelerator, by comparing their performance interfaces (Fig. 5). The first configuration is the original Protoacc, and the second is a smaller configuration of Protoacc with the number of parallel serialization pipelines reduced from six to one. From the interface, if the message type is *hpbench.m1*, we can infer that, if the total bytes are below 47KB, the original Protoacc is faster. And once the total bytes exceed 47KB, the alternative configuration is faster. This is because, when the message size is below 47KB, the bottleneck is still in processing the message—since the original Protoacc has more pipelines to process the message in parallel, it is faster. Once the message size exceeds 47KB, the bottleneck shifts to the generation of the memory reads/writes, and the alternative configuration is faster, because it has a higher frequency.

Finally, Fig. 6 shows the extracted performance interface for Darwin, and Fig. 7 shows the performance interface for

```python
freq = 1.8*10**9 # 1.8GHz
clk_period = 1/freq
@perf_interface
def latency_protoacc_serialize(msgs):
    cycles = 0
    # Iterate over each message of a list of messages
    for msg in msgs:
        # hpbench.m* are Hyperprotobench msg formats
        if msg.type == hpbench.m1:
            cycles += max(1468, msg.total_bytes/16+310)
        elif msg.type == hpbench.m2:
            cycles += max(2172, msg.total_bytes/16+514)
        elif msg.type == hpbench.m3:
            ...
        else:
            raise NotImplementedError(
                f"message␣type␣not␣supported"
            )
    return cycles*clk_period
```

```python
freq = 2*10**9 # 2GHz
clk_period = 1/freq
@perf_interface
def latency_protoacc_alternative_config_serialize(msgs):
    # Latency interface for another protoacc
     configuration where the number of parallel
     pipelines is reduced from 6 to 1.
    cycles = 0
    for msg in msgs:
        if msg.type == hpbench.m1:
            cycles += max(3609, msg.total_bytes/16+310)
        elif msg.type == hpbench.m2:
            cycles += max(4566, msg.total_bytes/16+514)
        elif msg.type == hpbench.m3:
            ...
        else:
            raise NotImplementedError(
                f"message␣type␣not␣supported"
            )
    return cycles*clk_period
```

Figure 5: Interfaces for default ProtoAcc (top) and an alternative configuration of ProtoAcc (bottom). We speculate that the frequency of the alternative ProtoAcc configuration could be 2GHz (instead of 1.8GHz at the top) because the design is simpler.

```python
freq = 250*10**6 #250MHz
clk_period = 1/freq
num_pe = 4
@perf_interface
def latency_darwin_gact(dna_pairs):
    cycles = (dna_pairs.ref_dna_length + num_pe + 2)*
     dna_pairs.query_dna_length/num_pe
         + num_pe + 2 + 3*dna_pairs.steps
    return cycles*clk_period
```

Figure 6: Latency interface for Darwin GACT for DNA alignment.

```python
freq = 1*10**9 # 1GHz
clk_period = 1/freq
@perf_interface
def latency_menshen(pkts):
    if pkts.type == 0:
        # length of the packets stream is 100
        cycles = max(1320, pkts.sum_nr_words + 176)
    else:
        ...
    return cycles*clk_period
```

Figure 7: Latency interface for Menshen.

Menshen. The interface for Menshen is extracted per packet-stream with a fixed number of packets but of different sizes. We do not show interfaces for VTA because (unlike the other accelerators) it is a programmable domain-specific processor, so it takes "programs" as input. VTA instruction sequences contain thousands of instructions produced by compiling a high-level program with TVM [15]. These performance interfaces are therefore program-dependent and long. We expect developers to use other ltc tools instead of reading these.

5.3.2 Performance interfaces are accurate

We report in Table 3 the accuracy of the ltc-generated performance interfaces for latency. As a baseline, we use the Verilator cycle-accurate simulator to run the workloads on the accelerators' RTL. We compare the prediction provided by the performance interfaces to the values reported by Verilator. The performance interface for Menshen is only evaluated using the 100-packet testbench; the other testbenches contain too few packets to fill the pipeline, so lpn2pi's assumptions don't hold. Of course, this does not affect the LPN's accuracy (§5.2).

Accelerator	Prediction error	
	Average	Max
JPEG	7.04%	23.39%
Protoacc	2.40%	3.83%
Darwin	0.05%	0.06%
Menshen	9.43%	9.43%
VTA	19.49%	58.93%

Table 3: Prediction accuracy of extracted performance interfaces.

The average relative error is low (<20%) for all five accelerators, despite performance interfaces being approximate. They aim to capture the major factors that affect latency, not predict precisely the latency, and (as discussed in §4.2) lpn2pi introduces some inaccuracies.

The extracted performance interfaces for JPEG and VTA have the largest maximum errors. As already explained, lpn2pi does not capture the influence of bottleneck shifts on latency (§4.2). In the JPEG decoder, the input is a stream of image blocks. If one segment of blocks is highly compressed and another is less compressed, the bottleneck for processing segments of blocks will shift back and forth within the accelerator. Similarly, in VTA, each of the parallel components (fetch, load, compute, or store) can be the bottleneck during different periods while processing the instructions.

In future work, we plan to extract a performance interface for each phase of the input stream and add the latencies spent in each phase to derive the final start-to-end latency.

5.3.3 LPN-based performance simulation is up to 3 orders of magnitude faster than existing simulators

Another set of questions is "How do I generate code optimized for accelerator X", "How can I do that quickly, in compile-and-run cycles typical of software development workflows", and "How can I evaluate my envisioned workload on an accelerator that isn't available just yet?" These questions might be relevant directly to developers, or to tools, such as the TVM compiler for deep-learning models mentioned in §1. A common approach to answer such questions, when the real hardware is not available, is to use cycle-accurate simulators.

lpn2sim provides substantial benefits, up to three orders of magnitude. Fig. 8 shows lpn2sim's speedup over Verilator. All cycle-accurate simulators simulate both performance and functionality, which is wasteful when only performance questions are being asked. Speedups are more significant with larger accelerators, because there is more functionality that the underlying LPN abstracts away.

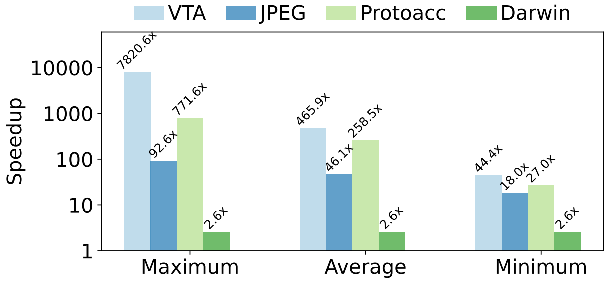


Figure 8: Simulation speedup: LPN-based simulation vs Verilator.

Table 4 shows the absolute simulation times. We believe that orders-of-magnitude changes in performance simulation time, such as going from ∼2 hours to ∼20 seconds, can bring about qualitative changes in how the tools are used.

	Simulation time	
Accelerator	**Cycle-accurate Verilator**	**LPN-based lpn2sim**
VTA	119 min	19 sec
JPEG	2159 min	38 min
Protoacc	25 sec	0.08 sec
Darwin	0.13 sec	0.05 sec

Table 4: Simulation time: LPN-based simulation vs. Verilator.

To get a feel for the impact of faster simulation time on developer productivity, we benchmark the auto-tuning process in the TVM compiler, which optimizes deep-learning models for accelerator targets (§5.1). Auto-tuning can be done either upon initial compilation, or be manually triggered whenever there are changes to the model, to the hardware, or to its configuration. We compare end-to-end compilation time when TVM uses Verilator vs. lpn2sim. Fig. 9 shows the outcome for the 10 auto-tune tasks in our workload (§5.1). As part of this auto-tuning, TVM generates 1,500 sequences of instructions, ranging from 62 to 159,947 instructions.

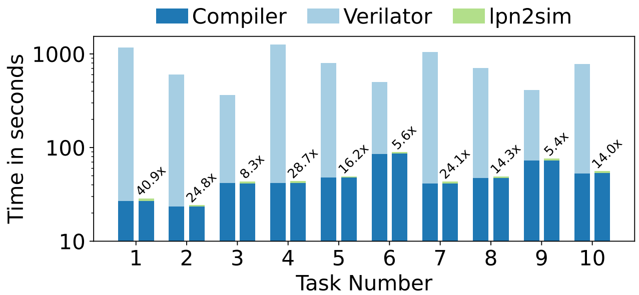


Figure 9: End-to-end compilation time, including auto-tuning. On top of the compiler+lpn2sim bars we overlay the compiler+lpn2sim vs. compiler+Verilator speedup. Despite the log-scale y-axis, one needs to zoom in to see the small amount of time taken by lpn2sim.

lpn2sim reduces auto-tuning time to a negligible amount, turning a highly *non*-interactive process into an interactive compile-and-run cycle. This enables software engineers to think differently about optimization, and to do it more often and without the accelerator hardware. Even if engineers had access to the actual hardware accelerator, using lpn2sim to auto-tune allows many more developers to do so in parallel. Compared to cycle-accurate simulation, it saves not only time but substantial amounts of compute resources and energy.

5.3.4 LPN-based tools enable performance verification

Consider the JPEG decoder in the autonomous driving scenario described in §1. To ensure safe operation in all circumstances, engineers need hard guarantees on the accelerator's performance, particularly for unseen and untested workloads. lpn2smt makes it possible to prove non-trivial bounds that are difficult to infer from source code or semantic interfaces.

The first example is determining, for some image compression ratio x%, the worst-case and best-case latency, and the corresponding worst-case and best-case inputs. Take some specific examples that may be relevant to the engineers: For a typical 90% compression ratio, lpn2smt proves that the worst-case decoding latency is 2,290 cycles for 12 RGB (18 YCrCb) 8×8 macro blocks. If the input images consist of 4 least-compressed and 14 maximally-compressed macro blocks, the best-case decoding latency is 1,717 cycles, and the difference between worst-case and best-case latency is 33%. At a 75% compression ratio, the worst-case and best-case decoding latencies are 3,063 and 2,540 cycles, respectively. lpn2smt took less than 2 minutes to find and prove these bounds.

Similarly, a Protoacc user may wonder about such bounds for serializing a message with a fixed number of bytes. We used lpn2smt to prove that, for message types with 16 fields (total 10KiB), the latency is between 726 and 1,074 cycles. SoC designers could leverage this kind of proofs when incorporating third-party accelerator blocks into their design and reason about performance implications.

lpn2smt can also be used to prove bounds on the accuracy of the performance interfaces produced by lpn2pi. Using lpn2smt, we verified formally that the latency predicted by JPEG's performance interface will always be within at most 43% of the LPN's prediction for 12 RGB (18 YCrCb) 8×8 macro blocks. This result is significant, because the input space is 64^{18} possible images, and thus infeasible to explore directly. This bound is not tight, but guaranteed to be correct.

Of course, the strength of the guarantees depends on the accuracy of the LPN. Validation tools (see §6 below) could provide confidence levels to accompany vendor-provided LPNs.

6 Discussion

In this section, we present further thoughts on how LPNs can help accelerator vendors, whether LPNs leak intellectual property, and how LPNs can be validated against the RTL.

Using LPNs in the accelerator design stage. An LPN can be written even before the accelerator's RTL is finalized. This LPN can be released to software engineers in the same organization, who can then start optimizing software for the accelerator using lpn2sim, as well as identify mismatches in performance expectations early (using lpn2pi and lpn2smt), before the design is finalized. Since accelerator vendors often release SDKs along with their accelerators, the LPN can help speed up development by providing visibility into the expected performance behavior of the accelerator before it is built.

How much proprietary information does an LPN reveal? To ensure that LPNs can be shared beyond the same organization and with software developers at large, they must not leak proprietary information. We argue that this is the case, since (1) most of the information revealed through the structure of the LPN is typically already revealed in architectural block diagrams that are made public by vendors, and (2) while LPNs provide additional information about the latency of the different compute stages, they do not describe how the accelerator achieves this latency, nor give circuit-level details and micro-architectural implementation details that are central to achieving competitive frequency and power consumption. That said, concerned vendors could still provide lower time-resolution LPNs, i.e., LPNs with coarser-grained delay functions; this reduces accuracy to safeguard proprietary details.

Validating LPNs. Since LPNs are distilled manually, they can contain mistakes; hence, after being constructed, LPNs should be validated. Developers could validate the LPN against the RTL using their RTL testbenches. Validating an LPN against the RTL is similar to how engineers validate the RTL itself using functional simulators, code reviews, and testbenches. Nevertheless, we plan to pursue building automated tools that can *formally* validate LPNs against the RTL.

7 Related Work

Petri nets have long been used to model and evaluate the performance of systems [24]. Furthermore, languages modeling a system of queues and actors are not a new idea. Kahn networks [52], dataflow networks [3, 23], and synchronous languages [8] share similarities with LPNs: more or less explicitly describing the flow of tokens in the system.

Analytical modeling of accelerators: Amid the rise of domain-specific accelerators and the need for efficient code generation, research explored semi-analytical modeling for performance models of Domain Specific Accelerators (DSA). For example, to search for good tiling and mapping of loop nests on dense tensor accelerators, [65] proposed performance models that can quickly evaluate the performance of running various loopnests on a family of accelerators. [62] tackles a similar problem for sparse tensor accelerators, and [33] for a SmartNIC. Those approaches use domain-specific knowledge in their modeling, so they typically don't offer abstractions or methodologies that can be reused in other domains. LPNs are domain-agnostic and provide a general substrate for building performance models of accelerators. There are also analytical models for accelerators that focus on data movement costs or asynchronous operations with the CPU [1, 19, 75], rather than the performance of the accelerator itself. Those models have a coarser modeling granularity than LPNs.

Performance models in the hardware community: The monograph [26] covers performance modeling techniques in detail. Analytical models based, for example, on Amdahl's law have studied various computing scenarios to establish performance trends [27, 36]. Similarly, the roofline model [84] allows simple modeling to compute performance upper bounds. Other analytical models [56, 57] build good predictors of processor performance from a few numbers: number of cache misses, branch mispredictions, etc. Finally, interval simulation [14, 30, 37] measures the distribution of performance-structuring events (cache misses and mispredictions) and profile the performance of the machine around those events to produce performance models. The way we construct performance interfaces from LPNs leverages similar principles.

Machine learning has been used to produce so-called predictive performance models of systems [25, 41, 47, 72]. These models are incomplete representations of performance, as they can only answer the questions they were trained on.

The use of simulators [10, 39, 76] to model performance is a battle-tested strategy. To make simulation faster, sampled simulation has been proposed [83, 85]. Challenges include computing warm states (caches, predictors, etc.) and identifying representative parts of benchmarks [6, 38, 67]. Finally, FPGAs [18, 54, 77, 78] can be used to speed up simulation, but FPGA simulation is possible only when the RTL is available, and compilation for FPGA is slow.

In contrast to these approaches, LPNs not only produce accurate *executable* models of hardware but can also be transformed into other useful representations (such as performance interfaces) to address broader performance questions.

LPNs for accelerators are complementary to host simulators like gem5 [10]. One can replace the gem5+RTL simulation mode with gem5+LPN for accelerators. gem5+RTL is normally bottlenecked by the RTL simulation, and gem5+LPN would shift the bottleneck to gem5.

8 Conclusion

Performance interfaces promise to offer a standardized view of accelerator performance. Despite the complexity of accelerators and system software, the LPN IR we propose can accurately represent the dynamics of various accelerators, and ltc can answer non-trivial and valuable performance questions.

9 Acknowledgments

We are grateful to Katerina Argyraki, Ed Bugnion, Jim Larus, and Diyu Zhou for their help in shaping the ideas presented here. We thank our shepherd, Abhishek Bhattacharjee, and the anonymous reviewers for their help in improving our paper.

References

[1] Altaf, M. S. B., and Wood, D. A. LogCA: A high-level performance model for hardware accelerators. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 375–388.

[2] Apache Versatile Tensor Architecture. https://tvm.apache.org/vta.

[3] Arvind, Gostelow, K. P., and Plouffe, W. Indeterminancy, monitors, and dataflow. In *Symp. on Operating Systems Principles* (1977).

[4] AWS Inferentia Accelerators for Deep Learning Inference. https://aws.amazon.com/machine-learning/inferentia/.

[5] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[6] Baddouh, C. A., Khairy, M., Green, R. N., Payer, M., and Rogers, T. G. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2021).

[7] Beamer, S. A case for accelerating software rtl simulation. In *IEEE Micro Journal* (2020).

[8] Berry, G., and Gonthier, G. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.* (1992).

[9] NVIDIA Bluefield-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[10] Binkert, N. L., Beckmann, B. M., Black, G., Reinhardt, S. K., Saidi, A. G., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Altaf, M. S. B., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[11] Boonstoppel, P., Cadar, C., and Engler, D. R. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[12] Bosshart, P., Gibb, G., Kim, H., Varghese, G., McKeown, N., Izzard, M., Mujica, F. A., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conf.* (2013).

[13] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[14] Carlson, T. E., Heirman, W., and Eeckhout, L. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis* (2011).

[15] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Symp. on Operating Sys. Design and Implem.* (2018).

[16] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems* (2018).

[17] Chiosa, M., Maschi, F., Müller, I., Alonso, G., and May, N. Hardware acceleration of compression and encryption in SAP HANA. In *Intl. Conf. on Very Large Databases* (2022).

[18] Chiou, D., Sunwoo, D., Kim, J., Patil, N. A., Reinhart, W. H., Johnson, D. E., Keefe, J., and Angepat, H. FPGA-Accelerated Simulation Technologies (FAST): Fast, full-system, cycle-accurate simulators. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2007).

[19] Culler, D. E., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. In *Proc. of Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993).

[20] Darwin: A co-processor for long read alignment. https://github.com/yatisht/darwin. Accessed 1-Dec-2023.

[21] Darwin. Darwin test data. https://github.com/yatisht/darwin/tree/master/RTL/GACT/test_data. Accessed 1-Dec-2023.

[22] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[23] Dennis, J. B. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974* (1974), B. J. Robinet, Ed., Lecture Notes in Computer Science.

[24] Diallo, O., Rodrigues, J. J., and Sene, M. Chapter 11 - Performance evaluation and Petri nets. In *Modeling and Simulation of Computer Networks and Systems*, M. S. Obaidat, P. Nicopolitidis, and F. Zarai, Eds. Morgan Kaufmann, 2015.

[25] Dubach, C., Jones, T. M., and O'Boyle, M. F. P. Microarchitectural design space exploration using an architecture-centric approach. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2007).

[26] Eeckhout, L. *Computer Architecture Performance Evaluation Methods*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[27] Esmaeilzadeh, H., Blem, E. R., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Intl. Symp. on Computer Architecture* (2011).

[28] Facebook: Video transcoding with Mount Shasta. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[29] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A. M., Chung, E. S., Chandrappa, H. K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A. G. Azure accelerated networking: SmartNICs in the public cloud. In *Symp. on Networked Systems Design and Implem.* (2018).

[30] Genbrugge, D., Eyerman, S., and Eeckhout, L. Interval simulation: Raising the level of abstraction in architectural simulation. In *Intl. Symp. on High-Performance Computer Architecture* (2010).

[31] Google HyperProtoBench. https://github.com/google/HyperProtoBench. Accessed 1-Dec-2023.

[32] Google-Intel Infrastructure Processing Unit (IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html.

[33] Guo, Z., Lin, J., Bai, Y., Kim, D., Swift, M., Akella, A., , and Liu, M. Lognic: A high-level performance model for SmartNICs. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2023).

[34] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition* (2016).

[35] Hill, M., and Janapa Reddi, V. Gables: A roofline model for mobile socs. In *Intl. Symp. on High-Performance Computer Architecture* (2019).

[36] Hill, M. D., and Marty, M. R. Amdahl's law in the multicore era. *Computer* (2008).

[37] Huang, J.-C., Lee, J. H., Kim, H., and Lee, H.-H. S. GPUMech: GPU performance modeling technique based on interval analysis. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2014).

[38] Huang, J.-C., Nai, L., Kim, H., and Lee, H.-H. S. TB-Point: Reducing simulation time for large-scale gpgpu kernels. In *Intl. Parallel and Distributed Processing Symp.* (2014).

[39] Hughes, C. J., Pai, V. S., Ranganathan, P., and Adve, S. V. RSIM: Simulating shared-memory multiprocessors with ILP processors. *Computer* (2002).

[40] Intel QAT: Accelerating data compression and encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[41] Ipek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. Efficiently exploring architectural design spaces via predictive modeling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

[42] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symp. on Networked Systems Design and Implem.* (2022).

[43] Iyer, R., Argyraki, K., and Candea, G. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *Symp. on Operating Sys. Design and Implem.* (2024).

[44] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[45] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The case for performance interfaces for hardware accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[46] Jensen, K. Coloured Petri nets: Basic concepts, analysis methods and practical use. *EATCS Monographs on Theoretical Computer Science* (1995).

[47] Joseph, P. J., Vaswani, K., and Thazhuthaveetil, M. J. A predictive performance model for superscalar processors. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2006).

[48] Jouppi, N. P., Yoon, D. H., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P. C., Ma,

X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. A. Ten lessons from three generations shaped Google's TPUv4i : Industrial product. In *Intl. Symp. on Computer Architecture* (2021).

[49] Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Intl. Symp. on Computer Architecture* (2017).

[50] Kaggle. Div2k jpeg image dataset. https://www.kaggle.com/datasets/mingyuouyang/div2k-jpeg-0400.

[51] Kaggle. Flickr image dataset. https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset.

[52] Kahn, G. The semantics of a simple language for parallel programming. In *Information Processing, Proceedings of the 6th IFIP Congress 1974* (1974).

[53] Karandikar, S., Leary, C., Kennelly, C., Zhao, J., Parimi, D., Nikolic, B., Asanovic, K., and Ranganathan, P. A hardware accelerator for protocol buffers. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2021).

[54] Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R. H., Bachrach, J., and Asanovic, K. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Intl. Symp. on Computer Architecture* (2018).

[55] Kim, M. A., and Edwards, S. A. Computation vs. memory systems: Pinning down accelerator bottlenecks. In *Intl. Symp. on Computer Architecture* (2010).

[56] Lee, B. C., and Brooks, D. M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

[57] Lee, B. C., Collins, J. D., Wang, H., and Brooks, D. M. CPR: Composable performance regression for scalable multiprocessor models. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2008).

[58] Li, M., Zhang, M., Wang, C., and Li, M. AdaTune: Adaptive tensor program compilation made efficient. In *Advances in Neural Information Processing Systems* (2020).

[59] Liu, J., Maltzahn, C., Ulmer, C. D., and Curry, M. L. Performance characteristics of the BlueField-2 SmartNIC. https://arxiv.org/abs/2105.06619, 2021.

[60] Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. Offloading distributed applications onto SmartNICs using IPipe. In *ACM SIGCOMM Conf.* (2019).

[61] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. SymPy: symbolic computing in Python. *PeerJ Computer Science 3* (Jan. 2017), e103.

[62] Nayak, N., Odemuyiwa, T. O., Ugare, S., Fletcher, C. W., Pellauer, M., and Emer, J. S. TeAAL: A declarative framework for modeling sparse tensor accelerators, 2023.

[63] Nider, J., and Fedorova, A. S. The last CPU. In *Workshop on Hot Topics in Operating Systems* (2021).

[64] Norrie, T., Patil, N., Yoon, D. H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N. P., and Patterson, D. A. Google's training chips revealed: TPUv2 and TPUv3. In *IEEE Hot Chips Symposium* (2020).

[65] Parashar, A., Raina, P., Shao, Y. S., Chen, Y.-H., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. S. Timeloop: A systematic approach to DNN accelerator evaluation. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software* (2019).

[66] PCI Express Technology. https://www.mindshare.com/files/ebooks/PCI%20Express%20Technology%203.0.pdf.

[67] Perelman, E., Hamerly, G., Biesbrouck, M. V., Sherwood, T., and Calder, B. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Conf.* (2003).

[68] Performance interfaces (project website). https://dslab.epfl.ch/research/perf.

[69] Peterson, J. L. Petri nets. In *ACM Computing Surveys* (1977).

[70] Pourhabibi, A., Gupta, S., Kassir, H., Sutherland, M., Tian, Z., Drumond, M. P., Falsafi, B., and Koch, C. Optimus Prime: Accelerating data transformation in servers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2020).

[71] Protocol buffers. http://code.google.com/p/protobuf/. Accessed on 1-Dec-2023.

[72] Qiu, Y., Xing, J., Hsu, K., Kang, Q., Liu, M., Narayana, S., and Chen, A. Automated SmartNIC offloading insights for network functions. In *Symp. on Operating Systems Principles* (2021).

[73] Ranganathan, P., Stodolsky, D., Calow, J., Dorfman, J., Hechtman, M. G., Smullen, C., Kuusela, A., Laursen, A. J., Ramirez, A., Wijaya, A. A., Salek, A., Cheung, A., Gelb, B., Fosco, B., Kyaw, C. M., He, D., Munday, D. A., Wickeraad, D., Persaud, D., Stark, D., Walton, D., Indupalli, E., Perkins-Argueta, E., Lou, F., Wu, H. K., Chong, I. S., Jayaram, I., Feng, J., Maaninen, J., Lucke, K. A., Mahony, M., Wachsler, M. S., Tan, M., Penukonda, N., Dasharathi, N., Kongetira, P., Chauhan, P., Balasubramanian, R., Macias, R., Ho, R., Springer, R., Huffman, R. W., Foss, S., Bhatia, S., Gwin, S. J., Sekar, S. K., Sokolov, S. N., Muroor, S., Rautio, V.-M., Ripley, Y., Hase, Y., and Li, Y. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2021).

[74] Smith, J. E. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst. 2*, 4 (1984), 289–308.

[75] Sriraman, A., and Dhanotia, A. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 733–750.

[76] Sánchez, D., and Kozyrakis, C. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Intl. Symp. on Computer Architecture* (2013).

[77] Tan, Z., Qian, Z., Chen, X., Asanovic, K., and Patterson, D. A. DIABLO: A warehouse-scale computer network simulator using FPGAs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2015).

[78] Tan, Z., Waterman, A., Avizienis, R., Lee, Y., Cook, H., Patterson, D. A., and Asanovic, K. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Design Automation Conf.* (2010).

[79] Tork, M., Maudlej, L., and Silberstein, M. Lynx: A SmartNIC-driven accelerator-centric architecture for network servers. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2020).

[80] Ultra-Embedded. High-throughput JPEG decoder. https://github.com/ultraembedded/core_jpeg. Accessed 1-Dec-2023.

[81] Veripool. The Verilator simulator. https://www.veripool.org/verilator/. Accessed 1-Dec-2023.

[82] Wang, T., Yang, X., Antichi, G., Sivaraman, A., and Panda, A. Isolation mechanisms for high-speed packet-processing pipelines. In *Symp. on Networked Systems Design and Implem.* (2022).

[83] Wenisch, T. F., Wunderlich, R. E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. C. SimFlex: Statistical sampling of computer system simulation. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2006).

[84] Williams, S., Waterman, A., and Patterson, D. A. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (2009).

[85] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Intl. Symp. on Computer Architecture* (2003).

[86] Zuberek, W. Timed Petri nets definitions, properties, and applications. *Microelectronics Reliability* (1991).

A Artifact Appendix

Our work is part of the umbrella project on performance interfaces [68], which aims to increase performance clarity for systems code and hardware, as well as formally prove performance properties. All artifact materials are available on the project website.

The artifact has three parts: LPN, ltc, and the benchmarks. Both LPN and ltc are available through an `lpnlang` Python package. For the first part, LPN includes several constructs for developers to build their own LPNs. The LPNs we built also serve as examples of how to use those constructs. For the second part, we provide simple scripts to run ltc, which includes using lpn2pi to extract executable performance interfaces, using lpn2smt to verify performance properties, and using lpn2sim for fast performance simulation. Note, a new LPN may contain structures that cannot be analyzed by the tools in ltc; in such cases, ltc will raise an error. Automatically transforming a general LPN as defined in §3 to an LPN amenable to analysis by ltc's tools is left as a future work.

In the third part, we provide benchmarks that are used in the paper and scripts to easily run them. Note that, to simulate the Protoacc RTL, the whole SoC needs to be simulated, and the simulation is slow. The simulation speed we provided in §5 for the Protoacc RTL does not include the SoC simulation. The artifact contains also a Docker image with a ready environment to run the tools and the benchmarks. As LPN simulation normally runs for a very short amount of time, reproducing the results requires that the hardware be kept stable: disable hyperthreading, disable CPU frequency variation, etc.

We will be updating LPNs to improve readability, usability, simulation speed, and accuracy. ltc is also subject to updates that improve accuracy of lpn2pi and/or the time it takes lpn2smt to prove/disprove performance properties. The ltc toolchain will evolve to include more tools and improvements.

A.1 Formal definition of an *input class*

To fully understand how input classes are implemented and used in ltc, we present here a more detailed definition to complement that of §4.1.

A trace e of an LPN is a sequence of transition-commit records, represented as tuples $\langle T_{id}, K_I, K_O, s\rangle$. A trace characterizes the outcome of executing an LPN (as per the definition of "execution" in §4.1). The first component of a record is the transition T_{id} whose commit was recorded. The second and third component is the input set K_I, respectively output set K_O, of token IDs corresponding to the tokens consumed (respectively produced) by the commit of T_{id}. Once a token is consumed, it vanishes forever, so a token identifier can appear at most twice in a trace: as part of the commit that produced it and (possibly) as part of the commit that consumed it. The fourth component of a trace record is a sequence number s that represents the transition's commit timestamp augmented with sequencing information s.t. $\langle T, *, K_O, s\rangle \wedge \langle T', K_I', *, s'\rangle \wedge s < s' \Rightarrow$ transition T committed before transition T' (and thus K_O was available at the same time as K_I'), even if T and T' committed at the same timestamp.

In a valid LPN trace, an input token can never be consumed before it is produced, i.e., for all records $\langle T_{id}, K_I, K_O, s\rangle$ and $\langle T_{id}', K_I', K_O', s'\rangle$, $s < s' \Rightarrow K_I \cap K_O' = \emptyset$

For the rest of this section, timestamps are no longer relevant, so we drop them from our notation. We define the operator $[|.|]$ that, for a given LPN, takes a set of initial input tokens and produces a trace e of the execution of that LPN. We define the relation $\sim_1$ between pairs of traces that determines if the two traces are equivalent modulo "harmless" permutations of records as follows ($l_1 +\!\!+\, l_2$ concatenates sequences l_1 and l_2):

$$\begin{aligned} &\mathrm{pre} +\!\!+ [\langle T_{id_1}, K_{I_1}, K_{O_1}\rangle; \langle T_{id_2}, K_{I_2}, K_{O_2}\rangle] +\!\!+ \mathrm{pos} \\ &\sim_1 \\ &\mathrm{pre} +\!\!+ [\langle T_{id_2}, K_{I_2}, K_{O_2}\rangle; \langle T_{id_1}, K_{I_1}, K_{O_1}\rangle] +\!\!+ \mathrm{pos} \end{aligned}$$

A permutation as shown above is harmless if (1) $id_1 \neq id_2$; (2) T_{id_2} did not consume a token produced by T_{id_1} in the corresponding commit, i.e., $K_{O_1} \cap K_{I_2} = \emptyset$ (trace validity already implies that $K_{O_2} \cap K_{I_1} = \emptyset$); and (3) T_{id_1} and T_{id_2} do not conflict, i.e., they do not share an output or an input place.

We define relation $e1 \sim e2$ as the reflexive, transitive closure of $\sim_1$ over the set of traces of a given LPN. We can now define the set of all traces that are harmless permutations of an initial trace e as $\bar{e} = \{e' | e \sim e'\}$.

Given a trace e, we abstract it by dropping all the token IDs and keeping only the cardinality of the input and output sets in each record. Formally, we obtain the abstract trace $\alpha(e) = \mathrm{map}(\gamma, e)$ by applying the operator $\gamma(\langle T_{id}, K_I, K_O\rangle) = \langle T_{id}, |K_I|, |K_O|\rangle$ to each record in e.

We say that the inputs (i.e., sets of initial tokens) i and i' are in the same *input class* if and only if $\{\alpha(e) | e \in \overline{[|i|]}\} = \{\alpha(e) | e \in \overline{[|i'|]}\}$.

A.2 Input class separation algorithm

Both lpn2pi and lpn2smt rely on a pre-processing step that partitions a user-defined input space into input classes. This pre-processing tool employs symbolic execution [13].

Before diving into the details of the tool, we first define the concept of a *conflict-free* transition. A transition T is conflict-free if and only if its input places are not input places for any other transition, and its output places are not output places for any other transition.

The tool symbolically executes the LPN with symbolic inputs, i.e., input space defined by the user. The tool has three main steps: (1) it first groups LPN transitions into sorted strongly-connected components (SCCs). Transitions are grouped into SCCs according to the edge directions regardless of the edge functions. (2) It then iteratively commits conflict-free transitions. The timestamp of a commit is computed locally based on the tokens (locked in the input places) and on the transition's delay. After a transition becomes enabled, the tool commits it immediately without waiting for a delay and without synchronizing with other transitions' commits. This implies that a commit with a timestamp *ts1* can be materialized before a commit from another transition with *ts2*, where *ts1* > *ts2*, The produced tokens will carry the timestamp of the commit that produced them. (3) Once there are no more conflict-free transitions to commit, it commits one conflicting transition at a time, synchronously (with priority given to transitions in SCCs preceding other SCCs); this implies that the earliest commit is materialized first.

After symbolically executing the LPN exhaustively, the tool will find multiple paths. The input constraints associated with each path defines an input class.

A.3 Estimating $\overline{g}_T$ and ε_T in lpn2pi

To complement the description in §4.2, we provide further details on how $\overline{g}_T$ and ε_T are estimated, as well as assumptions made by lpn2pi.

We first define loops and properties of loops that lpn2pi handles. A loop in LPN is an alternating sequence of places and transitions $P_n \to T_1 \to P_1 \to \ldots \to T_n \to P_n$. Assume, for simplicity, that weights of edges are constants. lpn2pi only handles loops with the following properties:

1. The loop does not fully contain another loop, i.e., no strict subset of places and transitions in this loop forms another loop.

2. The loop has one and only one place with initial tokens.

3. The loop guarantees token conservation. Without loss of generality, assume the place with initial tokens is P_n, and it has M initial tokens. Token conservation means that initial tokens in P_n flow through the transitions, potentially changing in quantity, but eventually all M tokens flow back to P_n. This completes an iteration through the loop. More formally, a loop is token conserving if $\frac{w_{T_nP_n}}{w_{P_nT_1}} \times \frac{w_{T_1P_1}}{w_{P_1T_2}} \times ... \times \frac{w_{T_{n-1}P_{n-1}}}{w_{P_{n-1}T_n}} = 1$, where $w_{T_iP_j}$ is the weight of the edge from T_i to P_j.

Given these assumptions, the loop delay Δ and parallel factors F_{T_i} for each transition T_i are calculated as shown below. Recall that, initially, $\overline{g}_T$ of each transition is set to $T.\delta$, and that $\varepsilon_T = N \times \overline{g}_T$, where N is the number of commits for transition T.

$$\Delta = \sum_{i=1}^{n} \left(T_i.\delta + \overline{g}_{T_i} \times \left(\frac{F_{T_i}}{C} - 1 \right) \right)$$

$$\text{with} \quad F_{T_1} = \frac{M}{w_{P_nT_1}}, \quad F_{T_2} = F_{T_1} \cdot \frac{w_{T_1P_1}}{w_{P_1T_2}}, \quad F_{T_3} = \ldots$$

$$C = \min(F_{T_1}, F_{T_2}, \ldots, F_{T_n})$$

$$\text{and} \quad \overline{g}_{T_k} = \max\left(\overline{g}_{T_k}, \frac{\Delta}{F_{T_k}} \right)$$

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

IronSpec: Increasing the Reliability of Formal Specifications

Eli Goldweber Weixin Yu Seyed Armin Vakil Ghahani Manos Kapritsos
University of Michigan
{edgoldwe, weixinyu, arminvak, manosk}@umich.edu

Abstract

The guarantees of formally verified systems are only as strong as their trusted specifications (specs). As observed by previous studies [22, 52], bugs in formal specs invalidate the assurances that proofs provide. Unfortunately, specs—by their very nature—cannot be *proven* correct. Currently, the only way to identify spec bugs is by careful, manual inspection.

In this paper we introduce IronSpec, a framework of automatic and manual techniques to increase the reliability of formal specifications. IronSpec draws inspiration from classical software testing practices, which we adapt to the realm of formal specs. IronSpec facilitates spec testing with automated sanity checking, a methodology for writing *Spec-Testing Proofs (STPs)*, and automated spec mutation testing.

We evaluate IronSpec on 14 specs, including six specs of real-world verified codebases. Our results show that IronSpec is effective at flagging discrepancies between the spec and the developer's intent, and has led to the discovery of *ten* specification bugs across all six real-world verified systems.

1 Introduction

Formal verification has emerged as a promising technique for increasing the robustness of complex systems by helping developers prove that their implementation meets a formal specification. As promising as this approach is, it has a fundamental Achilles' heel: its guarantees of eliminating *all* bugs in the implementation rely on the *specification being correct*.

The crucial observation that the guarantees of a mechanized proof are only as strong as their specifications is not new and was first identified in 1985 [33]. Specifications (a.k.a. specs) are inherently trusted, rather than proven correct. Relying on trust alone is not enough to ensure that specs remain bug free. If a spec contains a bug, proving that the system meets this spec may be meaningless; the *proven* system could also contain a bug that is hidden by the buggy spec.

The correctness of specifications is the rock upon which the entire edifice of formal verification is built.

Despite the importance of writing correct specs, current best practices rely solely on manual inspection. Developers argue [25, 26] that because specs are typically small compared to the size of the corresponding proof and implementation, it is feasible to manually inspect specs thoroughly enough to ensure that they **capture the intended behavior** of the system. While expert developers are more likely to write correct specs, they are not infallible. As formal verification becomes widely adopted, more and more non-experts will write specs, only exacerbating the risk of introducing bugs. Thus, it is imperative that the process of writing specs be as robust as possible.

In fact, several studies [22, 32, 52], through extensive manual effort, have shown that formally verified systems—many of which were developed by experts in formal verification—contain critical bugs, which originate with problems and inconsistencies in their specs. For example, in January 2022, Notional Finance found a double-spending vulnerability in a deployed verified smart contract missed by manual inspection [34]. In this case, part of the spec was vacuous, causing it to be too weak, and thus the proof would still pass with a *buggy* implementation.

Since a spec is a formal expression of a developer's intent, *proving* the spec correct is ultimately impossible. Ensuring a spec matches a developer's intent will always be best-effort. Whilst no approach can guarantee a bug-free spec, that does not mean attempts to do so must exclusively rely on extensive manual effort and system expertise to resolve. Indeed, there are no structured or automated approaches for a developer to debug this complicated state space. To fill this gap, we propose a means to better handle this challenge.

Inspired by classic testing techniques [17, 24, 48], we introduce IronSpec, a spec testing framework. To enable testing specs, IronSpec adapts the automation of mutation testing [19, 31] and sanity checking along with a customized manual testing approach inspired by unit testing. Together, this framework introduces a systematic way to boost assurance that a spec captures the intended behavior of the system.

If there is a bug in a spec, it originates in the same manner as any other type of bug; there is a disconnect between the intent of the developer and what is written. A spec is *incorrect* if it is too weak, allowing for the existence of even a single implementation that exhibits undesired behavior, or if the spec is too strong, precluding some desired behavior. To identify spec bugs, we leverage the insight that spec bugs manifest themselves as a consequence of *a disconnect of intent* to search for and highlight such disconnects through structure and automation.

IronSpec aids in pinpointing where the developer's intent diverged from the current spec by providing various tools that encapsulate this notion. Common cases where the intent of the developer deviated from the spec can be flagged with IronSpec's Automatic Sanity Checker. If a system has a passing proof, IronSpec leverages this to provide additional automation with spec mutation testing. Mutation testing can automatically identify cases where the behavior of the implementation differs from the spec, by using the proof to identify relevant mutations. The hints of potential intent disconnect provided by automation are bolstered by a manual methodology for writing *Spec-Testing Proofs (STPs)*. STPs are inspired by traditional unit testing and allow developers to test if their understanding of what behavior the spec should allow matches the current spec. STPs can be used to investigate the hints provided by automation to either confirm the existence of a bug or to absolve the disconnect as intended behavior.

We evaluate IronSpec by testing six specs produced in-house, two specs containing artificial bugs that were studied in Abreu et al. [6], and six specs of open-source verified systems. We demonstrate the effectiveness of the automation and manual testing methodology of IronSpec by describing *ten* spec bugs found across a verified Distributed Validator Protocol [4], a verified SAT solver [7], a verified QBFT system [2], a formal spec of the Eth2.0 spec [1], daisy-nfsd [15], and a verified AWS Encryption SDK library [3].

Overall, this paper makes the following contributions:

- Introduces IronSpec, a spec testing framework that allows developers to pinpoint places where the current spec may have diverged from their original intent.
- Proposes an Automatic Sanity Checker, a testing methodology for writing *Spec-Testing Proofs (STPs)*, which are applicable to test specs even in the absence of a completed proof or implementation, and describes how to adapt mutation testing to specs to automatically identify divergences between the spec and the implementation.
- Demonstrates the effectiveness of IronSpec, by illustrating how we applied IronSpec to *six* real-world, verified systems leading to the discovery of *ten* spec bugs.

2 Manually Scrutinizing Specifications

Relying on manual inspection alone to ensure an intended specification is not practical. Fonseca et al. [22] performed a study aimed to challenge the assumption that just because a system is verified, it is bug-free. In this study, the authors thoroughly examined three formally verified distributed systems, IronFleet [26], Verdi [53], and Chapar [41] and identified sixteen bugs across their specifications, verification tools, and their unverified shim layers. Two of these bugs were found to be in specifications. This study was chiefly manual and required close examination of the respective specifications to identify. The authors do introduce some basic automation, yet their techniques still rely predominantly on manual effort and expertise in the system. This work demonstrated the need for and acknowledges the lack of a more rigorous and automated approach to testing formal specifications. Similarly, Yang et al. [54] conducted a bug study of compilers and discovered two bugs within the verified compiler CompCert due to underspecification, and similarly observed that specifications are complex and lack scrutiny.

The concerning discovery of these previous works identifies the gap that this work aims to fill; to provide a means for developers to help automatically and methodically identify specification bugs across the spectrum of specifications.

Complicating this problem, specifications can take on different forms, making uniform debugging approaches difficult. In their simplest form, specifications can be in-line predicate assertions [29]; boolean functions that check the state of the system against some property. A more specific class of predicate assertions based on the Floyd-Hoare style logic [21, 28] are preconditions and postconditions, which establish invariants about the state of the program before and after the execution of a piece of code. For more complex systems, rather than directly proving properties about the system, it can be easier [26, 53] to prove state machine refinement [37]. For refinement, the specification is an abstract state machine that encapsulates the desired behavior of the system.

To highlight the subtlety of trying to manually ensure a specification is correct, consider an *incorrect* specification for a simple `Sort` method found on line 3 of Specification 1. This `Sort` method takes a sequence of integers as input and promises to return a sorted sequence of integers in ascending order. The specification for this method is a single postcondition which ensures that the value at every index in the output sequence is less than or equal to the value at subsequent indices. At first glance, this may seem to be a correct specification for `Sort`—a mistake that many newcomers to verification make.

However, this specification is incorrect, as it neglects to mention any relationship between the input and output sequences. This is considered a buggy specification because a proof could still pass even with an *incorrect* implementation that exhibits undesired behavior, erroneously giving the illusion of correctness. For example, if the input sequence were $[1,6,7,2]$, an incorrect implementation could arbitrarily return $[1,42,100]$ or even the empty sequence $[\,]$. The *incorrect* implementation for `Sort` in Specification 1 is triv-

Specification 1 Incorrect Sort Spec

```
method Sort(input:seq<int>)
  returns (out:seq<int>)
  ensures forall i | 0 <= i < |output| - 1 ::
           out[i] <= out[i+1]
{ return []; }
```

Specification 2 Correct Sort Spec

```
1 method Sort(input:seq<int>)
2   returns (out:seq<int>)
3   ensures forall i | 0 <= i < |output| - 1 ::
4            out[i] <= out[i+1]
5   ensures multiset(input) == multiset(output)
6 { /* body omitted */ }
```

ial and always returns an empty sequence. Yet, the proof for this method would still pass, as this trivial implementation satisfies the incorrect, too-weak specification.

Manually identifying a spec bug, like that in Specification 1, can be challenging. In fact, a correct specification for `Sort` should also capture the relationship between the input and output by adding an additional post-condition to ensure that the multiset of the input is equal to the multiset of the output, see line 5 in Specification 2.

The opposite case, where a specification is too strong, can be equally as important and challenging to manually identify. For example, if we replace line 5 in Specification 2 with `ensures input == output`, the specification becomes unnecessarily strong. Multisets do not take order into account, whereas sequences do, so the updated postcondition is overly strong. The only input and output pair that could satisfy this specification is if the sequences are identical and already in ascending order. Even if one has a correct implementation of `Sort`, proving that the implementation upholds this specification is impossible. To debug the inevitably failing proof, the developer must examine their implementation for bugs, check their proof for missing invariants *and* manually inspect their spec to make sure it captures the intended behavior. Having high confidence in the spec would make this scenario much more unlikely and would give the developer more time to focus on the proof itself, knowing they are proving the right property.

3 How To Test A Specification

It is challenging to diagnose spec bugs because specs are trusted, and a buggy spec can often be at odds with a developer's original understanding of the system. Complicating the problem, specs are often intended to be abstract, allowing different, correct implementations to meet the spec. Hence, we introduce IronSpec, a framework for testing specs to help gain confidence that a spec is bug-free. This work represents the first systematic effort to bridge the gap between the mature and extensive work in software testing and the lack of rigor in ensuring spec correctness.

IronSpec is inspired by the insight that the existence of a spec bug is inherently due to a disconnect between what the developer intended and what properties were actually captured in the spec. IronSpec provides tools to allow a tester to identify and test possible occurrences where the original intent of the developer may have diverged from the current spec. Some aspects of IronSpec only rely on the spec and have no dependence on the existence of an implementation or a passing proof. However, if there is an implementation and a corresponding passing proof, IronSpec can leverage this to use the implementation as an additional reference point to help focus the testing process.

This section introduces and provides a high-level overview behind the ideas of why each testing component of IronSpec is useful in exposing disconnects between the intent of the developer and their spec. Section 4 discusses each in more detail.

3.1 Testing Specifications In The Absence Of A Passing Proof

Akin to test-driven development [10], it is desirable to test a spec without requiring a proof or corresponding implementation. If there is a bug in the spec when it comes time to write a proof, a developer may struggle and expend unnecessary manual effort in debugging in the wrong place. The Automatic Sanity Checker and *Spec-Testing Proofs (STPs)* provide two frames of reference for a tester to check their specs against, even in the absence of an implementation and proof.

Regardless of the context of the system, it is clearly never intended for a verified method to be permitted to return arbitrary values. If the spec is too weak, an incorrect implementation might be free to return *any* value, unconstrained by the spec. The Automatic Sanity Checker raises high-confidence flags when the spec of a verified method fails to properly constrain its output based on the given input. The sanity checker also alerts the developer to partially constrained input and output, which provide weaker hints to the existence of spec bugs but are also worthwhile to investigate further.

Because the Automatic Sanity checker only looks for underconstrained input and output, this technique can be used even in the absence of an implementation or proof. Section 4.1 describes the Automatic Sanity Checker in more detail.

The Automatic Sanity Checker excels at automatically finding common spec bugs by leveraging generic code patterns, but cannot leverage any user-provided hints and insights. We address this gap by introducing a methodology for manually writing *Spec-Testing Proofs (STPs)*. STPs are inspired by traditional unit tests and are proofs about the spec for context-specific input and output. STPs help developers expose differences between the expected behaviors they intend to include in the spec and what is currently permitted. This testing methodology is useful in the presence of a passing or failing proof, but can also be applied in the absence of a proof. Section 4.2 explains how to write STPs and interpret their results.

An STP is, by definition, a proof; this is the key difference between STPs and standard unit tests. Since STPs are proofs, STPs help to answer different questions than what unit tests allow for. Instead of attempting to prove a general property, an STP demonstrates the validity of the spec for a specific, concretized value or a range of values. This testing methodology exploits the insight that crafting proofs for specific cases is often less challenging than producing a comprehensive proof and can frequently be proved by the verifier with minimal manual intervention. Each STP is a small proof about *distinct* properties of the spec. The steps of writing STPs are generic, and so can be useful tools in investigating the correctness of many variations of specs.

Differing from a failed unit test, if an STP fails to verify, it could be for various reasons. The STP may fail due to a divergence between the expectation of the spec and the STP, indicating a bug. If the tester suspects that a disconnect caused the failed proof, the appropriate next step is to write a concrete *Counterexample* STP. The counterexample proves that unintended behavior is permitted by the spec. Alternatively, an STP may fail because the STP body lacks sufficient proof annotations for the verifier to prove the final postcondition. Distinguishing between a spec bug and the need to add proof to the body of the STP is impossible to immediately diagnose for every case because in this work we are targeting undecidable programs.

3.2 Testing Specifications With The Assistance Of A Passing Proof

Even when a system is verified with a passing proof, it is still possible for the system to contain bugs if the spec itself is buggy; thus testing a spec at this point is still very valuable. A too-weak spec could allow for a proof to pass with an incorrect implementation, falsely giving the illusion of correctness. Alternatively, even if the current implementation contains no bugs, a too-weak spec could allow for a buggy update to the current implementation, such that a proof would still pass with the same too-weak spec. Relying on a developer to write a bug-free implementation given a buggy spec, goes against the very reason to verify systems in the first place; so it is just as vital to identify spec bugs when the proof passes.

Using the Automatic Sanity Checker and writing STPs are applicable when testing a spec with a passing proof, but the proof and implementation together contain untapped information that can further assist testing. Like the spec, the implementation also captures the intent of the developer. Identifying the difference of intent between the behavior allowed by the spec and what is actually in the implementation, calls the developer's attention to potential disconnects. IronSpec can take advantage of the proof and implementation to automatically test a spec with *mutation testing*. Mutation testing identifies cases when the spec is weaker than the current implementation. IronSpec uses the passing proof as a reference point to automatically distinguish cases where the existing implementation is weaker than the behavior allowed by the spec. Further details concerning how IronSpec adapts mutation testing to specs are described in Section 4.3.

Departing from traditional mutation testing, IronSpec starts with a spec, implementation, and passing proof and then only mutates the spec. IronSpec relies on an existing passing proof to indicate whether a mutation should be killed, whereas traditional mutation testing relies on a test suite. A mutation is kept and considered *alive* if the original proof still passes with the mutated spec, indicating that the implementation also meets this different spec. The behavior allowed by the original spec but not the mutated spec serves as an example of a subset of behavior that may not be intended.

The existence of even a single alive spec mutation serves as a flag to the developer. An alive mutation is clear evidence that a different spec still allows the proof to pass with the unmodified implementation, and represents specific behavior unaccounted for by the original spec. The difference between the original spec and the passing mutated spec is a strong hint for the tester to determine if that specific behavior is intended. Identifying alive mutations is accomplished automatically, but understanding the implication of any such alive mutation cannot be automated and ultimately still relies on the developer's intuition to understand.

An alive mutation is simply a hint highlighting a divergence between the spec and the implementation. However, not all alive mutations immediately lead to the discovery of a spec bug. If a spec is correct but also weaker than the current implementation, there is a chance for an alive mutation to be considered a false positive and marked as intended behavior. In a contrasting, albeit rare case, if both the spec and implementation are buggy, but the implementation is not weaker than the buggy spec, then no alive mutations may be found.

To reduce the chance of false positives only a subset of the generated mutations are eventually considered. Logically equivalent or weaker mutations than the original spec and mutations that trivially make the proof pass can be safely ignored. The details for how specs are mutated and what constitutes valid mutations are expounded upon in Section 4.3.

Note that we deliberately mutate only specs and not implementations for two reasons. Firstly, specs are smaller than implementations, therefore reducing the number of mutations necessary to consider. Secondly, mutating only the spec rather than the implementation is advantageous for automation. Specs, being boolean functions, enable automatic filtering of irrelevant mutations. Assuming the proof passes given the original spec, any logically weaker spec mutation will still allow the proof to pass and does not provide any new relevant information. By automatically checking the relative logical strength of a mutated spec in relation to the original, weaker mutations can be identified and ignored. This automation is impossible when mutating the implementation, as determining relative logical strength is not possible in all cases. Logical relationships can be determined automatically for boolean functions, like specs, whereas not all imperative code shares this attribute. Implementation-based mutations would increase the manual burden on the tester, as many more false positives would be an unavoidable outcome that would require manual effort to sift through. The process of automatically filtering spec mutations is further explained in Section 4.3.2.

In certain cases, mutation testing is also useful in identifying too-strong specs. A spec in the Hoare-Logic style can also be considered incorrect by virtue of having a too strong precondition. IronSpec's mutation testing is still applicable in this case. If the spec mutation target is a precondition, rather than attempting to identify where the spec is disconnected from the implementation due to weakness, IronSpec reverses the criteria used to determine relevant mutations by considering mutations that are *weaker* than the original spec.

Mutation testing does not provide complete coverage of spec testing but rather focuses the attention of the tester on a disconnect between the spec and the implementation. STPs can be used to help fill this gap. Focusing on writing STPs about the discrepancy hinted at by an alive mutation leads to a more efficient way of identifying bugs. STPs guided by the hint of alive mutations can allow a tester either to arrive at a counterexample, showing a bug in the spec, or to absolve the alive mutation as intended behavior.

4 The IronSpec framework

IronSpec consists of three spec testing tools; an Automatic Sanity Checker, a methodology for writing *Spec-Testing Proofs (STPs)*, and an automatic mutation testing framework. Each assists in identifying and flagging divergences between the developer's intent and the existing spec.

The IronSpec prototype is built in C# as an extension to Dafny [40], a verification-aware programming language that enables verification with the Z3 SMT solver, and also supports practical imperative implementations by compiling to C#, Java, JavaScript, and Go. IronSpec was applied to test specs written in Dafny, but the concepts of how to test specifications are not Dafny-specific and could be re-implemented in other environments.

4.1 Automatic Sanity Checker

The Automatic Sanity Checker (ASC) examines the input, output, and spec of verified methods to identify cases where the spec may be weaker than intended. The ASC implementation consists of approximately 300 lines of C# code and achieves this check by traversing the AST of the method under test while maintaining some local state. Table 1 outlines the properties that are checked and their assigned severities. All of these properties can be determined by examining the AST, and as such, they can be checked efficiently without invoking a verifier. Either of the HIGH severity flags signifies a high likelihood of spec bugs, whereas the other severity levels indicate a cause for additional manual inspection. Both HIGH severity flags reveal a weakness in the flagged spec. If the postconditions do not depend on the input, then the weakness is in regard to the lack of necessity for that input. Whereas, if the postconditions do not constrain the output, an implementation with a passing proof could return arbitrary output values. Regardless of the particular functionality of the system, either case represents a clear disconnect between the intent of a correct spec and the current spec.

Table 1: Automatic Sanity Checking Flags

Flag Severity	**Condition**
LOW	Post conditions only depend on a portion of the input
MED	Only part of the output is constrained by the postconditions
HIGH	None of the postconditions depend on any of the input
HIGH	None of the output is constrained by any of the postconditons

The power of the Automatic Sanity Checker arises from exploiting the relationship between a spec and the input/output of its corresponding method. Both HIGH severity flags signal the condition when the spec constraints on the input/output of the method are non-existent. If no postcondition depends on any of the input values, then an obvious aspect of the spec is missing. The buggy sort spec in Specification 1 exemplifies this scenario. The spec is not constrained at all by the input, making the spec weak enough to allow for a proof of an incorrect implementation to pass. Similarly, if a method has an output not constrained by its postconditions, an incorrect implementation can return *any* output. The lower severity flags hint to partial violations of the general properties and do not immediately indicate bugs; rather, they signal a missing part of the spec that could be the source of a bug.

4.2 STP Methodology

The testing methodology outlines four classes of STPs. The first three help guide developers in understanding the **Usefulness**, **Correctness**, and **Provability** of their specs. Lastly, if there is a bug in the spec, developers can prove its existence with a **Counterexample** STP. The methodology focuses on specs written following the Hoare-Logic style [51] but can be applied to any type of predicate-based spec. All types of STPs enable the developer to prove a specific property about their spec. A developer proves that context-specific input and

Lemma 3 General Precondition STP

```
lemma PreconditionSTP(in:InType)
  requires TestInputProperty(in)
  ensures Precondition(in)
          // or !Precondition(in)
```

Lemma 4 General Postcondition STP

```
1 lemma PostconditionSTP(in:InType,out:OutType)
2   requires TestInputProperty(in,out)
3   ensures Precondition(in)
4   ensures Postcondition(in,out)
5           // or !Postcondition(in,out)
```

output values are valid or invalid, and gauge if those results match their *intent* based on their understanding of what the spec should or should not permit. Lemma 5 is an example of an STP which tests if a sort spec is strong enough to reject specific *invalid* values. A passing STP shows that the intent of the developer matches the spec and is a proof for that particular property of the spec.

4.2.1 Writing STPs

The construction of different types of STPs share many similarities, but the results are interpreted differently. STPs also enable decoupling of pre- and postconditions so that they can be tested individually. The general form for these STPs are found in Lemmas 3 and 4.

Usefulness STPs help to answer the question of whether the preconditions are weak enough to remain useful; the preconditions should accept all intended valid inputs. Usefulness STPs follow the general form of Lemma 3. The specific input values are defined as part of the precondition for this lemma as the `TestInputProperty`, and should be a value that the test writer *expects* to be a valid input allowed by the spec. The postcondition for a Usefulness STP should be the preconditions from the spec, i.e. `ensures Precondition(in)`.

Correctness STPs examine whether the postconditions are strong enough to reject all intended invalid outputs. Writing Correctness STPs is based on the general form of Lemma 4. To test if the postcondition is strong enough to reject buggy behavior, the test writer supplies an output value that is *expected* to be invalid and should not be allowed by the spec i.e. `ensures !Postcondition(in)`. To isolate testing the postcondition from the precondition, the test writer should also prove that the undesired output does not satisfy the spec as a result of an invalid input value (Line 3 in Lemma 4), ideally with a separate Usefulness STP validating the input.

Conversely to Usefulness and Correctness STPs, Provability STPs test whether the preconditions are strong enough and whether the postconditions are weak enough for the existence of a provable implementation. Provability STPs are most useful before having a passing proof, as a passing proof is evidence that the spec has this property. That said, they can still provide value in the presence of a passing proof, as they can test the strength of transitions in a state machine (see Section 5.2.2).

Lemma 5 Correctness STP Example - Incorrect Sort Spec

```
lemma CorrectnessSTPSort(
    input:seq<int>, sorted:seq<int>)
  requires input == [42, 1, 500]
  requires sorted == [42, 500]
  ensures !SortSpec(input, sorted)
{ }
```

STPs for Provability are concerned with both preconditions and postconditions, thus follow the structure from both Lemmas 3, and 4. Precondition STPs prompt the test writer to prove that *expected* invalid input should not pass the precondition, i.e. `ensures !Precondition(in)`. Whereas postcondition STPs check that input and output *expected* to be permitted by the spec is allowed by the postconditions, i.e. `ensures Postcondition(in,out)`.

If suspecting a spec bug, a test writer can also directly write a Counterexample STP. A passing Counterexample STP is concrete evidence of a bug in the spec. Counterexample STPs can take on two different forms but are still derivative of Lemma 4 if concerned with postconditions and Lemma 3 for preconditions. A Counterexample STP can either show that an expected **valid input-output pair is rejected** by the spec or that an expected **invalid input-output pair is accepted**.

4.2.2 Adding Proof Help To STPs

When an STP fails to verify, it could be due to a divergence between the expectations of the test writer and the current spec, indicating a spec bug, which can be confirmed with a Counterexample STP, or it could be the result of the fundamental undecidability of this type of problem. If it is the latter case, it is possible to circumvent this roadblock in some instances by adding additional proof to the STP body.

The process of proving an STP is no different than writing a proof for any lemma, but the specificity of the STP narrows the scope necessary to reason about. However, before spending the manual effort to add proof annotations to the body of an STP, the first step is to negate the conclusion, i.e. the ensures of the STP. Negating the conclusion transforms an STP into a Counterexample STP. Thus, if the proof now passes there is a clear indication of a bug.

As an example of the process of writing an STP, consider the **Correctness** STP in Lemma 5 for the incorrect sort spec from Specification 1. This STP tests that the sort spec should reject the case when the output sequence is sorted, but only contains a subset of the original input. The Sort method does not have a precondition, so any input sequence would satisfy a Usefulness STP, so this step can be skipped. Running a verifier on this STP would initially result in failing to prove the postcondition automatically. Before spending manual effort to

Table 2: Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
COI	Constant Operator Insertion
UOR	Unary Operator Replacement
ENO	Expression Negation Operator
VNOR	Variable Name Operator Replacement
SOR	Set Operator Replacement
HOR	Heap Operator Replacement

prove this STP, the first step is always to negate the postcondition (i.e. changing Line 5 to `ensures SortSpec(input, sorted)`), transforming the Correctness STP into a Counterexample STP. The attempt to prove this counterexample would now pass and serves as a concrete example of where the spec has diverged from the test writer's understanding.

4.3 Mutation Testing

If the system has a passing proof, IronSpec can leverage the proof and implementation as a reference point for further automation. IronSpec systematically generates a set of specification *mutations*, slight syntactical modifications of the original spec, but only considers those that are not weaker than the original spec. If the original proof still passes with one of these stronger specs, this alerts the developer to the original spec being weaker than the implementation; a disconnect that may hint at an unintentional spec weakness.

All mutations are subjected to three verification-assisted checks, outlined in the following subsections. Each of these checks filter the set of mutations by discarding irrelevant mutations; any discarded mutation is deemed *killed*. If a mutation is still *alive* after all three checks, it serves as a hint of a potential spec bug.

4.3.1 Mutation Generation

We generate mutations inspired by the method-level mutation operators from MuJava [44,45] and from a study that used the Z3 SMT solver to optimize a set of mutation operators based on subsumption relationships [23]. We further introduce an additional predicate-based mutation operator, Set Operator Replacement (SOR). SOR introduces mutations about set inclusion, for example, an expression, $e \in s$, would be mutated to become, $e \notin s$ or vice versa.

The IronSpec prototype is implemented in Dafny, so all mutations are applied to expressions in the Dafny AST. For Dafny expressions that reason about the heap, we introduce the Heap Operator Replacement (HOR) mutation operator, which mutates expressions containing the Dafny keyword `old`. The full list of the mutation operators used in IronSpec is shown in Table 2.

Lemma 6 *IsAtLeastAsWeak* Lemma

```
lemma IsAtLeastAsWeak(p:Params)
    requires OriginalPredicate(p:Params)
    ensures MutatedPredicate(p:Params)
```

Predicate 7 *Mutation* Target Example

```
predicate SafetyProperty(p:Params)
{ SubPredA(p) ==> SubPredB(p) }
```

Each generated mutated spec is the result of IronSpec applying a single mutation operator at a time. The set of all mutated specs consists of all possible single-operator mutations for a given spec applied to each subexpression in the mutation target.

An example of one of the many possible spec mutants starting with the single postcondition from Specification 1 would be: `forall i | 0 <= i < |output| - 1 :: out[i]` **<** `out[i+1]`. This mutation is generated using the Relational Operator Replacement (ROR) mutation operator which generates mutations by replacing relational-based operators from the set of `{==, <, <=, >, >=, !=}`. One application of this mutation operator results in replacing the <= to a < in the RHS of the forall expression.

4.3.2 First Pass: Logical Redundancy

Not all mutations produced from the original spec are relevant. A spec defines a set of behaviors, and a passing proof shows that the behavior of a specific implementation is a subset of the behavior allowed by the spec. A spec that is weaker than the original would allow a larger set of implementations to satisfy this subset property. Any mutated spec that is logically equivalent or *weaker* than the original spec would not provide any new information to the tester about the current implementation and can be safely disregarded.

A mutation can cause a spec to become weaker if it weakens a postcondition or if it strengthens a precondition. Either case allows for a larger set of implementations to satisfy the spec. Therefore, for each mutation to a postcondition, IronSpec tests if the mutation *is at least as weak* as the original spec.

Definition 4.1. Given predicates S and S' with parameters p, S *is at least as weak* as predicate S' iff $\forall p.S'(p) \implies S(p)$

IronSpec captures this definition by automatically formulating Lemma 6 for the original and mutated specs. If this lemma passes, then the mutated spec must be equivalent to or weaker than the original spec, indicating that it can be killed.

Conversely, if the mutation modified a precondition, IronSpec checks the opposite, to see if the mutation *is at least as strong* as the original spec. The lemma to check if a spec is at least as strong as the original is similar to Lemma 6, but with the requires and ensures reversed.

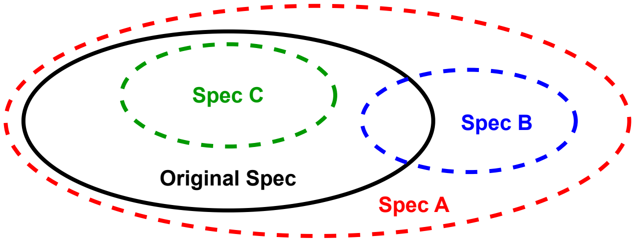


Figure 1: A mutated spec can either be strictly weaker (Spec A), strictly stronger (Spec C), logically equivalent, or partially stronger and weaker (Spec B) than the original spec.

As an example of Definition 4.1, consider Figure 1, where each circle represents the set of behaviors allowed by each respective spec. Any behavior in the circle encapsulated by the Original Spec is still inside the set of allowable behaviors of Spec A, making Spec A strictly weaker than the Original Spec and thus would be automatically discarded based on the result of Lemma 6. Both Specs B and C are not at least as weak as the Original Spec and would survive the *Logical Redundancy* pass.

The *IsAtLeastAsWeak* lemma is generated automatically to test the spec's overall safety property, rather than directly testing the mutated expression. This is important to avoid false positives. For example, consider if `SubPredB(p)` in Predicate 7 contains the mutated expression. Testing the *IsAtLeastAsWeak* Lemma with just `SubPredB(p)` may fail, indicating that this mutation is stronger than the unmodified version of `SubPredB(p)`, but this mutation may cause its caller, Predicate 7, to become weaker.

4.3.3 Second Pass: Vacuity

IronSpec's second pass aims to identify the mutations that cause vacuity [36]. For example, if a mutation to `SubPredA(p)` from Predicate 7 resulted in it always evaluating to *false*, then Predicate 7 would always be true. A vacuous spec would allow for the system's proof to pass trivially because any behavior of the system would be allowed by the vacuous spec. Checking vacuity is more complicated than purely checking if the mutated predicate is itself vacuous, as the conditions of a predicate that calls the mutated predicate, in conjunction with the mutated predicate could result in the caller becoming vacuous. This is especially important with specs that are state machines where a mutation could cause a state transition to become *false*, removing that behavior from the spec. IronSpec automatically generates a lemma to check for vacuity by considering the full call path.

4.3.4 Third Pass: Full Proof

The final check is to see if the full proof will pass with the mutated spec. In this final pass, the system is re-verified with the addition of the mutated spec to ensure that no intermediary lemmas now fail. If the full proof passes, the mutation is considered *alive* and serves as a flag to the developer to reexamine the spec.

4.3.5 Hierarchical Classification of Alive Mutations

Rather than providing the tester with a list of all alive mutations, IronSpec performs an additional pass to characterize the alive mutations, minimizing the output to the most relevant. To maximize the hint provided by an alive mutation, IronSpec evaluates the set of alive mutations to calculate a Direct Acyclic Graph (DAG) indicating which mutations are weaker or stronger in relation to one another. The DAG is structured so that each node is stronger than all of its children. The tester need only further concern themselves with the root of each connected component of this *mutation DAG*, as all children are weaker than the root in each component. This hierarchical classification is inspired by previous research to classify and remove equivalent mutations [8,23,46,50].

4.4 Using Alive Mutations As Hints For STPs

When testing specs, human intuition is always the final oracle, thus, STPs are still needed to finish the investigation started by mutation testing. On their own, alive mutations only indicate a relative divergence between the spec and the implementation, but these hints can be used to write focused STPs. The relative strength of an alive mutation can be used to shrink the state space necessary to test, focusing on the divergence between the original spec and the mutation.

Armed with an alive mutation, a test writer can effectively exploit its hint by deviating from the standard guidelines of writing STPs and work **backwards** from the *spec difference*. The spec difference is the set of behaviors allowed by the original spec S and not by the alive mutation S'; essentially $S - S'$. The spec difference embodies the fundamental insight IronSpec is based on; it captures a specific disconnect between the original spec and the implementation. The behavior allowed by this reduced expression is permitted by the original spec, but not by the more restrictive mutation. The spec difference uniquely presents the tester with this subset of behavior to determine if that particular disparity is intended.

Working backwards allows the test writer to find concrete values that satisfy only the spec difference, achieving more concentrated STPs. Typically, when writing STPs, a tester starts by manually specifying values they intend for the spec to allow or disallow. This process increases in difficulty with the additional constraint that these intended values also need to satisfy the spec difference. The shift of working backwards helps to alleviate this burden.

Driven by the insight that the actual semantic change between the mutation and the original spec is small—only a single mutation—the expression of the spec difference is min-

Table 3: Spec bugs identified using the Automatic Sanity Checker. All bugs were confirmed with Counterexample STPs based on the initial hint of either MED or HIGH flags.

Bug	Specification	Method Name	Flag
TS1	TrueSat [7]	Formula Ctor	HIGH
TS2	TrueSat [7]	Start	MED
ETH1	Eth2.0 [1]	on_block	MED
AES1	AWS ESDK [3]	Encrypt	MED
AES2	AWS ESDK [3]	Decrypt	MED

imal. Working backwards allows the test writer to generate *any* input and output, and then use the verifier to check that the input and output are accepted by the expression constituting the spec difference. After generating such values, the final decision relies on the tester to decide whether the input-output pair is intended. At this stage, the existence of an *unintended* value is a counterexample to the original spec.

5 Evaluation

We evaluate the effectiveness of the IronSpec prototype by applying the Automated Sanity Checker, the STP Methodology and the automated mutation testing framework to test 14 different specifications written in Dafny [40]. Six of these specifications are produced in-house and include artificially introduced bugs, with an additional two specs containing artificial bugs described by Abreu et al. [6]. Six of the specifications are of real-world, open-source verified systems, which include: QBFT [2], DVT [4], TrueSat [7], Eth2.0 [1], daisy-nfsd [15] and an AWS Encryption SDK library [3].

When testing a spec, the ultimate oracle is the test writer, thus the final step is always to write an STP. When testing a spec, a tester could start with any aspect of IronSpec. We discuss the various facets of IronSpec by highlighting their use in supplying the initial hints used to discover *ten* spec bugs, all confirmed by their corresponding authors.

We consider all spec bugs identified and discussed in this section useful and significant; all could have allowed or did allow an incorrect implementation that would violate safety **while still allowing the proof to pass**.

The IronSpec artifact is publicly available on GitHub [5].

5.1 Automatic Sanity Checking Evaluation

Applying the Automatic Sanity Checker to the six open-source verified systems led to the discovery of five spec bugs across three specs, listed in Table 3.

Of the bugs identified, only TS1 was identified immediately with a HIGH severity flag, whereas the other four bugs were each discovered in less than an hour by writing STPs based on the hint of MED severity flags. The corresponding implementation for all five spec bugs appeared correct, but the specs were buggy, being too weak. To confirm these spec bugs, we wrote buggy implementations as Counterexample STPs for each spec and demonstrated that the proof still passed.

Spec bugs TS2, ETH1, AES1, and AES2 were identified by investigating each respective MED severity flag. We found that in these cases, the bug was a result of the output consisting of a complex datatype with many sub-fields and having postconditions concerning only a subset of these fields. This combination allows for a different implementation to update the remaining unspecified fields arbitrarily.

A MED severity flag is not as strong of a hint of a spec bug as a HIGH severity flag because *the unspecified fields may or may not be critical for safety*. The HIGH severity flag raised for TS1 was; "None of the postconditions depend on any of the input." This spec bug allows a buggy implementation to completely ignore the input values when constructing the output. The authors have remedied bugs TS1 and TS2 with a pull request we submitted.

The two bugs, AES1 and AES2, from the AWS Dafny Encryption SDK library (ESDK) [3] are both cases of spec weakness. The Dafny ESDK is a verified SDK used as a reference to build ESDKs for other languages. These bugs exist for the high-level methods of `Encrypt` and `Decrypt`. They are caused by a combination of the postconditions under-constraining the output and because the postconditions of sub-methods are not exported. This underspecification allows for the proof of trivially incorrect implementations for `Encrypt` and `Decrypt` to pass, such as returning a ciphertext or plaintext consisting of a zero byte regardless of the input.

Specs with output containing complex datatypes with many sub-fields are a critical source of spec bugs. Judging from the results of applying the Automatic Sanity Checker, under-constraining complex output can easily be overlooked. To avoid these types of spec bugs, it is vital to specify the expected values for all sub-fields of the output.

5.2 STP Methodology Evaluation

In this section we describe our experience in writing Usefulness, Correctness, and Provability STPs for specs following the Hoare-Logic style; and in the cases of identifying a spec bug, Counterexample STPs. We discuss the effectiveness of these STPs to expose differences in what behaviors the spec allows in contrast to a test writer's expectations in the presence of artificially introduced bugs.

We also discuss a case study, where following the STP methodology we discovered three spec bugs in a verified QBFT protocol. We wrote STPs for all open source specs, and when an STP failed to verify and the result conflicted with our understanding of the spec, we wrote Counterexample STPs to prove the existence of a spec bug. For brevity, the case study in this subsection focuses on the QBFT spec, where STPs acted as the initial flag that hinted at the existence of a spec bug.

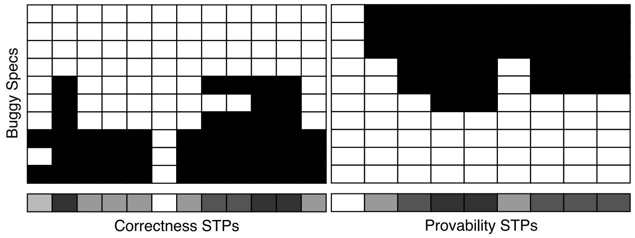


Figure 2: STP coverage for various buggy sort specs

5.2.1 STPs For Contrived Spec Bugs

We wrote STPs for five specs written in-house that follow the Hoare-Logic style. These specs include methods for finding the max of two integers (*Max*), sorting a sequence of integers (*Sort*), searching for an integer in a sequence (*Binary Search*), a cryptocurrency token creation contract (*Token-with-revert-external-wre*), and an auction contract (*SimpleAuction-with-revert-external-wre*). The *Token* and *SimpleAuction* specs were modified from Cassez et al. [14].

We wrote a total of 70 STPs for all variations of the contrived specs. The bugs introduced into the specs vary, but the resulting specs comprised an approximately equal split between too-strong and too-weak specs. Only 30% of these STPs (21) needed additional proof help and of those, each STP needed, on average, an additional 1.7 lines of proof help. The STP suite was successful in all cases in identifying introduced spec bugs—confirming the notion that a failing STP is a reliable flag suggesting a discrepancy between the intent encoded in the STPs and the spec.

Not all classes of STPs are useful in identifying all spec bugs because of the different natures of each type of STP. To demonstrate why writing a diverse suite of STPs is useful, consider Figure 2 that shows the coverage of 21 STPs across 10 different bugs for a `Sort` spec. In this experiment, the suite of STPs consisted of Correctness and Provability STPs because the different `Sort` specs were all postconditions, and Usefulness STPs are only concerned with preconditions. Each row corresponds to a different introduced bug and each column matches to a different STP. A darkened cell indicates that a specific STP successfully identified the bug after transforming the failing STP into a passing Counterexample STP. The bug was identified if a row contained a **single darkened cell**, whereas each column gives insight into the **coverage** of a single STP in identifying different bugs. The bottom row is a heat map corresponding to the ratio of bugs identified by each STP. The two STPs that uncovered zero spec bugs tested trivial enough cases such that they passed with all of the buggy specs. Depending on the spec bug, there were cases where the Correctness STPs were insufficient to identify the bug on their own, and there were cases where the same was true of the Provability STPs. So, when writing STPs it is important to have good coverage of different types of STPs to increase testing effectiveness.

Lemma 8 Simplified QBFT Provability Adversary STP

```
lemma AdversaryForwardMessageSTP(
    a:Adversary,
    a':Adversary,
    inMsgs: set<Message>,
    outMsgs: set<Message>)
  requires validAdversaryConfig(a,a',inMsgs)
  requires inMsgs == {ProposalMsg(CS1,block)}
  requires outMsgs == {NewBlockMsg(CS1,block)}
  ensures AdversaryNext(a, a', inMsgs, outMsgs)
```

5.2.2 STPs: QBFT Case Study

Writing STPs for the QBFT spec [2], a Byzantine fault-tolerant consensus protocol used in the Ethereum ecosystem [47], led to the discovery of three spec bugs confirmed by the authors. The spec in this system consists of a single safety property, *Blockchain Consistency*, and the environment, which includes the high-level distributed system, the network, and the adversary which are all modeled as a state machine. Upon manual inspection of these STPs, we found that the adversary spec was *incomplete*, based on our understanding of what the adversary spec should be. The overall proof still passed even in the presence of these three bugs because they essentially cancel each other out; two making the spec weaker than it should be, and the other making the spec stronger.

The first bug identified in the adversary spec was an example of the spec being too strong; limiting the actions of what an adversarial node *should* be able to do. The initial hint indicating the possibility of this case was provided by failing Provability STPs. The initial reason for writing Provability STPs was to answer if the adversary spec was too strong; which is answered by the general form of Provability STPs. An overly restricted adversary model would weaken and perhaps invalidate the guarantees of the overall proof. Following the guidelines for writing STPs in Section 4.2.2, negating the conclusion of the failing Provability STPs led to the discovery of a passing Counterexample STP. Lemma 8 is a simplified example of such a failing Provability STP.

The failing simplified STP in Lemma 8 hints at the fact that the ability of the adversary to extract signed data structures is unnecessarily restricted. In the system model for QBFT, and other Byzantine fault-tolerant consensus protocols, a Byzantine node should be allowed to behave arbitrarily while not violating cryptographic assumptions. In this QBFT spec, adversaries are only able to extract and forward `CommitSeals(CS)` from a subset of received message types. The STP in Lemma 8 specifies the behavior of an adversary node receiving a `Proposal` message signed with a quorum of `CS1`, and constructing and sending a *NewBlock* message containing the `block` and `CS1` data structures copied from the `Proposal` message. The postcondition for this STP stipulates that this scenario constitutes a valid state transition from state `a` to state `a'`. After observing that this STP failed to immediately

Lemma 9 Simplified QBFT Adversary Correctness STP

```
lemma AdversaryForgeMessageSTP(
    a:Adversary,
    a':Adversary,
    inMsgs: set<Message>,
    outMsgs: set<Message>)
  requires validAdversaryConfig(a,a',inMsgs)
  requires outMsgs == {ProposalMsg(CS2)}
                        // forged msg
  ensures !AdversaryNext(a,a',inMsgs,outMsgs)
```

verify, negating the conclusion to, `!AdversaryNext(a, a', inMessages, outMessages)`, resulted in the proof passing for this transformed counterexample STP.

While investigating the implications of the behavior in the failing STP, we modified the adversary spec, weakening it to allow an adversary to forward `CS1` regardless of what message first contained it. After making this change, the full system's safety proof failed. To differentiate the proof failure from a now incomplete lemma, we constructed and proved a concrete counterexample resulting in a violation of the system's safety property, confirmed by the authors.

The second bug in the adversary spec is an example of the spec being too weak. This weakness is the reason why we can show a concrete counterexample to safety after addressing the first spec bug. The spec allows an adversarial node to send a `Proposal` message containing a `block` data structure with arbitrary values, including using the `CommitSeals` of honest nodes even if the adversary had not previously received such `CommitSeals` in previously received messages. `CommitSeals` are only used to make a final decision of committing a block, but this weakness in the spec allows an adversary to propose new blocks containing `CommitSeals` as if from honest nodes. This behavior allows an adversary to send a message that appears to be signed by an honest node, violating the security assumptions made by the QBFT system model. The STP in Lemma 9 is a simplified version of the Correctness STP used to discover this spec bug.

The third bug identified is related to the previous bug and is concerned with the underspecified spec of the function *getNewBlock()*. This function is empty-bodied and only contains the spec. Due to the underspecification of this function, a caller of this function, including an honest node can immediately send a message, such as a `Proposal` message, containing a full quorum of commit seals. If a buggy implementation is provided for this function, it too, could lead to a violation of the safety property.

5.2.3 STP Discussion

STPs enable fine-grain testing of specs and have been effective at helping to identify all ten spec bugs in the six open-source verified systems. By leveraging the insight that writing proofs for specific values is easier than a general proof, the manual effort required to write STPs remains minimal.

In the QBFT spec the presence of three spec bugs, two manifesting as a weakness in the spec and the other counteracting the first by overly restricting the adversary, makes manually or automatically identifying these bugs extremely difficult. Following the STP testing methodology, we efficiently, and without being experts in the system, identified these disconnects between what was written in the spec and our understanding of the intent of the spec.

5.3 Mutation Testing Evaluation

We applied IronSpec's automatic mutation testing to a set of six in-house specs, the two spec examples from Abreu et al. [6], and the spec of six open-source verified codebases. The evaluation attempts to answer how prevalent *alive* mutations are in specs, and how useful the provided hints are in assisting to identify spec bugs.

All mutation testing experiments were performed on a cluster of 21 servers where each node was equipped with two Intel E5-2660 v2 10-core CPUs at 2.20 GHz and with 256GB ECC Memory. In each experiment, one root node would create all mutations and send all subsequent verification requests in each stage of the mutation testing process to be processed in parallel at the other 20 nodes in the cluster using Dafny version 3.8.1. The results from running IronSpec can be found in Table 4 and are further explained in the following subsections.

5.3.1 In-House Specifications

In addition to the five in-house specs introduced in Section 5.2.1, we applied mutation testing on a simple key-value store state machine spec.

The top half of Table 4 contains the experimental results of running mutation testing on the in-house specs. Each buggy spec was tested with a correct implementation (C) and an incorrect implementation (I). Mutation testing all in-house specs with a correct spec and a correct implementation resulted in no alive mutations.

Mutation testing identified relevant alive mutations, regardless of whether the implementation is correct. For all six incorrect specs, mutation testing resulted in helpful alive mutations. The only exception is Sort (C), whose implementation contained additional loop invariants that caused the proof to fail when using weaker preconditions. In all other cases alive mutations were useful hints in manually identifying a weakness in the spec.

5.3.2 Alive Mutations in Open Source Systems

The Div and NthHarmonic specs are simple buggy specs introduced by Abreu et al. [6], where the authors proposed initial techniques to repair simple spec errors in Dafny. The alive mutations IronSpec found for these specs coincide with

Table 4: Results from running IronSpec's automatic mutation testing. In-House, buggy, specs marked with "(C)" correspond to experiments with a buggy spec but a correct implementation, whereas "(I)" indicates a buggy spec with an incorrect implementation. The Predicate Name is the specific mutation target within a spec. Spec LOC is the size of the mutation target, and Proof/Impl LOC is the size of the full end-to-end implementation and proof. Mutations are the total number of mutations generated, Alive Mutations indicate the number of alive mutations after all three passes and hierarchy classifications.

	Specification	**Predicate Name**	**Spec LOC**	**Proof/Impl LOC**	**# Mutations**	**# Alive Mutations**	**Time**
In-House Specs	**Max (C)**	maxSpec	2	5	80	1	11.3s
	Max (I)			7		4	7.5s
	Sort (C)	sortSpec	1	55	50	0	4.5s
	Sort (I)			4		1	7.3s
	Binary Search (C)	searchSpec	4	31	170	1	10.4s
	Binary Search (I)			18		2	24.3s
	KV SM (C)	Query Op	4	187	37	7	21s
	KV SM (I)					7	28.8s
	Token-wre (C)	GInv	1	87	13	1	7.8s
	Token-wre (I)			91		1	7.8s
	SimpleAuction-wre (C)	GInv	9	181	187	3	15.25s
	SimpleAuction-wre (I)					3	15.5s
Open-Source Specs	**Div**	Div	3	14	50	3	3.5s
	NthHarmonic	NthHarmonic	1	4	11	2	3s
	QBFT	NetworkInit	3	15071	44	3	80 min
	QBFT	AdversaryNext	48		197	7	162 min
	QBFT	AdversaryInit	3		35	4	80 min
	Distributed Validator	AdversaryNext	23	24747	110	7	191 min
	daisy-nfsd	GETATTR	4	18	35	1	4.3 min
	daisy-nfsd	WRITE	7	54	119	3	4.6 min

the conclusions made by Abreu et al. in demonstrating that these specs are buggy by being too weak.

QBFT Of the 44 generated mutants for the initial state of the network state machine spec, `NetworkInit`, three mutations remained *alive* as the roots of their respective components in the *mutation DAG*. Upon manual inspection of the surviving mutants, the spec differences all referenced an aspect of the Network's state that was never mentioned elsewhere. Thus, any value for part of the state would be considered "safe". These mutations do not imply the existence of a bug, but neither are they strictly false positives; rather they are examples of spec bloat. These alive mutations should still serve as flags to the developer, forcing them to answer the question of whether this state is needed, and if so why are these parts of the state not referenced?

The alive mutations for the `AdversaryNext` and `AdversaryInit` predicates, both parts of the adversary state machine spec can be considered false positives. The alive mutations were all *stronger* mutations, but it is always safe to restrict the actions allowed by an adversarial node. Some alive mutations implied that the proof would still pass with no adversaries in the system, or only taking trivial actions. This observation led us to question and then to test with STPs if the adversary spec was initially more restricted than it should be, leading to the bugs discussed in Section 5.2.2.

DVT The Distributed Validator Technology Protocol (DVT) spec and proof [4] captures the behavior of an Ethereum Validator, where a group of nodes coordinates to perform the Ethereum validator duties. The DVT spec consists of the desired *non-slashable attestation* property and the environment, with the latter defined as the high-level distributed system, an adversary, and the network. All aspects of the environment are modeled as state machine specs. The *non-slashable attestation* property ensures that the system avoids committing a slash-able offense and produces valid attestations.

Applying mutation testing to the `AdversaryNext` predicate in the adversary spec resulted in seven *alive* mutations. One of the mutations was a false positive. Three of the mutations hinted towards a limitation of the messages allowed to be sent by an adversary, leading to a similar discovery as in the first QBFT bug. The remaining three alive mutations were concerned with the creation of attestations. This weakness lies in the spec's lack of specificity regarding the attestations an adversary can create. Armed with this observation, we show with a counterexample that this weakness could lead to a safety violation.

daisy-nfsd Applying the mutation framework to daisy-nfsd's [15] top-level NFS API spec resulted in alive mutations in two different methods' specs, `GETATTR` and `WRITE`. These mutations hint at the same spec weakness that both methods contain; one that would allow for a different trivial implementation to always return an error. This bug was confirmed by the authors as a known issue in their spec.

5.3.3 Combining STPs With Mutation Testing Hints

An alive mutation is a compelling hint that the spec may be weaker than intended, but it is just a hint; writing STPs (Section 4.4) is always the final step in testing. Consider the DVT spec bug from Section 5.3.2. The original mutation target predicate is non-trivial and consists of 22 lines including multiple quantified conjuncts. Working backwards from the alive mutations and focusing only on the expression derived by the difference between the original spec and an alive mutation, resulted in shrinking the 22-line predicate into only a single conjunct. Writing STPs concerning this single conjunct is much more tractable than writing STPs for the entire predicate. The tradeoff of the slightly increased manual effort to calculate the simplified expression and writing STPs concerning it outweighs the effort needed to consider the entire spec.

5.3.4 Mutation Testing Discussion

Mutation testing supplied the hints that led to the discovery of spec bugs in two verified codebases. These results exemplify the usefulness of adding automation to search for disconnects between the implementation and the spec. The insight of identifying tangible differences as potential areas of disconnected intent is a beneficial hint that can be leveraged to identify spec bugs. The results in Table 4 demonstrate that even with a small set of mutations, we were successful in identifying spec weaknesses.

The large increase in execution time of running mutation testing taken between different specs can be attested to the varying sizes of the full system proof and the time that it takes to verify the entire proof with the mutations that survive the first two passes. For instance, even running the full end-to-end proof once of the unmodified QBFT system can take approximately an hour to complete. The cost of running IronSpec on a large verified system is worth the execution time to debug a potential spec bug.

The results of testing specs with mutation testing demonstrate the effectiveness of this approach, but we did find that not all alive mutations led to the discovery of spec bugs. While the possibility of discovering false positive alive mutations exists, all cases were quickly diagnosed. Of the 61 alive mutations identified across all tested specs, we consider 13 to be false positives, because the spec weaknesses they hint at were deemed intended. All mutations for QBFT's `AdversaryNext` and `AdversaryInit` were considered false positives. A single alive mutation in the set of alive mutations for both DVT `AdversaryNext` and daisy-nfsd `WRITE` were also characterized as false positives. The one false positive mutation found in DVT `AdversaryNext` was classified as such because it would have only allowed the adversary to make attestations already created, which would not have led to any unintended, incorrect behavior.

Verified methods that modify ghost state are at a higher risk of mutation testing producing false positives. Ghost state is only maintained for the sake of the proof, and often, underconstrained postconditions related to ghost state would not result in a buggy implementation. The false positive mutation in daisy-nfsd's `WRITE` method hinted towards underconstrained ghost state that was modified in the method's implementation. Nevertheless, the daisy-nfsd authors confirmed that a different implementation, which modified this ghost state differently, would not lead to a safety violation or break the proof. However, they did acknowledge that this weakness was not immediately apparent.

Rather than finding false positives, it is also possible, especially with larger systems, for no alive mutations to be identified. For QBFT, DVT, and daisy-nfsd there were other spec mutation targets we applied IronSpec to, which resulted in no alive mutations. For example, in both QBFT and DVT, the alive mutations identified were part of the trusted specified environment, whereas no alive mutations were found for their respective safety properties, *Blockchain Consistency* and *non-slashable attestation*.

The IronSpec prototype takes the first steps to bring automation and structure to testing specs. The prototype targets Dafny specs, but the conceptual techniques are not tied to Dafny.

5.4 Amount Of Manual Effort Required

In the same way testing traditional software systems requires developer effort, testing specifications does too. IronSpec provides a framework and automation aid to help developers in this endeavor. If one is willing to spend the effort to verify their system, it is worth spending a few additional hours to gain confidence in proving the intended property. While manual effort is unavoidable, this effort can be greatly reduced as the automation of IronSpec helps to focus the developer's attention on a few potentially problematic aspects of the spec.

The majority of manual effort we expended in applying IronSpec was spent on understanding each system well enough to interpret the hints from the automation of IronSpec and to write appropriate STPs. Even so, the amount of manual effort expended remained relatively low despite not having specific expertise in each system. For example, the specification bug identified in daisy-nfsd took approximately 1-2 hours to determine from first examining the code base and running IronSpec to confidently identifying the spec bug. When testing daisy-nfsd, it took minutes to run the mutation framework, and the rest of the time was spent comprehending the hint of the alive mutation, writing STPs, and understanding the underlying system well enough to determine if the flagged behavior was intended. In conversations with the daisy-nfsd authors, they admitted that this spec bug was subtle. Familiarity and expertise in a system and its spec will only help to further reduce the necessary manual effort.

Writing STPs for complex systems takes more effort than for simple examples. Yet, the limited scope of writing STPs with concrete values drastically limits the size of any potential additional proof annotations needed for those STPs in comparison to what would be necessary for proof of the unconstrained behavior. The effort of writing STPs varied per the complexity of the system. Writing a comprehensive suite of STPs ranged from a few hours to multiple days worth of effort for the larger QBFT and DVT specs. Writing-focused STPs based on alive mutations ranged from tens of minutes to hours per alive mutation.

6 Related Work

Kemmerer [33] first identified the potential benefits of testing specifications. Kemmerer proposed a technique based on symbolic execution to check if a spec satisfied the English-based functional requirements. Since then, several studies have proposed techniques to test informal user requirements [13, 18, 35, 42, 43].

The closest related previous work is the study by Fonseca et al. [22], which manually and painstakingly identified weaknesses in verified codebases, including two spec bugs. Other works have also begun to apply more structured approaches to increase reliability in formal methods. Kupfeman [36] discussed the possible advantages of vacuity and coverage checks for temporal-logic model-checking tools. Inspired by vacuity testing, and the concept of *unit proofs* from Chong et al. [16], Priya et al. [52] performed a case study of some AWS verified libraries, uncovering some hidden bugs. Bernardi et al. [11] also identified formal specifications as a weak point in the verification process, and proposed to reuse specifications once correct, for smart contracts. Le Traon et al. [38] even discussed the notion of applying a mutation analysis to Eiffel contracts.

With verification becoming more commonplace and the discovery of spec bugs in verified systems, a few, mostly manual efforts have attempted to identify spec bugs. The 2022 Notional Finance bug found in verified code inspired Certora to investigate ways to introduce testing into the verification process [49]. Recently, Abreu et al. [6] proposed initial efforts in using the dynamic invariant inference tool *Daikon* [20] to aid in automatically repairing specifications. When faced with a failing proof, their prototype assumes that the implementation is correct, and uses the implementation to generate test cases for the spec. Any failing tests present an opportunity to attempt to fix the spec by suggesting strengthening or weakening modifications. Of course, this approach only works if the implementation is correct, which partially defeats the purpose of performing verification in the first place.

Testing and formal methods share a close relationship and a common goal. Often, rather than questioning specs, developers have relied on specs or other formal methods to assist in testing traditional software [11, 16, 27, 39]. Works concerned with the Oracle problem [9, 12] have often utilized specs thus. There has even been work to test verification tools [30].

7 Conclusion

The correctness of specifications is the rock upon which the entire edifice of formal verification is built. As formal verification becomes increasingly popular, it is imperative that the foundation be as solid as possible.

This work proposes IronSpec, a systematic framework of manual and automated approaches to aid developers in finding bugs in their specs. We show how IronSpec was used to identify a number of subtle bugs in the specs of open-source codebases, without requiring copious amounts of expertise on the proven system. We believe IronSpec is a necessary step forward towards writing correct software.

Acknowledgements

We want to thank Shuangyu Lei for her work on an early version of the mutation testing framework. We thank the anonymous OSDI reviewers and our shepherd, Baptiste Lepers, for their insightful and useful feedback that we used to improve this paper. This work was supported by National Science Foundation grants CCF-2118512 and CCF-2018915.

References

[1] Eth2.0-dafny. https://github.com/Consensys/eth2.0-dafny/tree/master, 2021.

[2] Qbft formal specification and verification. https://github.com/Consensys/qbft-formal-spec-and-verification, 2021.

[3] Aws encryption sdk for dafny. https://github.com/aws/aws-encryption-sdk-dafny, 2023.

[4] Formal verification of the distributed validator technology protocol. https://github.com/Consensys/distributed-validator-formal-specs-and-verification, 2023.

[5] Ironspec. https://github.com/GLaDOS-Michigan/IronSpec, 2024.

[6] A. Abreu, N. Macedo, and A. Mendes. Exploring automatic specification repair in dafny programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 105–112. IEEE, 2023.

[7] C.-C. Andrici and Ş. Ciobâcă. Verifying the dpll algorithm in dafny. *arXiv preprint arXiv:1909.01743*, 2019.

[8] D. Baldwin and F. Sayward. *Heuristics for determining equivalence of program mutations*. Yale University, Department of Computer Science, 1979.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[10] K. Beck. *Test driven development: By example*. Addison-Wesley Professional, 2022.

[11] T. Bernardi, N. Dor, A. Fedotov, S. Grossman, N. Immerman, D. Jackson, A. Nutz, L. Oppenheim, O. Pistiner, N. Rinetzky, et al. Wip: Finding bugs automatically in smart contracts with parameterized invariants. *Retrieved July*, 14:2020, 2020.

[12] M. Böhme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[13] M. Brockmeyer. Using modechart modules for testing formal specifications. In *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 20–26. IEEE, 1999.

[14] F. Cassez, J. Fuller, and H. M. A. Quiles. Deductive verification of smart contracts with dafny. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 50–66. Springer, 2022.

[15] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the {DaisyNFS} concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, 2022.

[16] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 11–20, 2020.

[17] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[18] G. De Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2010.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[20] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[21] R. W. Floyd. Assigning meanings to programs. *Program Verification: Fundamental Issues in Computer Science*, pages 65–81, 1993.

[22] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 328–343, 2017.

[23] R. Gheyi, M. Ribeiro, B. Souza, M. Guimarães, L. Fernandes, M. d'Amorim, V. Alves, L. Teixeira, and B. Fonseca. Identifying method-level mutation subsumption relations using z3. *Information and Software Technology*, 132:106496, 2021.

[24] P. Hamill. *Unit test frameworks: tools for high-quality software development.* " O'Reilly Media, Inc.", 2004.

[25] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 99–115, 2020.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[27] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[29] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification i: A logical basis and its implementation. *Acta Informatica*, 4(2):145–182, 1975.

[30] A. Irfan, S. Porncharoenwase, Z. Rakamarić, N. Rungta, and E. Torlak. Testing dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 556–567, 2022.

[31] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[32] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, 2016.

[33] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE transactions on software engineering*, (1):32–43, 1985.

[34] U. Kirstein. Post-mortem analysis of the notional finance vulnerability — a tautological invariant, January 2022. [Online; posted 17-January-2022].

[35] A. Knüppel, L. Schaer, and I. Schaefer. How much specification is enough? mutation analysis for software contracts. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 42–53. IEEE, 2021.

[36] O. Kupferman. Sanity checks in formal verification. In *International Conference on Concurrency Theory*, pages 37–51. Springer, 2006.

[37] L. Lamport. Specifying systems: the tla+ language and tools for hardware and software engineers. 2002.

[38] Y. Le Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006.

[39] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 602–613, 2016.

[40] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. *ACM SIGPLAN Notices*, 51(1):357–370, 2016.

[42] M. Li and S. Liu. Reviewing formal specification for validation using animation and trace links. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 263–270. IEEE, 2014.

[43] S. Liu, J. A. McDermid, and Y. Chen. A rigorous method for inspection of model-based formal specifications. *IEEE Transactions on Reliability*, 59(4):667–684, 2010.

[44] Y.-S. Ma and J. Offutt. Description of mujava's method-level mutation operators. *Update*, 2016.

[45] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[46] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2013.

[47] H. Moniz. The istanbul bft consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.

[48] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[49] S. Phipathananunth. Using mutations to analyze formal specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 81–83, 2022.

[50] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[51] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121. IEEE, 1976.

[52] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel. Verifying verified code. *Innovations in Systems and Software Engineering*, 18(3):335–346, 2022.

[53] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Identifying On-/Off-CPU Bottlenecks Together with Blocked Samples

Minwoo Ahn[1], Jeongmin Han[1], Youngjin Kwon[2], Jinkyu Jeong[3]
[1]*Sungkyunkwan University,* [2]*KAIST,* [3]*Yonsei University*
{minwoo.ahn, jeongmin.han}@csi.skku.edu, yjkwon@kaist.ac.kr, jinkyu@yonsei.ac.kr

Abstract

The rapid advancement of computer system components has necessitated a comprehensive profiling approach for both on-CPU and off-CPU events simultaneously. However, the conventional approach lacks profiling both on- and off-CPU events, so they fall short of accurately assessing the overhead of each bottleneck in modern applications.

In this paper, we propose a sampling-based profiling technique called *blocked samples* that is designed to capture all types of off-CPU events, such as I/O waiting, blocking synchronization, and waiting in CPU runqueue. Using the blocked samples technique, this paper proposes two profilers, *bperf* and *BCOZ*. Leveraging blocked samples, bperf profiles applications by providing symbol-level profile information when a thread is either on the CPU or off the CPU, awaiting scheduling or I/O requests. Using the information, BCOZ performs causality analysis of collected on- and off-CPU events to precisely identify performance bottlenecks and the potential impact of optimizations. The profiling capability of BCOZ is verified using real applications. From our profiling results followed by actual optimization, BCOZ identifies bottlenecks with off-CPU events precisely, and their optimization results are aligned with the predicted performance improvement by BCOZ's causality analysis.

1 Introduction

Application profiling encompasses the analysis of two types of events: on-CPU events and off-CPU events. Profiling on-CPU events aims to analyze instructions executed on a CPU [1, 4, 7, 15, 17, 19–21, 33, 45, 53, 54]. In contrast, profiling off-CPU events aims to analyze waiting events within an application such as waiting for blocking I/O completion, locks, scheduling, etc [27, 35, 38, 39, 55, 58].

In the past, applications were clearly characterized as either CPU-boud or I/O-bound due to the use of slow I/O devices such as HDD or SATA SSD. Therefore, existing profiling tools have applied bottleneck analysis techniques separately for on-CPU events or off-CPU events. However, with the recent advancements in fast storage and many-core CPUs, modern applications often exhibit complex behaviors. Especially, their performance bottleneck is combined by on-CPU events and off-CPU events, necessitating the need for comprehensive profiling of both on-CPU and off-CPU events *simultaneously* and capturing their interactions. For example, with the emergence of NVMe SSDs and ultra-low latency SSDs, the critical path of I/O-intensive applications often shifts from I/O to CPU events [23, 30–32]. Consequently, studies have focused on optimizing on-CPU events rather than I/O events to enhance the performance of I/O-intensive applications [23, 30–32].

However, existing profilers focus on analyzing only on-CPU [17, 20, 33] or off-CPU events [3, 38], so they cannot analyze the complicated behaviors of modern applications. COZ [15], a state-of-the-art causal profiler, estimates performance gain through its virtual speedup approach. COZ intentionally delays competing threads to estimate the performance impact by optimizing certain code lines without actually optimizing them. However, COZ applies virtual speedup profiling exclusively to on-CPU events as it lacks the capability to incorporate execution information from off-CPU events. wPerf [58], a state-of-the-art off-CPU analysis profiler, traces waiting events between threads during the execution and reports the result in the form of a graph (called a wait-for graph). While wPerf can analyze interactions between on-CPU and off-CPU events, it falls short in assessing the actual impact of bottlenecks on application performance. Furthermore, although wPerf identifies off-CPU bottlenecks, it lacks detailed information about the application contexts related to these bottlenecks. These limitations require programmers to attempt optimizations for various bottlenecks to achieve actual performance improvements and demand additional effort to pinpoint the application code to be optimized (Section 2.2).

This paper introduces a new profiling technique called *blocked samples*, which is designed to capture off-CPU events. Drawing inspiration from the event-based sampling (e.g., Linux perf subsystem [17]), our approach employs sampling-based profiling of off-CPU events. Similar to Linux perf, the blocked samples technique periodically captures snapshots of events with designated information, such as the instruction pointer (IP) and the call stack of each thread. However, unlike Linux perf, which disables profiling while the target thread is blocked, the blocked samples technique overcomes this limitation by allowing sampling of off-CPU events while the target thread is blocked. Leveraging the information provided by blocked samples, we aim to devise profiling techniques that consider both on-CPU and off-CPU events simultaneously, identify bottlenecks and estimate their performance impact if optimized.

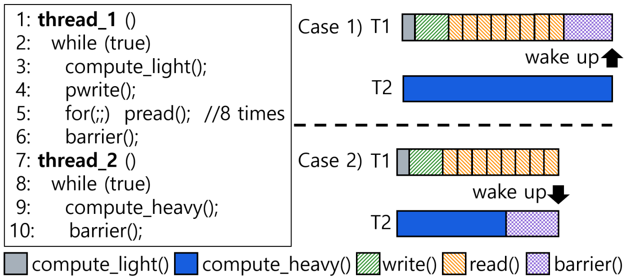
To demonstrate our idea, we devise two profilers, *bperf* and *BCOZ*. bperf is an easy-to-use sampling-based profiler, following the interfaces of the familiar Linux perf tool. bperf incorporates the blocked samples technique. By considering the sampling results that include blocked samples, bperf accurately calculates the overhead of each event. This enables bperf to provide precise overhead information for both on-CPU and off-CPU events. Additionally, bperf reports detailed information about the sampled off-CPU events such as call stacks, kernel stacks, and blocking types (e.g., I/O, synchronization, scheduling, etc.), assisting users in gaining a deeper understanding of overheads.

BCOZ is a causal profiler that leverages the concept of virtual speedup [15] for off-CPU events obtained by blocked samples. BCOZ provides estimated performance improvement not only of a single off-CPU event but also of a function involving multiple on-/off-CPU events. Additionally, BCOZ supports a per-subclass virtual speedup technique, which estimates the potential speedup if off-CPU events of a particular type are optimized, such as using faster I/O devices or eliminating CPU scheduling delays. This feature allows users to consider various optimization alternatives, such as upgrading I/O devices or assigning additional CPU cores.

Our evaluation results demonstrate that BCOZ successfully identifies bottlenecks of real applications including both on- and off-CPU events. Specifically, BCOZ identifies various bottlenecks of the RocksDB key-value store [50] with read-intensive and write-intensive workloads. We observe that with various memory configurations and workload patterns, diverse parts of the program exhibit distinct performance bottlenecks. BCOZ precisely identifies such bottlenecks and provides an estimated speedup of the optimization of each bottleneck. We verify that the reported virtual speedup results are also aligned with the actual speedup when various optimization techniques are applied. These results prove the effectiveness of BCOZ by profiling on- and off-CPU events simultaneously.

Our contributions are summarized as follows:

- We demonstrate the need for integrated profiling of both on-CPU and off-CPU events to overcome the limitations of conventional profilers (Section 2).
- We propose a new sampling technique called blocked samples, designed for capturing off-CPU events. By incorporating blocked samples with on-CPU samples, we enable the identification of application bottlenecks related to both on- and off-CPU events (Section 3.1).
- We introduce two profilers, bperf and BCOZ, that utilize the blocked samples technique. These profilers provide insights into the overhead of off-CPU events and uncover potential performance improvement opportunities that were previously unrecognized (Section 3.2 and 3.3).
- We present practical use cases of our profilers, bperf and BCOZ. Through profiling, we identify off-CPU bottlenecks in applications. Then, we validate the identified bottlenecks through simple optimizations or comparison with previous optimization studies (Section 4).



```
thread_1 ()
  while (true)
    compute_light();
    pwrite();
    for(;;) pread(); //8 times
    barrier();
thread_2 ()
  while (true)
    compute_heavy();
    barrier();
```

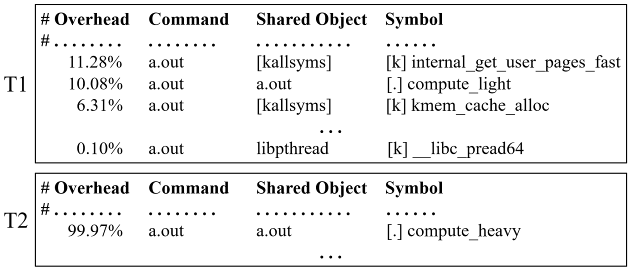
Figure 1: Motivational example of mixed on-/off-CPU events.

2 Background

2.1 Sampling-based Profiling

Sampling-based profiling (e.g., task-clock in Linux perf [17]) is a widely used and efficient method for profiling the application execution with low overhead [17, 20, 33]. It periodically captures (or samples) the execution information of programs on the CPU such as the instruction pointer (IP) and the callchain of each thread (or CPU). With the sampling results, profilers can report statistical overhead [17, 33], perform causal analysis [15], and visualize callchains of captured events [21]. However, identifying the exact bottleneck in sampling-based profiling remains challenging due to the absence of off-CPU events, such as I/O waiting, synchronization, and CPU scheduling.

We illustrate the limitations of profiling without off-CPU events using a simple example program. Figure 1 presents an example of an application involving both on-/off-CPU events. In this example, two threads are executed concurrently and are synchronized through a `barrier` that is implemented using a mutex and condition variable. *Thread 1* executes `compute_light`, which is a small on-CPU computation (iterative integer increment), one 4-KB disk write (`pwrite`), and eight 512-byte disk reads (`pread`). The two types of off-CPU events, hence disk write and reads, are the majority of the execution of Thread 1. *Thread 2* executes `compute_heavy` only, which performs a large on-CPU computation. We adjust the computation load of Thread 2 to make two distinct cases: *Case 1* where Thread 2 is on the critical path and *Case 2* where Thread 1 is on the critical path, as shown in the right



```
     # Overhead  Command   Shared Object  Symbol
     #........   ........  ...........    ......
        11.28%   a.out     [kallsyms]     [k] internal_get_user_pages_fast
T1      10.08%   a.out     a.out          [.] compute_light
         6.31%   a.out     [kallsyms]     [k] kmem_cache_alloc
                                ...
         0.10%   a.out     libpthread     [k] __libc_pread64

     # Overhead  Command   Shared Object  Symbol
T2   #........   ........  ...........    ......
        99.97%   a.out     a.out          [.] compute_heavy
                                ...
```

Figure 2: Sampling-based profiling result of Case 2.

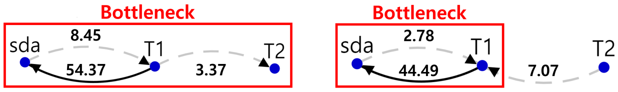
part of the figure. In Case 1, the bottleneck is `compute_heavy`. In Case 2, the bottlenecks are `compute_light` and the I/O events, and their optimization leads to performance improvement. However, in both cases, the optimization of one thread does not improve the performance indefinitely due to the barrier synchronization between the two threads; the critical path changes from one to the other. Hence each optimization has limited global impact on the performance [15, 58].

Limitation of On-CPU Analysis with Sampling. Consider Case 2 where the actual bottleneck is the off-CPU events (i.e., `pread`/`pwrite`), the on-CPU analysis only cannot identify the correct bottleneck. Figure 2 shows the profiling results of Case 2 with the Linux perf tool [17], a popular on-CPU sampling-based profiler. In the figure, each line represents the overhead portion, command name, shared object name, and event symbol; `[.]` and `[k]` indicate user-level and kernel-level symbols, respectively. Due to the tool's inability to capture off-CPU events caused by the blocking I/O events, it only reports the overhead of on-CPU events such as `compute_heavy`, `compute_light` and Linux kernel internal events. The C library function related to `pread` (i.e., `__libc_pread64`) is captured, but it includes only the on-CPU parts, indicating only 0.10% of the overhead. Consequently, the user may examine the results shown in the figure, identify the `compute_heavy` event as the significant overhead among on-CPU events, and focus on optimizing it even though the computation of Thread 2 is not on the critical path. This limitation necessitates off-CPU analysis and causality analysis to precisely identify the performance bottleneck.

2.2 Off-CPU Analysis

Basic Utilities. Basic utilities, such as `top` or `iostat`, can provide information of the overall I/O usage. However, these utilities are primitive and ineffective for detailed analysis of I/O events and correlating those to application contexts, such as IP and callchain.

Off-CPU Tools. Various off-CPU profiling tools [27, 35, 38, 39, 55, 58] exist to support profiling of off-CPU events. However, these profilers are limited in terms of (1) focusing on a specific type of off-CPU events (e.g., syncperf [35]) or (2) providing unsorted information (e.g., off-CPU time distribution [14], call-chains of off-CPU events [40]), which requires a programmer's additional efforts to identify important bottlenecks and their impact on the program performance. For example, off-CPU flamegraph [39] provides and visualizes the overhead of off-CPU events and their callchains. However, the tool does not provide the performance impact of each off-CPU event; an off-CPU event with the largest overhead does not necessarily mean its optimization results in performance improvement to the same extent [15, 58].



(a) Case 1 in Figure 1.　　(b) Case 2 in Figure 1.

Figure 3: *wait-for* graph results of Figure 1. For Case 1, the knot includes both I/O and synchronization (`barrier`), while the knot in Case 2 includes only I/O.

wPerf. wPerf [58] is a state-of-the-art profiler that traces all types of waiting events including off-CPU events and reports their relationship in the form of a wait-for graph. Moreover, wPerf identifies bottlenecks by analyzing the global impact of waiting events on other threads. Hence, wPerf reports the performance bottleneck by identifying waiting events for which all the worker threads are waiting to progress; these waiting events are called a knot. Each knot contains a bottleneck.

However, we observe two important limitations when identifying bottlenecks using wPerf. Let us explain the limitations using the example program in Figure 1.

Profiling Result. Figure 3 shows the profiling result of Case 1 and Case 2 in Figure 1 using wPerf. In this figure, the vertices labeled *T1* and *T2* correspond to Thread 1 and Thread 2, respectively, and *sda* represents the disk, hence an I/O device. `sda` ← T1 indicates that Thread 1 is waiting for completion of disk I/O requests (i.e., `pread` and `pwrite`), and `sda` --→ T1 represents the I/O device is waiting for I/O requests from Thread 1 (i.e., `compute_light`) [58]. Moreover, T1 --→ T2 (Case 1), and T1 ←-- T2 (Case 2) indicate the waiting period caused by the synchronization (`barrier`). The numbers on the edges are the global impact of each edge reported by the wPerf which can be interpreted as waiting time.

Firstly, wPerf does not precisely identify bottlenecks and their actual impact on the program performance. In Figure 3a, the red box denotes the knot of Case 1. Hence, Thread 1/2 and `sda` are the bottlenecks. In addition, `sda`, hence the disk, has the largest global impact; the edge has the weight of 54.37 which is an order of magnitude larger than the weight of the other edges. Hence, the profiling result mislead to optimizing I/O events associated with `sda`. However, as shown in Case 1 in Figure 1, optimizing `pread` or `pwrite` does not improve the performance since the real bottleneck is Thread 2 (i.e., `compute_heavy`). Consequently, wPerf's bottleneck identification can be imprecise and can yield a waste of ineffective optimization efforts.

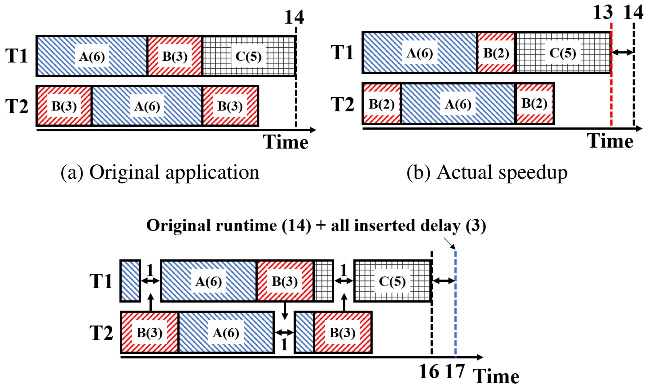
Secondly, wPerf can identify bottlenecks approximately but detailed information or context can be missing, which necessitates programmers' additional efforts to figure out bottleneck points in program codes. In Figure 3b, the knot includes T1 and `sda`. Both are in Thread 1, which are the correct bottlenecks of Case 2 in Figure 1. However, Thread 1 runs three different events, `compute_light`, `pread` and `pwrite`, each of which is on the critical path. As shown in Figure 1, the eight disk reads have the most significant impact on the program performance in Case 2. However, the profiling result of wPerf demonstrates only `sda` has the biggest impact and blurs the precise bottleneck identification among the two off-CPU events. For instance, the programmer may try to optimize `pwrite` but may fail to improve the write performance while the code `compute_light` has a latent performance improvement since the code is implemented inefficiently. Consequently, wPerf's analysis result is less informative and requires additional efforts before targeting events to optimize.

2.3 Causal Profiling

Optimization of bottlenecks identified through conventional profiling does not always guarantee performance improvement [15, 58]. This is especially true for multi-threaded applications since events with significant overhead may not be on the critical path (e.g., `compute_heavy` in Case 2 of Figure 1). Causal profiling [15] is a profiling methodology that estimates the actual impact of optimization on the program performance.

Virtual Speedup. Virtual speedup is a core technique for performing the causality analysis. It offers an estimation of potential performance improvement by virtually speeding up a particular event (e.g., a program code line) [1, 15, 26, 45, 46]. The virtual speedup technique does not require the actual optimization of a particular code line but can assess the causality of the performance optimization. This can be achieved by delaying concurrently running events (or threads). Hence, if a particular event is sped up by a certain amount, it can be simulated by delaying other concurrent events by that amount but not the target event itself.

Figure 4 illustrates an example of the virtual speedup technique by showing the timeline of a two-threaded application. Figure 4a shows the original timeline and Figure 4b shows the actual speedup case when the execution time of function B is optimized from 3 to 2, and as a result the total execution time is reduced by 1. Figure 4c shows the virtual speedup case. The virtual speedup technique speeds up a particular function (B in this case) virtually by delaying co-running threads on every invocation of the target function. This has an effect of which the target function is sped up since other than the function is delayed by the same amount. Hence, in this example, whenever B is called, the other thread is injected a delay of the amount to speed up. The virtual speedup technique measures the speedup by identifying the difference between the actual runtime, 16 in this case, and the expected execution time if function B is not sped up. The expected execution time without speedup is 17, which is obtained by that whenever B is invoked, all the threads are delayed by 1. Hence, the application's virtual speedup is 1 (17 minus 16).



(a) Original application

(b) Actual speedup

(c) Virtual speedup

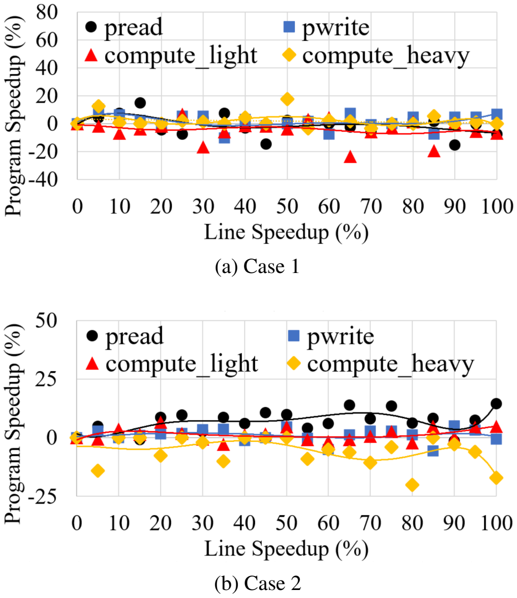
Figure 4: Illustration of virtual speedup when speeding up B. X(n) means function X runs for n time units.

COZ. COZ [15] is an implementation of the causal profiler employing the virtual speedup method. It reports to a user with information about the application's bottlenecks and provides an estimation of performance improvement for each optimization point. To apply the virtual speedup technique, COZ employs sampling-based profiling using the Linux perf subsystem. Periodically, COZ reads sampling results, IPs and callchains of each thread. Then, when the target code line to speed up is being executed, COZ applies the virtual speedup technique by delaying the execution of other threads, hence forcing sleep of co-running threads.

COZ carefully handles dependencies between threads and injects delays between threads with dependency in order to avoid incorrect estimation of virtual speedup. For example, if thread A is woken up by thread B, any injected delays to thread B while thread A is sleeping are considered the injected delays to thread A as well. This is because thread A is woken up after thread B has consumed its injected delays. In other words, if thread A is delayed by the same amount of injected delay after its wakeup, thread A experiences double delays from one source of delay injection. Accordingly, COZ manages dependencies arising from thread synchronization primitives (e.g., mutex, condition variable), and exempts delays during thread wakeups.

COZ employs two optimization methods to enhance its profiling performance. First, COZ processes multiple samples in batches. Second, it tries to skip consuming delays whenever all the threads need to consume the same amount of delay. This helps reduce the profiling time since otherwise, all the injected delays may increase the application's runtime significantly.

Profiling Result. While causal profiling is effective and informative in specifying bottlenecks and providing the estimated



(a) Case 1

(b) Case 2

Figure 5: Virtual speedup result of Figure 1.

outcome of bottleneck optimization, it lacks the capability of considering off-CPU events for profiling, often leading to incorrect profiling results. Figure 5 shows the profiling result of the example program in Figure 1 using COZ. In this figure, the x-axis represents the reduction in the execution time of a particular line, while the y-axis indicates the predicted overall runtime reduction of the program if that specific line speeds up by x%. For instance, a 0% program speedup indicates no performance improvement, whereas a 75% program speedup implies the runtime is reduced by 75%.

In Case 1, the actual bottleneck is `compute_heavy` but COZ fails to identify its potential performance improvement if it is optimized. Actually, COZ identifies marginal virtual speedup of all the four events in Case 1 as shown in Figure 5a. A similar phenomenon happens in Case 2 of the same example program as shown in Figure 5b. These results stem from the fact that COZ does not consider off-CPU events.

The causal profiling is an essential technique considering the ever-increasing complexity of applications on modern computer systems. However, the inability to profile off-CPU events is a critical limitation of existing causal profiling. The CPU performance improvement has stopped due to the end of Moore's law and Dennard scaling, and domain-specific accelerators, such as GPUs, FPGAs, and Smart SSDs, are gaining significant attention [25]. These heterogeneous computing environments make the behavior of applications more complicated with various off-CPU events for offloading computation to such accelerators. In addition, various low-latency I/O devices, such as CXL memory expander [16], RDMA-capable several-hundred gigabit network interface cards [49], flash-based or persistent memory-based solid-state drives [41, 59], are making application behavior increasingly complex. Traditional I/O-boundness or CPU-boundness is no longer a proper application classifier. The mixture of both I/O- and CPU-bound applications require sophisticated performance profiling methodology. Among these, profiling on-CPU and off-CPU events together is especially important for optimizing application performance.

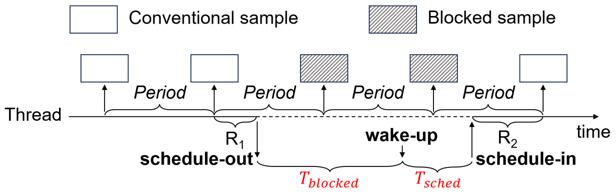


Figure 6: Blocked samples and conventional samples.

3 Design and Implementation

In this section, we present our methodology to profile on-CPU and off-CPU events simultaneously. Our methodology begins with proposing a new method of sampling off-CPU events, called *blocked samples* (Section 3.1). Then, we present two profilers for accommodating blocked samples: *bperf*, an easy-to-use sampling-based profiler (Section 3.2) and *BCOZ*, a causal profiler that profiles both on- and off-CPU events simultaneously and estimates potential speedup of optimizations (Section 3.3).

3.1 Blocked Samples

Blocked samples captures information of blocking events of threads, such as waiting for I/O completion, waiting for synchronization (e.g., mutex, condition variable), waiting for CPU scheduling, etc. The conventional on-CPU event sampling is thread-oriented (e.g., *task-clock* of the Linux perf subsystem [17]). When a thread executes its instructions on the CPU, the event-based sampling periodically collects a sample, hence snapshot, of thread context (e.g., IP and callchain) as shown in Figure 6. When a thread is blocked, the sampling is paused until the thread is woken up and resumes its execution. Different from the conventional on-CPU sampling, blocked samples augments the missing samples while threads are blocked, providing the execution context of blocking periods (shaded boxes in Figure 6).

Each blocked sample contains four attributes for tracking an off-CPU event: *IP*, *callchain*, *weight*, and *type*. The IP is the address of the last instruction before a thread is blocked. This is actually the return address of invoking a CPU scheduler (e.g., `schedule` or `io_schedule` in Linux). The callchain is the call stack of functions from the main function of a thread to the current instruction before being blocked.

The length of a blocking event can be varied (e.g., a few microseconds for CPU time-sharing or hundreds of milliseconds for disk I/O). Hence, one blocking event can contain multiple blocked samples of the same properties (i.e., identical IP, callchain, etc.). We encode the number of repeats to the *weight* field, saving space and time when handling blocked samples.

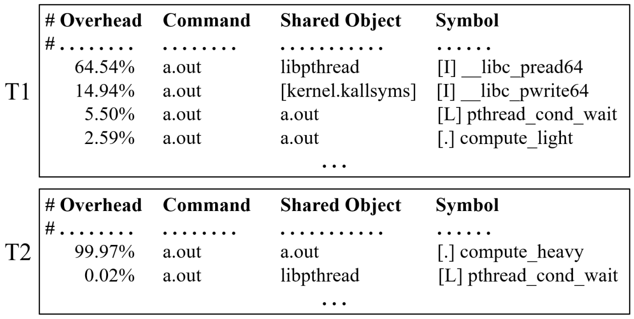
The *type* field is used to categorize blocked samples into the following subclasses:

- **I/O:** The I/O subclass corresponds to an off-CPU event that occurs when a thread requests and waits for an I/O event. Specifically, when a current thread submits a synchronous I/ O request and is blocked by invoking a CPU scheduler (e.g., a `task_struct` with `in_iowait` set in Linux), this blocking event is categorized as the I/O subclass.
- **Synchronization:** The synchronization subclass refers to an off-CPU event where a thread is waiting for a lock or condition variable. To identify synchronization subclasses, we slightly modified a process control block (e.g., `task_struct` in Linux) to include a field (`in_lockwait`) that identifies lock waiting. The field is set/cleared when a thread sleeps/wakes using the kernel-supported synchronization primitives (e.g., `futex` of Linux).
- **Scheduling:** The scheduling subclass refers to the off-CPU event that occurs when a thread is runnable but not scheduled on a CPU.
- **Others:** This subclass refers to off-CPU events that does not correspond to the above subclasses. Typically, off-CPU events related to sleeping (e.g., `usleep`) are in this category.

Implementation. Blocked samples are collected by extending the *task-clock* event in the Linux kernel's perf subsystem. The original sampling using task-clock collects samples by using a periodic timer (e.g., high-resolution timer). While a task is running on the CPU, the timer allows periodic sample collection. Meanwhile, when a task is off-CPU, hence blocked, the timer is paused until the task resumes execution.

Blocked samples augment the original task-clock-based sampling by including the blocking state of a thread. As the blocking state of a thread remains unchanged until the thread resumes its execution, our scheme captures the blocking state at the moment a thread resumes, rather than relying on the periodic timer. This is achieved by incorporating three task scheduling-related operations: *schedule-out*, *wake-up* and *schedule-in* functions.

First, when a thread is *scheduled out*, our scheme records the subclass of the thread's blocking, along with a timestamp marking the start of the off-CPU interval. Then, when the thread is *woken up*, our scheme records a timestamp to mark the end of the off-CPU interval. When the thread is eventually *scheduled in*, we record a timestamp and calculate the blocking interval ($T_{blocked}$) and the waiting interval for CPU scheduling (T_{sched}) as shown in Figure 6. For threads that are runnable but remain in the runqueue due to CPU contention, the off-CPU interval has no wake-up timestamp and hence belongs to the scheduling subclass of blocked samples. Finally, in the schedule-in function, if the blocking interval overlaps with one or more sampling points in time, a new sample is created. This sample contains the IP, callchain, weight and type attributes. For example in Figure 6, the two off-CPU events, $T_{blocked}$ and T_{sched} contain two sampling points. Hence, two blocked samples, one for blocking and the other for scheduling, are collected. If an off-CPU interval does not overlap with any sampling points, no blocked samples are collected. This approach minimizes the overhead of the collection of blocked samples, even with frequent off-CPU events, as the three hook points only perform timestamping.



```
    # Overhead  Command  Shared Object      Symbol
    # ........  ........ ...........        ......
       64.54%   a.out    libpthread         [I] __libc_pread64
T1     14.94%   a.out    [kernel.kallsyms]  [I] __libc_pwrite64
        5.50%   a.out    a.out              [L] pthread_cond_wait
        2.59%   a.out    a.out              [.] compute_light
                             ...

    # Overhead  Command  Shared Object      Symbol
    # ........  ........ ...........        ......
T2     99.97%   a.out    a.out              [.] compute_heavy
        0.02%   a.out    libpthread         [L] pthread_cond_wait
                             ...
```

Figure 7: bperf sampling results of Case 1 in Figure 1.

A single blocking event can encompass multiple sampling points. This means that the blocking event can generate multiple blocked samples. However, since these samples share identical attributes, our scheme avoids replicating them. Instead, it encodes the repetition using the *weight* field of a blocked sample. This approach reduces both the space and time overhead associated with handling blocked samples.

3.2 bperf

bperf is an online profiling tool that profiles applications using sampling-based profiling and provides statistics of sampling results. *bperf* is an extension of the Linux *perf* tool [17] to support blocked samples. Similar to perf, bperf can be an online or offline tool as its sampling-based profiling can be attached and detached at any time while a program is running and its profiling overhead is generally low (1.6%), as demonstrated in Section 4.4.

Basically, treating blocked samples has no significant difference from handling conventional on-CPU samples. Samples are classified using their IP and callchain. Using the information, their statistics are reported such as overhead portion, function symbol, and the object file as shown in Figure 7.

We extend the Linux perf tool to (1) interpret the *weight* field of blocked samples and (2) annotate a subclass to blocked samples. Firstly, when bperf processes blocked samples, the weight field denotes the number of repetitions of the same event. Therefore, this repetition is taken into account when calculating the statistics. Secondly, bperf examines the subclass of blocked samples and annotates their subclass type in the reported result. Currently, bperf uses the following annotations, `I` for the I/O subclass, `L` for synchronization, `S` for scheduling and `B` for the rest. This information enables the user to identify and analyze the performance impact of blocked samples distinguished by each subclass.

Figure 7 shows the profling result of the example program in Figure 1 using bperf. As compared to the results of the original perf sampling (Figure 2), bperf provides on-CPU and off-CPU events together, thereby allowing developers to understand the overhead of various events more precisely. In Thread 1, `pread` incurs the largest overhead, followed by `pwrite`, and `pthread_cond_wait` ranks third. In Thread 2, `compute_heavy` dominates the thread's execution time.

The advantages of using bperf are two-fold. First, bperf can allow in-depth analysis of blocking events and their interactions inside the operating system kernel. For example, `fsync` is a complex operation that accompanies many types of disk writes. Without bperf, only a tiny amount of user-level and kernel-level codes are collected as on-CPU samples. With bperf, various off-CPU samples are collected to allow in-depth understanding of the `fsync` operation, such as write-back of data blocks, waking up and waiting for `jbd2` file system journaling thread, write-back of journal blocks and commit blocks. Consequently, bperf allows profiling of the interaction between kernel services (e.g., `fsync` call) and accompanying off-CPU events (e.g., data-block writes, synchronization with `jbd2`) with the detailed information of their callchains.

Second, the profiling results using bperf can provide the following performance optimization guidelines.

- When profiling results show a substantial overhead attributed to I/O subclasses, this suggests that the application spends a significant portion of time waiting for I/O operations, such as synchronous I/O. To enhance performance under these conditions, upgrading to faster I/O devices could be considered. Alternatively, adopting asynchronous I/O interfaces could help minimize the blocking time associated with I/O operations [43, 58].
- When profiling results attribute a large overhead of scheduling subclasses, the result indicates the application threads are spending a large fraction of time in CPU runqueues waiting for scheduling. An optimization guideline can be (1) adjusting the number of threads [58], (2) allocating more CPU resources to an application [47, 58], (3) pinning threads to cores to avoid the performance noise caused by CPU load balancers [36, 37], etc.
- When the overhead of the synchronization subclass is notable, performance improvement can be attained by optimizing the events executed within the critical section through application analysis [35].

3.3 BCOZ

This section introduces BCOZ, a causal profiler designed to identify performance bottlenecks. BCOZ is an offline tool that is designed to help programmers identify performance bottlenecks and improve the performance of their programs. At its core, BCOZ profiles on- and off-CPU events collected by blocked samples and estimates performance improvement through virtual speedup. BCOZ precisely identifies interactions between on- and off-CPU events with symbol-level information obtained from blocked samples. This section explores the challenges of accurately estimating the virtual speedup of off-CPU events and discusses various features that are useful for analyzing applications with off-CPU events to optimize performance.

Sampling Kernel Codes. Off-CPU events are tightly coupled with the operation of the operating system kernel codes. Such events occur through the use of operating system services, such as blocking system calls, synchronization primitives, and multi-tasking. In particular, applications with a high number of blocked samples tend to include frequent interactions with kernel [8, 9]. Hence, not only the period during which a thread is blocked but also the kernel operations for such kernel services are important for analyzing and estimating the speedup of their optimization. Accordingly, it is necessary to capture samples for kernel operations and support virtual speedup on those. The original COZ captures only user-space samples for its virtual speedup. However, BCOZ captures samples from not only user space but also kernel space to identify the target of virtual speedup. The target of virtual speedup is basically selected as a part of user-space code. Then, every sample includes their callchain. If the sample includes the instruction pointer of the kernel space, the callchain contains both user-space and kernel-space ones because bperf collects both of them. BCOZ considers them as a unified callchain and identifies the target of virtual speedup by traversing the callchain from the kernel space to the user space.

Virtual Speedup of Blocked Samples. Estimating the virtual speedup of blocked samples is the core of the causality analysis of off-CPU events. BCOZ needs special care when processing blocked samples in order to produce correct virtual speedup estimation. Recall that blocked samples are captured when a thread is scheduled in. Such blocked samples are queued to the perf subsystem and are reported to BCOZ. BCOZ handles blocked samples in batches similar to what COZ does. Processing blocked samples indicates that if a blocked sample is the target of virtual speedup, a proper delay is injected to other co-running threads as in COZ [15]. A blocked sample contains a callchain from the kernel space to the user space. Hence, if the user-space callstack contains target code lines to speed up, the blocked sample becomes the target of virtual speedup. For example, if `pread` is the target of virtual speedup, not only on-CPU samples such as library codes or kernel I/O stack but also off-CPU samples for disk I/Os issued by these code lines are the target of the virtual speedup. This is because such on-/off-CPU samples have a callchain whose user-space part contains `pread`.

Additionally, special care is required in handling dependencies during the virtual speedup of blocked samples. As blocked samples are processed after a blocking event, hence an I/O event, is finished, the processing of blocked samples can incorrectly inject delays to threads which have depen-

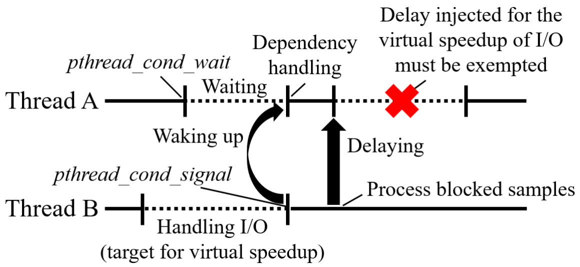


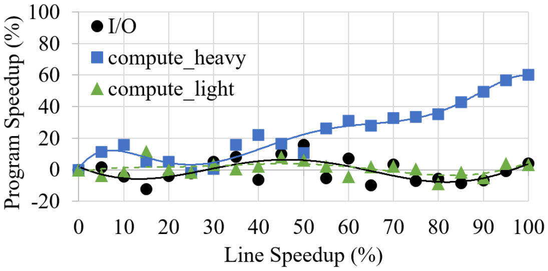
Figure 8: Illustration of virtual speedup when the target includes the off-CPU event while another thread waits for the completion of the target.

dency on the blocking event. For instance, as illustrated in Figure 8, thread A indirectly wait on I/O issued by thread B, and hence after thread B finishes its synchronous I/O, it wakes up thread A. The problem happens if the synchronous I/O becomes the target of speedup. When the blocked samples associated with the I/O are processed sometime after the wake-up operation, the delay injected to thread A, which is for the virtual speedup of the I/O event, makes thread A experience unnecessary delay. In other words, if thread A is injected with the delay, the situation is like the I/O is not sped up. This is because if the I/O event is sped up, the execution of threads waiting for the I/O event directly or indirectly should be boosted as the I/O waiting time is reduced. With the virtual speedup technique, such threads should not be delayed.

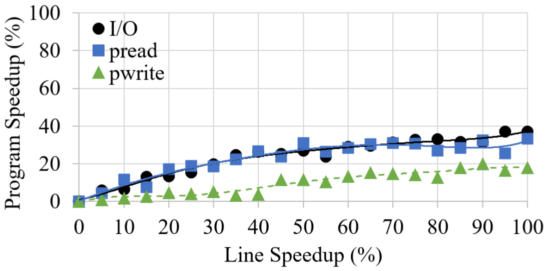
To this end, BCOZ processes blocked samples for virtual speedup before conducting the thread wakeup operations in synchronization primitives (e.g., `mutex_unlock` and `cond_signal`). In the example, the blocked samples of the I/O event are processed before waking up thread A. By doing so, once blocked samples contain the target for virtual speedup, the delays for the virtual speedup are not injected to thread A as thread A is not running. Therefore, the delays are exempted in thread A.

Subclass-Level Virtual Speedup. Optimizing off-CPU events sometimes requires different types of optimization attempts compared to optimizing code. For instance, enhancing the execution environment can often achieve greater performance gains than optimizing the application's code, such as upgrading from HDDs to flash-based SSDs in database applications [51]. Conversely, some applications may exhibit no or marginal performance improvement even when faster hardware devices are employed [23,30,32] due to application-side bottleneck. Similarly, the scheduling subclass off-CPU events have no particular code lines to attempt to optimize. These off-CPU events are challenging to optimize and may require prior knowledge of the optimization effects.

BCOZ provides a subclass virtual speedup technique designed to predict the performance gains from optimizing a specific type of off-CPU events. Hence, the target for virtual speedup is not selected from the application code but from the class of off-CPU events. Applying virtual speedup at subclass granularity is straightforward. During the sample processing, instead of checking whether the sample's callchain includes the target code, it checks whether the *type* field of blocked samples (Section 3.1) matches the target subclass.



(a) BCOZ results of Case 1 in Figure 1



(b) BCOZ results of Case 2 in Figure 1

Figure 9: Profiling results of Figure 1 using BCOZ.

Please note that the synchronization subclass does not support this subclass-level virtual speedup. From an actual optimization perspective, it is not possible to speed up the lock waiting period itself. Instead, the optimization focuses on speeding up the operations in the critical section. Hence speeding up the lock waiting period is invalid but speeding up the critical section is valid. In other words, the virtual speedup of critical sections can be done when the critical section is selected as the target of speedup. Therefore, we exclude the synchronization subclass from the subclass-level virtual speedup technique.

Selecting Virtual Speedup Targets. BCOZ supports both automated virtual speedup target selection and explicit target designation, similar to COZ. The *automated target selection* conducts virtual speedup for multiple targets in a single run by monitoring frequently sampled code lines during execution and dynamically changing the targets [15]. Where BCOZ differs from COZ is that conducting virtual speedup is not limited to on-CPU events executed by the target code line, but also includes virtual speedup for off-CPU events of the target. Additionally, BCOZ can designate an off-CPU subclass rather than a code line as a target for subclass-level virtual speedup. If a particular off-CPU subclass dominates in the sampling result, users can designate such subclass as a virtual speedup target to assess whether optimizing for that off-CPU event can yield performance improvements.

Profiling Results using BCOZ. Figure 9 provides a summary of the virtual speedup results of the example program depicted in Figure 1. This result illustrates that the actual performance improvements can be achieved by optimizing `compute_heavy` in Case 1 and I/O operations (especially `pread`) in Case 2. Furthermore, the virtual speedup results indicate the actual performance gains are bounded by the point at which the critical path moves. Consequently, by filtering out the false bottlenecks, we can prevent unnecessary optimization efforts for users.

4 Evaluation

In this section, we illustrate the experience of application profiling with bperf and BCOZ. The goal of our evaluation is to answer the following questions: (1) Do bperf and BCOZ identify bottlenecks precisely? (2) Does the estimated virtual speedup align with the actual speedup? (3) As compared to other state-of-the-art profilers (i.e., COZ and wPerf), are bperf and BCOZ more useful?

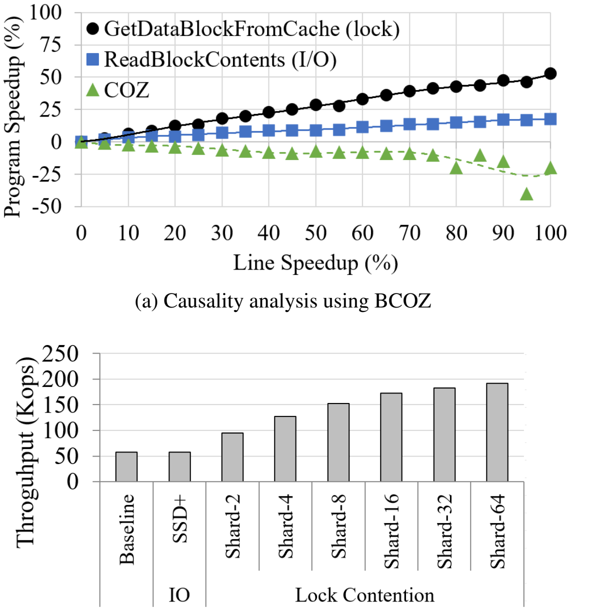
4.1 Experimental Setup

All experiments were conducted on a machine equipped with two Intel Xeon Gold 5218 CPUs (2.30 GHz, 16 physical cores), 375 GB DDR4 DRAM, and a flash-based SSD (PM983), which can deliver performance of up to 540 K I/O operations per second (IOPS). We modified the perf subsystem of the Linux kernel 5.3.7 to support blocked samples, which requires to modification of 295 lines of code. Furthermore, bperf was developed based on the perf tool of the Linux kernel. Finally, BCOZ is implemented based on the existing COZ code [13].

The application codes are complied to include frame pointers to correctly trace callchain. Hence, we disabled frame pointer omitting using `-fno-omit-frame-pointer` option. This is necessary for BCOZ (and COZ [15]).

We present virtual speedup results obtained through either the automated target selection mode or the subclass-level virtual speedup method, as explained in Section 3.3. In the automated target selection mode, the profiler produces virtual speedup graphs for frequently executed lines of code, from which we select the top-N results that demonstrate a positive speedup value. In the subclass-level virtual speedup method, a dominant off-CPU event is manually selected for conducting the virtual speedup profiling.

We measured performance using throughput, defined as the number of processed queries per second, where any performance enhancement reflects as an improvement in throughput. This is based on the assumption that reducing query processing time will naturally lead to increased query throughput, especially in environments where clients continuously submit queries, as in RocksDB. However, as the program speedup in the virtual speedup graphs indicates the percentage of the reduction in execution time, a simple conversion is necessary to translate it to the throughput improvement. A program speedup of y% means that the time it takes to process the same number of queries is now reduced to $(100-y)\%$ of the original, indicating that the throughput has increased by $\frac{100}{100-y}$ times. For instance, 75% program speedup is translated as 4x throughput improvement.



(a) Causality analysis using BCOZ

(b) Optimization results

Figure 10: Results of (a) causality analysis, and (b) actual optimization in *Prefix Dist*.

4.2 Case Study: RocksDB

In this section, we provide our experience of profiling RocksDB, a widely used log-structured merge (LSM) tree-based key-value store. By profiling RocksDB in various system configurations, we aim to identify off-CPU event bottlenecks that were previously difficult to pinpoint. We also validate these bottlenecks by attempting actual optimizations or comparing them with findings from previous studies on RocksDB optimization.

Optimization 1: Block Cache Contention. As a first optimization, we identify and address the bottleneck of block cache operations in a read-intensive workload. Figure 10 shows the profiling results for read-only execution of *Prefix Dist* [10], an open-sourced real-world workload by Facebook. In this experiment, the key-value size is 91 bytes (48 bytes key and 43 bytes value), the block cache size is 10 GB, the workload runs eight worker threads, and the dataset is set to 1-billion key-value pairs. The workload performs using a single shard to reproduce the well-known lock contention problem of RocksDB's LRU-based block cache [6].

Figure 10a illustrates the virtual speedup results using by BCOZ (solid line) and COZ (dotted line). In the results, two operations, `GetDataBlockFromCache` and `ReadBlockContents` are identified as bottlenecks. The

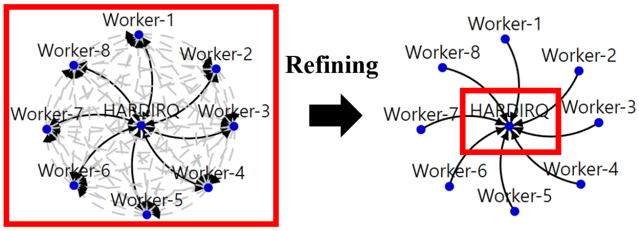
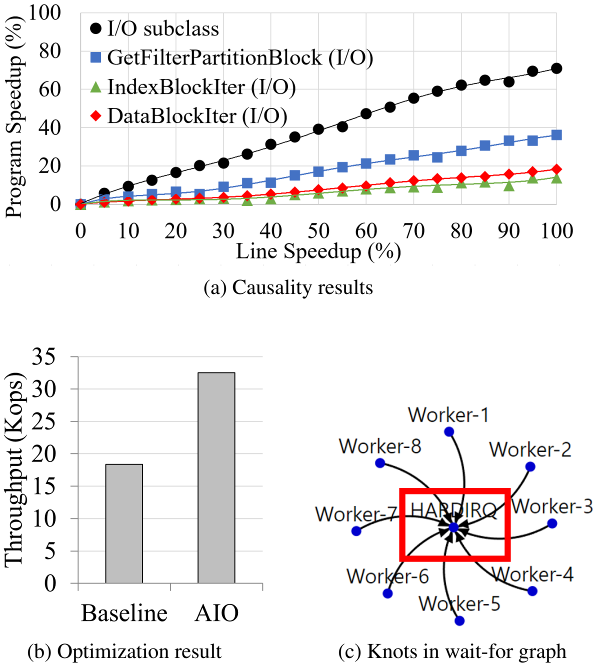


Figure 11: The wait-for graph and identified knots using wPerf

worker threads of RocksDB handles get operations by (1) looking up desired blocks (e.g., filter, index and data block) from the block cache (`GetDataBlockFromCache`) and (2) issuing block I/O requests to underlying disks upon a cache miss from the block cache (`ReadBlockContents`). The cache lookup operation is the real bottleneck since all the workers are contending on the lock of the block cache. As shown in Figure 10a, BCOZ shows up to 60% speedup when the cache lookup operation is optimized and up to 20% speedup when the block read I/O operation is optimized. Although the two operations accompany off-CPU events, COZ does not show any virtual speedup result for the two operations because it cannot take off-CPU events into account for profiling.

In order to verify the virtual speedup results, we conducted optimization of the two operations. First, we replaced the flash-based SSD with a faster one [44] which delivers performance of up to 1,500 K IOPS; this optimization is denoted as *SSD+*. We expect the speedup of `ReadBlockContents`. Figure 10b, however, shows no performance gain with SSD+ since the lock contention is the major bottleneck. Our second optimization is to apply sharding. The block cache is partitioned to multiple shards, which is denoted as `Shard-N` where N is the number of shards. As shown in the figure, Shard-N shows improved performance. The more shards there exist, the less lock contention occurs thereby showing more improved throughput. This tendency is shown in the virtual speedup analysis of BCOZ in Figure 10a.

To compare profiling capability of wPerf, we use wPerf to analyze the application again. Figure 11 shows the profiling results of wPerf, the initial knot (left) and the knot after trimming edges with a small global impact [58] (right). After trimming the low-weight edges (right), the wait-for graph identifies only the I/O (`HARDIRQ`) as the bottleneck. This profiling result requires a programmer's additional efforts to identify which user-level functions incur such bottlenecks. In addition, improving the I/O performance (SSD+) does not lead to actual performance improvement, which may increase the user's burden of profiling. The wait-for graph before trimming (left) is too complex but provides two bottleneck points: (1) worker --> HARDIRQ edge indicates the I/O waiting of worker threads and (2) worker <--> worker edge indicates lock waiting between worker threads. Therefore, the strategy that the user can try is to optimize the I/O event of the worker thread or to solve the lock contention. However, wPerf does not provide the potential speedup when the bottlenecks are optimized. One lucky programmer may attempt to optimize the lock contention and success to improve the performance. Meanwhile, one unlucky programmer may attempt to optimize the block read I/O operations and may not be able to improve the performance. Therefore, the missing causality analysis of wPerf can increase the burdens of users.



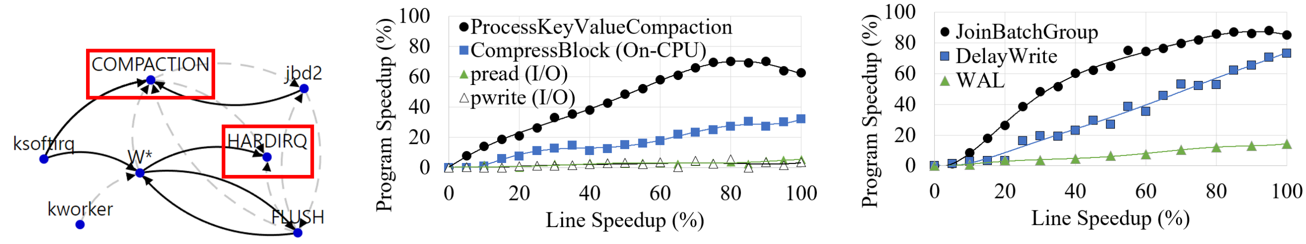
(a) Causality results

(b) Optimization result

(c) Knots in wait-for graph

Figure 12: Results of (a) causality analysis, (b) actual optimization, and (c) wPerf in *all random*.

Optimization 2: Block Read Operation. After resolving the block cache contention by sharding, we profiled RocksDB again with the *all random* workload in order to identify and optimize off-CPU events. The configurations of the workload remain unchanged, except that the block cache size is reduced to 128 MB to incur a large amount of off-CPU events, hence disk read I/Os. Figure 12 shows the profiling result of the workload using BCOZ and wPerf. The profiling results of BCOZ demonstrate the three bottleneck points, `IndexBlockIter`, `GetFilterPartitionBlock`, `DataBlockIter`, which handle index blocks, filter blocks and data blocks, respectively. All the operations are off-CPU-intensive operations. In addition, the I/O subclass-level virtual speedup is applied and depicted as `I/O subclass`. As shown in the figure, each of the three operations shows from 15% to 40% of speedup and their aggregated speedup is more than 70% (I/O subclass). Among the three operations, `GetFilterPartitionBlock` shows the largest expected speedup. An LSM tree has multiple levels and a key-value entry can exist in any of the levels. In each level, a key-existence test is done using a bloom filter. Therefore, `GetFilterPartitionBlock` performs the key existence testing using the filter blocks that need to be fetched from the disk. Since a single get operation traverses multiple



(a) Knots in the wait-for graph of wPerf.

(b) Compaction thread.

(c) Worker thread.

Figure 13: Results of (a) wPerf, (b) compaction thread, and (c) worker thread in *fillrandom*.

levels of the LSM tree, optimizing the filter read I/O operation is expected to show the largest performance gain.

Our optimization strategy is to apply asynchronous I/O to fetch filter blocks of the next levels of the LSM tree. Hence, for a get operation, our optimization performs a read of the filter blocks speculatively through the LSM tree. Hence, while the first level is processing the key existence test, the filter blocks of successive levels are speculatively fetched thereby reducing the read I/O waiting time. Our experimental results show that our optimization results in 77.3% of performance gain as shown in Figure 12b.

We compared the profiling result of BCOZ with wPerf. The wait-for graph of wPerf is shown in Figure 12c. As shown in the figure, the knot is `HARDIRQ`, which denotes the disk. However, wPerf provides no more information on the bottleneck. Hence, the user's additional effort is necessary to specify the bottleneck. In addition, the lack of the causality analysis causes users to hesitate about which of the I/O operations to speed up. However, BCOZ provides the causality analysis to off-CPU events and illustrates that the filter I/O operations have the largest potential speedup. This analysis is followed by successful optimization of the filter I/O operations by parallelizing filter I/O operations by using asynchronous I/O.

Optimization 3: Write-intensive Workload. Our next optimization attempt is the write-intensive workload, *fillrandom* against RocksDB. For this experiment, we utilized a key-value size of 1 KB, with 16 worker threads concurrently writing a total of 10 million records. Figure 13 shows the profiling result using BCOZ and wPerf. First, wPerf identifies two potential bottlenecks: the `COMPACTION` thread and the `HARDIRQ` I/O thread. The initial wait-for graph was too complex and we applied the *merging similar threads* technique [58] to obtain the wait-for graph of the figure. From the wait-for graph, we can identify two bottlenecks: (1) the compaction operation of the LSM tree and (2) the write I/O operations of the worker threads (`W*`, the merged worker threads). Since the worker threads perform write-ahead-logging (WAL), WAL write I/Os can be the bottleneck. For the compaction, wPerf does not specify among the operations of compaction the significant overhead. In addition, wPerf reports that `HARDIRQ` shows a bigger global impact than `COMPACTION`.

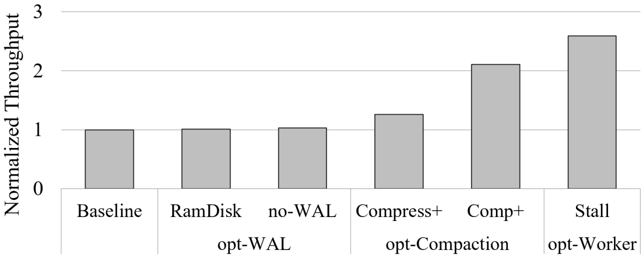
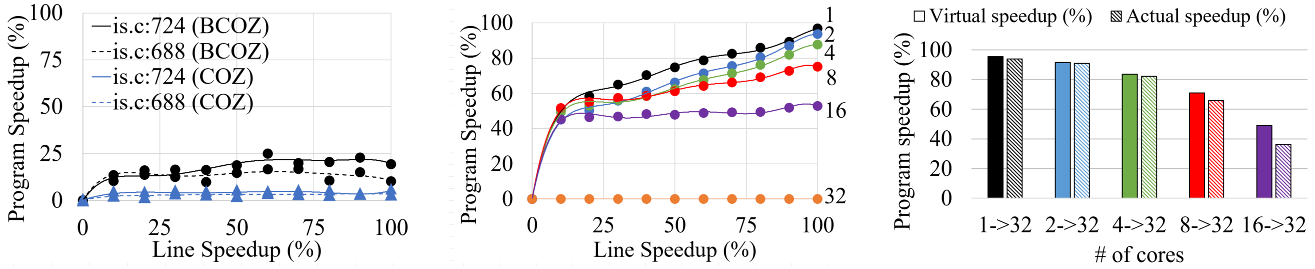


Figure 14: Results of actual optimizations in *fillrandom*.

Meanwhile, BCOZ provides more informative analysis results. First, BCOZ identifies the compression operation (`CompressBlock`), which is on-CPU, as the significant bottleneck as shown in Figure 13b. Although BCOZ can estimate the potential speedup of I/O operations (`pread` and `pwrite`), their speedup is expected to be marginal. Second, BCOZ expects marginal speedup for the WAL operations as shown in Figure 13c. Third, BCOZ analyzes that the contention between the worker threads (`JoinBatchGroup`) is the bottleneck and its optimization can potentially improve the performance. Finally, BCOZ identifies the write throttling (`DelayWrite`) of the RocksDB's memtable write policy as the bottleneck.

The actual optimizations of the identified bottlenecks are performed as follows. First, the WAL operation is optimized by (1) using a *RamDisk* as the WAL storage and (2) disabling WAL (*no-WAL*). Second, the compaction operation is optimized by (1) disabling compression (*Compress+*) and (2) allocating many compaction threads (*Comp+*). Figure 14 shows the performance of the RocksDB *fillrandom* workload when the optimizations are applied. Third, the writeback stall is relieved by increasing the number of maximum memtables from 2 to 16 (*Stall*). As shown in the figure, WAL-related optimizations show marginal performance gain. This result indicates that BCOZ estimates the potential speedup correctly. In addition, *Compress+* and *Comp+* show improved performance, which is also predicted by BCOZ. Also, *Stall* shows the largest performance gain, which is estimated by BCOZ in Figure 13c. In the meantime, wPerf only identifies `COMPACTION` and `HARDIRQ` as the potential bottleneck points. `HARDIRQ` has shown marginal performance since worker threads are reported to wait for the I/O thread but accelerating WAL opera-



(a) Line-level virtual speedup

(b) Subclass-level virtual speedup

(c) Virtual speedup vs. actual speedup

Figure 15: Virtual speedup results of NPB-is under the CPU contention.

tions show no performance gain. From these results, BCOZ is effective in specifying performance bottlenecks.

Profiling Accuracy. Although BCOZ is proven to effectively identify performance bottlenecks, predicting the potential speedup accurately is sometimes difficult and intricate. For example, BCOZ predicts that optimizing the block cache lock contention can lead to 50% of program speedup (or doubling the throughput) as shown in Figure 10a. However, after relieving the block cache contention by applying sharding, the performance has been improved by four times, which is more than two times higher than the expected improvement. This is because sharding not only relieves the lock contention of cache lookup (`GetDataBlockFromCache`) but also reduces the contention of other operations (e.g., `PutDataBlockToCache`). Furthermore, certain optimizations may speed up a target code line but increase the amount of other events. For example, BCOZ predicts that optimizing `CompressBlock` will result in a 32% program speedup (or 1.47x throughput improvement) in Figure 13b, but the actual throughput improvement is 1.26x. This is because, by disabling compression, `CompressBlock` is no longer invoked, but this optimization has the side effect of increasing the total amount of disk I/Os.

Therefore, virtual speedup results can under- or overestimate actual speedup. However, the strength of BCOZ (and COZ) lies in their ability to precisely identify specific lines that could potentially lead to actual speedup when optimized. Our evaluation results have shown that optimizing these predicted lines can indeed lead to actual speed improvements.

Takeaway. During the profiling of RocksDB, we have identified bottlenecks that align with those mentioned in existing RocksDB optimization studies. Firstly, we examined the overhead and virtual speedup results of the compaction, which involved a mix of off-CPU events (I/O subclass) and on-CPU events. The potential for performance improvement through compaction optimization, as indicated in numerous studies [2,12,28,42,48], was validated using bperf and BCOZ. The significance of the proposed profiling techniques has become evident as the existing Linux perf tool or COZ could not easily or accurately predict the bottlenecks in the absence of blocked samples. Furthermore, the necessity to optimize operations related write stalls and batched group writing was also identified by BCOZ. These operations have been also identified as bottlenecks in previous studies [11,18,24,29,34,52,56]. Finally, we have examined the bottleneck caused by I/O subclass off-CPU events during the execution of RocksDB and validated the performance improvement.

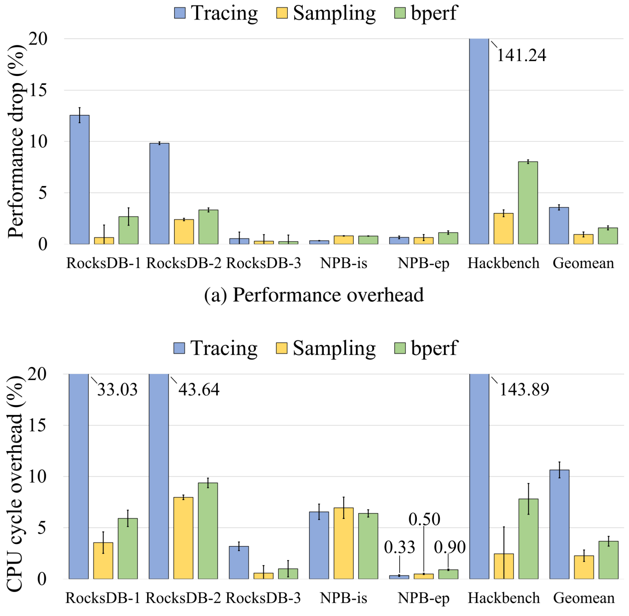
4.3 Case Study: NAS Parallel Benchmark

In this section, we evaluate the effectiveness of the scheduling subclass and subclass-level virtual speedup. For this experiment, we use a compute-intensive workload, *is* (integer sort) from the NAS parallel benchmark [5].

As discussed in Section 3.3, virtual speedup of application code lines can be ineffective if application threads are contending on the CPU cores. To demonstrate this situation, we intentionally controlled the number of CPU cores assigned to the NPB-is workload. The workload is configured to run 32 threads and the number of cores is varied from 1 to 32 cores. Figure 15a illustrates the profiling result of the main computation code lines using COZ and BCOZ when the number of cores is limited to one. As shown in the figure, COZ has estimated marginal performance improvement. On the contrary, BCOZ has predicted potential performance improvement if these code lines are optimized. Under high CPU contention (32 threads vs. 1 core), off-CPU events are frequent as the threads are frequently scheduled out due to the high CPU contention. In this case, BCOZ is able to estimates the optimization opportunity that when such scheduling subclass off-CPU events are removed, the performance can be improved.

Figure 15b presents the profiling results for the scheduling subclass-level virtual speedup as the number of cores increases from 1 to 32. With the highest CPU contention (using only one core), the estimated program speedup is at its peak. As the number of cores increases, thereby reducing CPU contention, the estimated program speedup decreases. Since the number of assigned CPU cores is fewer than the program's 32 threads, these profiling results seems valid.

To validate the virtual speedup profiling results, we measure the program performance with varying the number of cores. This approach reflects the optimization strategy of allocating additional CPU cores to mitigate CPU contention. Fig-



(a) Performance overhead

(b) CPU cycle overhead

Figure 16: Overhead analysis results of bperf.

ure 15c shows both the virtual and actual program speedups as the number of cores changes from X to 32 (X->32), transitioning from high CPU contention to no CPU contention. As shown in the figure, the actual speedup generally corresponds with the speedup predicted by BCOZ. These results confirms that it is important to correctly profile off-CPU events in highly parallel workload, and BCOZ leverages blocked samples to provide valuable profiling results to users.

4.4 Profiling Overhead

bperf. We compare the overhead of bperf with existing profiling techniques, (1) *tracing* which profiles only off-CPU events (`sched_switch` and `sched_wakeup`) using Linux perf's tracing mode [17, 38, 58], and (2) *sampling* which samples only on-CPU events using Linux perf's sampling mode (`task_clock`) [17]. All these tools have the same goal of profiling statistical overheads of target programs, but their profiling coverage are different: *tracing* focuses on off-CPU events, *sampling* focuses on on-CPU events, and *bperf* covers both on- and off-CPU events. For the sampling methods, the sampling period is set to 1 ms, which is identical in all the experiments in the evaluation section.

Figure 16 shows the overhead of the three profilers with the workloads used in the experiments as well as additional workloads, *NPB-ep* [5] and *hackbench* [22] to cover on-CPU-intensive and off-CPU-intensive cases, respectively. Among the workloads, *RocksDB-X* indicates the RocksDB workload used in Optimization-*X* in Section 4.2. We measured two types of overhead. First, the performance overhead refers to the performance drop of profiled applications (runtime increase for hackbench and throughput decrease for the rest of applications) as compared to the baseline, which runs without any profilers (Figure 16a). Second, the extra CPU overhead refers to the additional CPU cycles consumed by using the profilers as compared to the baseline (Figure 16b). The values shown in the figures are the average of 10 runs.

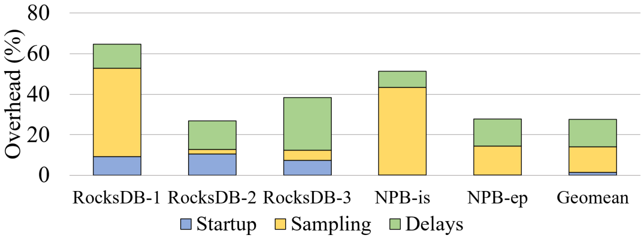


Figure 17: Overhead breakdown results of BCOZ.

Overall, the three profilers show acceptable performance drops, no more than 4% on average. Tracing, sampling and bperf have shown the average performance drop by 3.6%, 0.9% and 1.6%, respectively. Specifically, tracing shows notable performance drops with the workloads showing frequent off-CPU events (i.e., RocksDB-1, 2 and hackbench). Tracing records profiling information (i.e., IP and callchain) on every thread state transitions, leading to high overhead when off-CPU events are frequent. In contrast, bperf hooks into all the state transitions like tracing but only records timestamps for each transition. Profiling information is recorded only if an off-CPU interval overlaps with sampling points, resulting in low profiling overhead.

As compared to sampling, bperf enables profiling of off-CPU events at a low cost. bperf shows the additional performance overhead by only 0.7% (1.6% for bperf – 0.9% for perf) and extra CPU cycles by 1.4% (3.7% for bperf – 2.3% for perf). Considering that the profiling capability of bperf is greater than perf, we believe these overheads are acceptable.

As the mechanism of bperf inherits from perf, it can be attached and detached at any time while the profiling target application is running. As the overhead of bperf is low, we believe it can be an online tool for collecting statistical overheads of applications in production.

BCOZ. Figure 17 illustrates the overhead of profiling applications with BCOZ. The BCOZ overhead is categorized into three parts: startup, sampling, and delays. Startup represents the overhead of collecting debug information, which is the duration between BCOZ's bootstrapping and the application's main function (i.e., `libc_start_main()`), on average 1.4%. Sampling encompasses the overhead incurred when there's no virtual speedup delay (i.e., 0% line speedup) consisting of the bperf's sampling overhead and BCOZ's intervention to read samples and verify whether they include speedup targets, on average 12.6%. Finally, delays entail additional overhead when BCOZ is fully enabled, averaging 13.6%. The delays overhead does not refer to the amount of delays injected for virtual speedup, but the increase in the end-to-end execution time of the application.

The overhead of BCOZ is not light considering its end-to-end overhead of 27.6% on average and up to 64.7%. However, such profilers show similar performance overheads. For example, COZ has demonstrated its overhead of 17.5% on average and up to 65% [15]. As these profilers insert additional delays while the application is running, the end-to-end execution time can be increased. These overheads can be reduced when the inserted delays are limited, cooling-off times are inserted between virtual speedup experiments, etc [15]. Such remedies can be effective in reducing the profiling overhead, but it remains uncertain whether the results provided are sufficient to identify bottlenecks and their potential for performance gain.

5 Related Work

On-CPU Event Profilers. Conventional profilers [17, 19–21, 33] that rely on existing on-CPU events (such as CPU usage and execution time) face challenges when identifying bottlenecks in modern applications. This is primarily because the event with the longest execution time in a multi-threaded application does not necessarily represent the critical performance path, and they do not account for off-CPU events. In the context of multi-threaded applications, there are causal profiling studies aimed at analyzing the impact of optimizing each individual event on overall application performance [1, 4, 7, 15, 45, 53, 54]. COZ [15], for instance, provides performance improvement predictions by applying virtual speedup to each event using the sampling results from the Linux perf subsystem. However, COZ's virtual speedup is limited to on-CPU events sampled by the Linux perf subsystem. It is not capable of estimating virtual speedup of events that include off-CPU events.

Off-CPU Event Profilers. Existing studies have focused on analyzing off-CPU event bottlenecks [27,35,38,39,55,57,58]. Some studies analyze application bottlenecks by measuring the duration of off-CPU events [27,39,55]. However, in multi-threaded applications, the longest event may not always represent the critical path of the application [15, 58]. Furthermore, nested off-CPU events can have varying performance impact on event duration and overall application performance [58]. Therefore, analyzing performance using the duration of off-CPU events leads to incorrect conclusions.

Other studies identify application bottlenecks by specifically targeting off-CPU events related to synchronization [35, 57]. However, as mentioned earlier, the off-CPU bottlenecks in modern applications are diverse and encompass various aspects, including device I/O. Therefore, relying on profiling specific off-CPU events has limitation of supporting the various applications.

wPerf [58] is a state-of-the-art study focused on analyzing off-CPU bottlenecks in applications, wait-for graphs are constructed to identify off-CPU events that act as bottlenecks. However, as discussed in Section 2.2, wPerf has several limitations. wPerf does not precisely pinpoint the performance bottleneck of applications. Also, wPerf lacks the capability of causality analysis, so it could not analyze the actual impact on application performance when optimizing a performance bottleneck. Finally, wPerf identifies bottlenecks but misses detailed information, requiring additional efforts from developers to understand the exact performance bottleneck.

6 Conclusion

Existing profilers face limitations when it comes to identifying modern application bottlenecks that involve a mix of on- and off-CPU events. These profilers treat on- and off-CPU events as separate dimensions, making it difficult to perform comprehensive profiling and interpret the results. Moreover, even if the bottleneck of an application is identified, it remains uncertain whether optimizing the bottleneck will result in actual performance improvements. To address this problem, this paper introduces a sampling technique called blocked samples, which enables the identification of application bottlenecks by integrating on- and off-CPU events within the same dimension. We present bperf, a Linux perf tool that utilizes the proposed blocked samples technique to identify application bottlenecks based on event execution time, and BCOZ, a causal profiler that offers a virtual speedup for off-CPU events. By profiling the RocksDB application using these two profilers, we are able to uncover previously unidentified bottlenecks related to I/O and synchronization tasks. Furthermore, by virtually speeding up these tasks, we identify optimization possibilities that were overlooked in existing RocksDB optimization studies.

We plan to extend blocked samples to include richer information for profiling. The current blocked samples consider the operations inside an I/O device as a black box. However, I/O devices may have their internal operations, which can be the hint of performance optimization opportunities for applications. For example, disk-internal events, such as garbage collection, and valid page copying, are important events for storage applications to establish their optimization strategies. In this regard, we plan to augment blocked samples with I/O device-internal operations thereby allowing applications to employ expanded optimization strategies.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, for their valuable comments and feedback. This work was supported in part by Samsung Electronics and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688). Jinkyu Jeong is the corresponding author.

Availability

The source code is available at https://github.com/s3yonsei/blocked_samples.

References

[1] Minwoo Ahn, Donghyun Kim, Taekeun Nam, and Jinkyu Jeong. Scoz: A system-wide causal profiler for multicore systems. *Software: Practice and Experience*, 51(5):1043–1058, 2021.

[2] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *USENIX Annual Technical Conference*, pages 753–766, 2019.

[3] BCC (BPF Compiler Collection). https://github.com/iovisor/bcc/tree/master.

[4] Zachary Benavides, Keval Vora, and Rajiv Gupta. Dprof: distributed profiler with strong guarantees. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–24, 2019.

[5] NAS Parallel Benchmarks. Nas parallel benchmarks. *CG and IS*, 2006.

[6] Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache.

[7] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. A parallelism profiler with what-if analyses for openmp programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 198–211. IEEE, 2018.

[8] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Tappan Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[9] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *FAST*, volume 21, pages 17–32, 2021.

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference*, 2020.

[13] Source code of COZ. https://github.com/plasma-umass/coz.

[14] cpudist in bcc. https://github.com/iovisor/bcc/blob/master/tools/cpudist.py.

[15] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[16] CXL memory expander. https://semiconductor.samsung.com/us/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/.

[17] Arnaldo Carvalho De Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, 2010.

[18] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. Trianglekv: Reducing write stalls and write amplification in lsm-tree based kv stores with triangle container in nvm. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4339–4352, 2022.

[19] DTrace. http://dtrace.org/blogs/.

[20] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, pages 120–126. ACM, 1982.

[21] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[22] Hackbench. https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/sched/cfs-scheduler/hackbench.c.

[23] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Read as needed: Building wiser, a flash-optimized search engine. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 59–73, 2020.

[24] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. Flatlsm: Write-optimized lsm-tree for pm-based kv stores. *ACM Transactions on Storage*, 2023.

[25] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[26] JCOZ. https://github.com/Decave/JCoz.

[27] Jprofiler. https://www.ej-technologies.com/products/jprofiler/overview.html.

[28] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. Bolt: Barrier-optimized lsm-tree. In *Proceedings of the 21st International Middleware Conference*, pages 119–133, 2020.

[29] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[30] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast nvm storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[31] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *USENIX Annual Technical Conference*, pages 603–616, 2019.

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[33] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[34] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043. IEEE, 2021.

[35] Tongping Liu, Guangming Zeng, Abdullah Muzahid, et al. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313. ACM, 2017.

[36] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[37] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 852–863. IEEE, 2011.

[38] Off-CPU analysis. https://www.brendangregg.com/offcpuanalysis.html.

[39] Off-CPU Flame Graph. https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html.

[40] offcputime in bcc. https://github.com/iovisor/bcc/blob/master/tools/offcputime.py.

[41] INTEL. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[42] Fengfeng Pan, Yinliang Yue, and Jin Xiong. dcompaction: Delayed compaction for the lsm-tree. *International Journal of Parallel Programming*, 45:1310–1325, 2017.

[43] Jongwon Park and Jinkyu Jeong. Speculative multilevel access in lsm tree-based kv store. *IEEE Computer Architecture Letters*, 21(2):145–148, 2022.

[44] Specification of pm1735 nvme ssd. https://semiconductor.samsung.com/us/ssd/enterprise-ssd/pm1733-pm1735/.

[45] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):27, 2019.

[46] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. Only relative speed matters: Virtual causal profiling. *ACM SIGMETRICS Performance Evaluation Review*, 48(3):113–119, 2021.

[47] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[49] Connectx nics. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[50] RocksDB. https://github.com/facebook/rocksdb.

[51] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: reducing write stalls and write amplification in lsm-tree based kv stores with a matrix container in nvm. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 17–31, 2020.

[53] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 15–26, 2017.

[54] Adarsh Yoga and Santosh Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501, 2019.

[55] Yourkit (Java and .NET profiler). https://www.yourkit.com/.

[56] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. Adoc: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 65–80, 2023.

[57] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400, 2016.

[58] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, 2018.

[59] SAMSUNG. Ultra-low latency with SAMSUNG Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

A Artifact Appendix

A.1 Abstract

Blocked samples is a profiling technique based on sampling, that encompasses both on- and off-CPU events simultaneously. Based on blocked samples, we present two profilers: *bperf*, an easy to-use sampling-based profiler and *BCOZ*, a causal profiler that profiles both on- and off-CPU events simultaneously and estimates potential speedup of optimizations.

A.2 Scope

Our artifact can be used to identify bottlenecks across various applications and pinpoint code lines in need of optimization. Particularly, our approach is an efficient profiling technique for applications where both on- and off-CPU events are mixed.

A.3 Contents

Our artifact consists of three subdirectories: `blocked_samples` (source code of Linux kernel with bperf), `bcoz` (source code of BCOZ), and `osdi24_ae` (OSDI'24 artifacts evaluation). Descriptions of each subdirectory are as follows.

blocked_samples. This directory includes an extended Linux perf subsystem for blocked samples. Blocked samples is a profiling technique based on sampling, that encompasses both on- and off-CPU events simultaneously. Furthermore, the original Linux perf tool is replaced with our bperf (`blocked_samples/tools/perf`).

bcoz. This directory includes source code of BCOZ. BCOZ is a causal profiler that leverages the concept of virtual speedup for both on-CPU and off-CPU events using blocked samples. At its core, BCOZ profiles on-/off-CPU events (i.e., blocked samples) collected by our extended Linux perf subsystem and estimates performance improvement through virtual speedup. BCOZ is extended from COZ [15], a causal profiler for only on-CPU events.

osdi24_ae. This directory is for the OSDI '24 artifacts evaluation. It includes instructions for reproducing the experimental results in the paper.

The instructions in the *Getting Started with Blocked Samples* section of `README.md` in the root directory help verify whether blocked samples functions correctly.

A.4 Hosting

The GitHub repository for the artifacts is available on https://github.com/s3yonsei/blocked_samples.

A.5 Requirements

The Linux kernel version for blocked samples is 5.3.7 and we have verified that blocked samples operates correctly on the Ubuntu 20.04 LTS server. We will soon release the support for blocked samples on the latest Linux kernel version in the repository.

dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving

Bingyang Wu[1] *Ruidong Zhu*[1] *Zili Zhang*[1] *Peng Sun*[2] *Xuanzhe Liu*[1] *Xin Jin*[1]
[1] *School of Computer Science, Peking University* [2] *Shanghai AI Lab*

Abstract

Low-rank adaptation (LoRA) is a popular approach to finetune pre-trained large language models (LLMs) to specific domains. This paper introduces dLoRA, an inference serving system for LoRA models. dLoRA achieves high serving efficiency by dynamically orchestrating requests and LoRA adapters in terms of two aspects: (*i*) dynamically *merge* and *unmerge* adapters with the base model; and (*ii*) dynamically *migrate* requests and adapters between different worker replicas. These capabilities are designed based on two insights. First, despite the allure of batching without merging a LoRA adapter into the base model, it is not always beneficial to unmerge, especially when the types of requests are skewed. Second, the autoregressive nature of LLM requests introduces load imbalance between worker replicas due to varying input and output lengths, even if the input requests are distributed uniformly to the replicas. We design a credit-based batching algorithm to decide when to merge and unmerge, and a request-adapter co-migration algorithm to decide when to migrate. The experimental results show that dLoRA improves the throughput by up to 57.9× and 26.0×, compared to vLLM and HugginFace PEFT, respectively. Compared to the concurrent work S-LoRA, dLoRA achieves up to 1.8× lower average latency.

1 Introduction

Large language models (LLMs) are changing the landscape of modern applications. LLMs such as GPT4 [1] and Llama-2 [2] are pre-trained on a large corpus to achieve outstanding capabilities on generic tasks. These pre-trained LLMs (a.k.a. base LLMs) can be fine-tuned to a specific domain to optimize particular application scenarios, e.g., fine-tuning Llama-2 for better code generation [3]. LLM platforms [4–7] provide fine-tuning APIs and services for developers to fine-tune LLMs and build domain-specific applications. For example, OpenAI provides fine-tuning APIs for fine-tuning GPT-4 and Completions API to access these fine-tuned LLMs [4].

Low-rank adaptation (LoRA) [8, 9] is a popular approach to fine-tuning LLMs. It is a type of parameter-efficient fine-tuning [10] that reduces fine-tuning costs by updating only a small portion of model parameters. LoRA exploits the low dimensionality of parameter updates in fine-tuning and represents them with pairs of two small matrices called LoRA adapters. Fine-tuning a base LLM amounts to training a LoRA adapter for a specific domain while keeping the base LLM unchanged. Compared to the fully fine-tuning GPT-3 175B, LoRA can reduce the number of updated parameters by 10,000× and the GPU consumption by 3× while achieving comparable model quality [8]. At inference time, the LoRA adapter can be merged with the base LLM thus introducing no extra inference overhead.

Serving a set of LoRA model fine-tuned on a base LLM (i.e., LoRA as a service) introduces new challenges to LLM inference serving. Existing LLM inference systems such as Orca [11] and vLLM [12] focus on serving a single model, while in the scenario of LoRA as a service, there are multiple models. Conceivably, one can use Orca or vLLM to serve each LoRA model and adopt an existing model serving orchestrators like SHEPHERD [13] and AlpaServe [14] to manage multiple LoRA models. This simple approach does not consider the characteristics of LoRA model serving and has the following two fundamental problems.

First, serving each LoRA model separately introduces a high memory footprint, and suffers from low GPU utilization as the GPUs cannot be efficiently multiplexed across models. The problem is particularly acute for LLMs as LLMs have large sizes. An alternative approach is to directly use *unmerged* LoRA adapters, i.e., keeping the base LLM unchanged and storing the set of LoRA adapters alongside. When serving different types of requests, it can batch the shared base LLM computation across requests to increase efficiency. This unmerged approach, however, does not work well when the requests are skewed on a particular LoRA adapter, whereas the merged approach can further reduce computational costs to increase efficiency.

Second, LLM tasks have variable input and output lengths, which naturally introduces load imbalance between different worker replicas. Due to the autoregressive pattern of LLMs, simply dispatching requests to different worker replicas uniformly does not work well, as the execution time and GPU memory consumption of requests are diverse [11, 12]. The LoRA as a service scenario further exacerbates the problem, as LoRA adapter orchestration across worker replicas also needs to be taken into consideration. The dependency between LoRA adapters and requests as well as the GPU memory competition between them make the problem even harder than traditional load balancing problems.

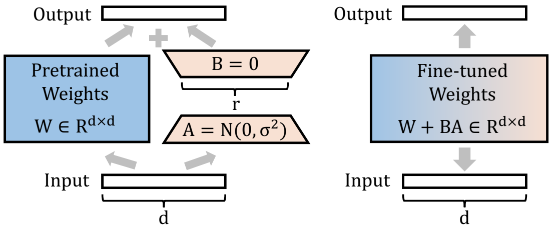
We introduce dLoRA, a new inference serving system for LoRA models to address these two problems. Compared to previous practices in LoRA serving, dLoRA further improves efficiency with two special capabilities. First, dLoRA can dynamically *merge* and *unmerge* LoRA adapters with the base model in each worker replica. Second, dLoRA can dynamically *migrate* LoRA adapters and requests between worker replicas. Exploiting the capabilities of dLoRA efficiently has two technical challenges. The first one is how to decide when to merge and unmerge adapters. The second one is how to decide which adapters and requests to migrate. We propose two techniques to address these two problems.

To address the first problem, we propose a dynamic cross-adapter batching technique. Based on the request arrival pattern and current state, dLoRA dynamically switches between merged and unmerged inference with different batching strategies to reduce the end-to-end latency. To decide an appropriate switching time, dLoRA dynamically adjusts the switching threshold with the thresholds tuning according to the request pattern to reduce switching overhead. dLoRA adopts a credit-based batch generator to generate potentially efficient batching plans without harming the fairness of requests. A final decision is made by taking all factors into account, including execution time, queuing delay, and switching overhead.

To address the second problem, we propose a request-adapter co-migration technique. Except for proactively dispatching requests based on current requests and adapter distribution, it also dynamically migrates requests and replicates adapters to handle unpredictable load imbalance across replicas. We formulate the problem as an integer linear programming (ILP) problem and compute an optimal solution to minimize the load imbalance. To address the high overhead of ILP, we amortize the overhead by reducing the frequency of ILP solving and relaxing the problem to a selective migration problem inspired by selective replication [15–17].

In summary, we make the following contributions.

- We identify the inefficiencies of current LLM serving systems in the LoRA model serving scenario, and articulate the challenges of serving LoRA models.
- At the worker level, we propose a dynamic cross-adapter batching technique to dynamically switch between merged and unmerged modes to reduce the end-to-end latency.
- At the cluster level, we propose a request-adapter co-migration technique to dynamically migrate requests and adapters to balance the load across the cluster.
- We design and implement dLoRA with the preceding two techniques. The evaluation results based on real-world workload traces show that dLoRA achieves up to 57.9× and 26.0× higher throughput than vLLM [12] and HuggingFace PEFT library [10], respectively. By leveraging dynamic orchestration mechanisms, dLoRA also achieves up to 1.8× lower average latency compared to the concurrent work S-LoRA [18].



(a) LoRA fine-tuning. (b) LoRA inference.

Figure 1: LoRA optimization.

2 Background and Motivation

2.1 Parameter-Efficient Large Language Models

Large language models. Large language models (LLMs) try to maximize the next token predictability given the previous tokens. At the inference time, LLMs show an autoregressive pattern. For each request, an LLM iteratively generates tokens based on the prompt (i.e., input tokens) and previous output tokens once a time until it generates an end-of-sentence marker. This autoregressive nature makes LLM inference exhibit two characteristics. The first one is variable inference latency depending on the input and output lengths [12, 19]. The second one is significant GPU memory consumption of intermediate states of requests. To reduce redundant computation, LLMs cache the intermediate states of previous tokens, called key-value (KV) cache, in the GPU memory [20]. Similar to prior work [12], we use intermediate states to refer to the key-value cache. As the size of the KV cache is proportional to the number of input and output tokens, the memory consumption of LLM inference is also variable. The KV cache consumes a large amount of GPU memory and may bound the performance of LLM inference due to the limited GPU memory capacity [12, 19].

Low-rank adaptation. Fine-tuning adapts a pre-trained LLM to a specific domain without training an LLM from scratch. Low-rank adaptation (LoRA) [8, 9, 21] is a popular class of parameter-efficient fine-tuning methods [10], as it can achieve competent performance by only fine-tuning a small number of trainable parameters, called the *adapter*. Furthermore, LoRA does not introduce extra inference latency. Inspired by the phenomenon of low "intrinsic rank" of weight updates, the core of LoRA is to represent each weight update as two rank composition matrices with much smaller ranks. During fine-tuning, LoRA only needs to optimize these two rank composition matrices, while keeping the pre-trained weights frozen. Figure 1 shows an example. For a pre-trained weight W with the shape of $d \times d$, LoRA represents weight updates ΔW as two smaller matrices A and B with the shape of $r \times d$ and $d \times r$ respectively, where r is much smaller than d. During fine-tuning, as shown in Figure 1(a), LoRA only updates A and B and keeps W frozen, which significantly reduces computation and memory consumption. At the inference time, as shown in Figure 1(b), LoRA can merge the multiplied matrix $B \times A$ (i.e., ΔW) into W to eliminate extra inference overhead. Given the benefits of LoRA, it is widely adopted to enhance the capability of LLMs, such as long sequence [22] and multimodal input [23]. It can be used in all dense layers of LLMs, but it is typically used to adapt attention weights [8].

2.2 Inference Serving Systems

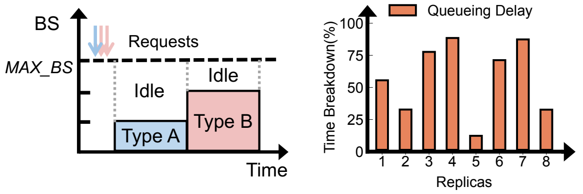
LLM serving systems. Many techniques have been developed to improve the efficiency of LLM inference by leveraging the characteristics of LLMs. Orca [11] proposes iteration-level scheduling to batch requests of different lengths at the granularity of iterations. Completed requests can be removed immediately and newly arrived requests can be inserted into the batch without waiting for the completion of the current batch. The state-of-the-art solution vLLM [12] further introduces an on-demand block-based GPU allocation mechanism called PagedAttention to reduce GPU memory fragmentation caused by variable and unpredictable KV cache, thereby increasing the maximum batch size. However, they only focus on the single LLM serving scenario. When serving multiple LoRA LLMs, they cannot share the common base model and thus cause severe redundant memory consumption.

HuggingFace PEFT [10] is a popular library for parameter-efficient fine-tuning. It can also be used to serve multiple LoRA LLMs shared with the same base LLM. However, it can only serve requests destined to the same adapter once at a time by swapping between different adapters, leading to low efficiency. Besides, it lacks support for cluster-level management for requests and adapters.

Traditional DNN serving systems. Many DNN serving systems can orchestrate multiple DNN models in a cluster, such as SHEPHERD [13] and AlpaServe [14]. However, they also do not support sharing the base model among different models and do not target autoregressive LLMs. PetS [24] can serve multiple parameter-efficient non-autoregressive transformer models in a single server, but it cannot serve autoregressive LLMs and does not consider LoRA. Besides, it does not support cluster-level management for requests and models. In short, existing DNN serving systems also cannot serve multiple LoRA LLMs in the cluster wide efficiently.

2.3 Challenges

To serve a large number of requests, a serving system usually deploys multiple replicas of the same base LLM, with each handling a subset of the requests. Nevertheless, when employing existing systems (such as vLLM and PEFT) to serve LoRA LLMs, we identify two primary challenges.



(a) Challenge within replicas. (b) Challenge across replicas.

Figure 2: Challenges in existing LoRA serving.

GPU underutilization within a replica. In a model replica, a single base model is accompanied by multiple LoRA adapters for different types of requests. The aforementioned PEFT only accommodates batching requests with the same LoRA adapter (i.e., the same type of requests). When serving one type of request, PEFT forces other types of requests to wait until the completion of the current batch. The serving system has to handle low-frequency types of requests one by one, which cause severe GPU underutilization. Figure 2(a) illustrates an example. When three requests arrive simultaneously, even if the max batch size is three, PEFT has to process these three requests separately: one batch for a request destined to the type A adapter and another batch for two requests destined for the type B adapter. As a result, the serving system only utilizes 50% of the total GPU resources and doubles the total latency for the requests destined to the type B adapter. This example indicates that although LoRA does not introduce extra inference latency for a single request, it is still challenging to serve multiple LoRA LLMs efficiently.

Load imbalance across replicas. To manage multiple replicas in a cluster, a serving system usually adopts a global scheduler to dispatch each incoming request to a specific replica. However, there exists load imbalance across replicas from two aspects. First, due to the limited GPU memory, one adapter type may only reside in a subset of replicas. When a burst of requests destined for this adapter type arrives, only a few replicas are utilized, while other replicas are idle. Figure 2(b) shows an example that sending requests based on the Azure trace [25] adopted by a previous DNN serving work [14] to a cluster with eight replicas, where 32 types of adapters are uniformly loaded in the eight replicas. In this case, the burst of requests leads to severe load imbalance across replicas. The difference in queuing delay between replicas can be up to 8.0×. Second, even if the requests are dispatched uniformly across replicas, the variable input and output lengths of requests still lead to load imbalance inevitably. The statistics of ShareGPT [26], the datasets collected from real-world conversations with ChatGPT [27], show that the input and output lengths of requests are highly variable. The longest input length and output length of requests are longer than the average lengths by 636.7× and 163.9× respectively, which implies extremely diverse execution time and GPU memory consumption among requests. The load imbalance caused by variable input and output lengths undermines the system's

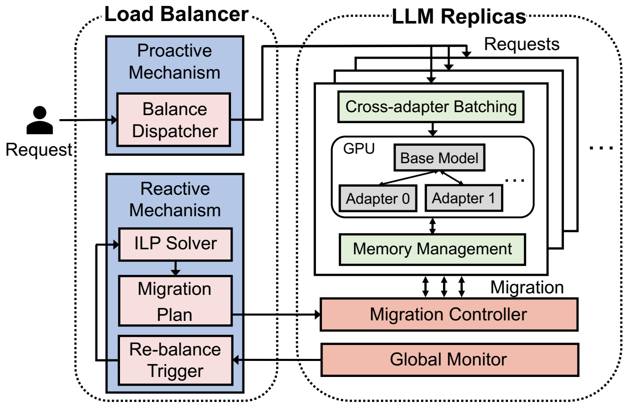


Figure 3: dLoRA architecture.

overall inference efficiency and significantly increases latencies for requests on the overloaded replica.

3 Overview

dLoRA is an inference serving system that serves multiple LoRA models in a cluster. Within a replica, dLoRA employs a novel cross-adapter batching technique to process requests to different LoRA adapters in a single batch and improves the GPU utilization (§4). Across replicas, dLoRA dynamically migrates LoRA adapters and requests in the cluster to achieve better load balancing (§5). Figure 3 shows the overall architecture of dLoRA.

Intra-replica. dLoRA deploys a set of worker replicas in a cluster. Each replica contains a subset of LoRA adapters and one base model on several GPUs.

Dynamic batching. Within a replica, dLoRA uses a local *cross-adapter batching* technique to process requests from the global scheduler. The replica maintains a queue to buffer incoming requests and schedules a batch of requests to the execution engine with dynamic batching to achieve optimal tradeoff between merged and unmerged inference.

Memory management. dLoRA efficiently manages the GPU memory of LoRA adapters and requests within a single replica. The memory manager allocates the GPU memory to the LoRA adapters and the intermediate states of requests. The two different types of memory may race for the limited GPU memory, which harms the serving performance. To mitigate this, the memory manager dynamically adjusts the memory allocation for adapters and requests based on the current GPU memory usage and workload pattern. Besides, the memory manager also swaps the unused adapters and requests to the host memory.

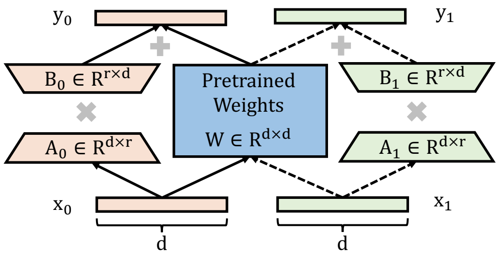


Figure 4: Unmerged inference.

Inter-replica. To manage multiple replicas in the cluster, dLoRA uses a load balancer to solve the aforementioned load imbalance problem. Due to the variable input and output lengths of LLM requests, dLoRA introduces proactive and reactive mechanisms to solve the imbalance problem before and after it occurs, respectively.

Proactive mechanism. The proactive mechanism, upon receiving user requests, directs them to a specific replica based on a two-fold dispatching policy. For short-term load dynamics, the mechanism proactively selects the replica with the most available resources to load the corresponding adapter and process the request. For long-term load dynamics, the mechanism proactively loads and replicates adapters for future predictable load spikes.

Reactive mechanism. The proactive mechanism alone is not sufficient since the resource usage (i.e., input and output length) of a request is variable. The load imbalance problem still occurs. To address this issue, dLoRA introduces a reactive mechanism to handle such a situation. Specifically, there is a global monitor that periodically collects the resource usage of each replica. Once the monitor detects a replica with a heavy load, it notifies the re-balance trigger. The reactive mechanism then employs a request-adapter co-migration algorithm to find the optimal migration plan and sends the plan to the cluster's migration controller. The controller then migrates requests' intermediate states and loads LoRA adapters across different replicas to achieve load balancing.

4 Dynamic Batching

4.1 Unmerged Inference

In §2.3, we discuss the limitations of merged inference in terms of significant queuing delay and poor GPU utilization when serving multiple LoRA LLMs. However, the characteristics of LoRA LLMs provides an opportunity.

Unmerged inference. Figure 1(b) shows that the fine-tuned weights and the pre-trained weights are merged to serve a request (i.e., $y = (W + BA)x$), as initially suggested in the LoRA paper [8]. This approach is termed *merged inference*. An alternative approach is *unmerged inference* to process requests with different types in a single batch. Specifically, unmerged inference separates the computation of the pre-trained LLM weights and each LoRA adapter weights. During inference, different types of requests can be batched together to share the same computation with the pre-trained LLM

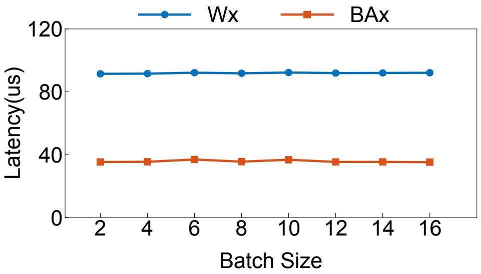


Figure 5: Non-negligible adapter computation overhead.

weights and process different computation with LoRA adapter weights at parallel.

Figure 4 demonstrates unmerged inference. Consider two different requests, x_0 and x_1, with their respective LoRA adapters, $adapter_0$ and $adapter_1$. During inference, the base model inference Wx is batched together as $W \times [x_0, x_1]$. Simultaneously, the LoRA adapter inference BAx is computed separately for each request as $B_0A_0x_0$ and $B_1A_1x_1$ in parallel. Finally, the base model inference and the LoRA adapter inference are aggregated to obtain the final results $y_0 = Wx_0 + B_0A_0x_0$ and $y_1 = Wx_1 + B_1A_1x_1$. Unmerged inference improves the GPU computation efficiency by improving the batch size. However, this solution poses a new challenge.

Extra Computation Overhead. Although unmerged inference accelerates the requests processing with different types, the benefit of unmerged inference technique is not a free lunch. Unmerged inference requires computing three matrix multiplications for each request, i.e., Wx, Ax and BAx, and a matrix addition for result aggregation. As a result, unmerged inference introduces extra computation overhead of two additional matrix multiplications (i.e., separate computation for adapter inference) and one additional matrix addition in each layer.

Figure 5 illustrates the overhead associated with unmerged inference. We conduct an experiment to compare the execution time of the base LLM computation, denoted as Wx (equivalent to original LoRA LLM inference), with the LoRA adapter computation BAx. The experimental setup follows §7. The results, depicted in the figure, reveal that the execution time for the LoRA adapter computation BAx is 38.9% of the baseline LLM computation Wx. This finding suggests that unmerged inference might not always yield performance benefits. On the contrary, performance may degrades when only processing few types of requests.

4.2 Dynamic Batching with Thresholds Tuning

In the preceding discussion, we highlight how unmerged inference incurs additional computational overhead, while merged inference leads to significant queuing delays and low GPU efficiency. A combined approach of these two batching methods can potentially offer a more balanced tradeoff between queuing delay and computational overhead. Yet, determining the optimal batching strategy, within a constrained scheduling time frame, is challenging. First, the decision needs to be made at the granularity of the iteration while each iteration of LLM inference is typically around tens or hundreds of milliseconds [11, 19]. To avoid the performance degradation, the decision must be made swiftly. Second, the unpredictability of request arrival patterns and request execution time further complicates this decision-making [13, 14, 26]. Last but not least, switching between the two inference methods introduces non-negligible overhead (i.e., an additional matrix multiplication BA and a matrix addition/subtraction between base LLM and adapter). We propose a dynamic batching technique to choose the inference method at runtime.

Algorithm 1 Dynamic Batching

```
 1: function DYNAMICBATCHING($B_{fcfs}, R, S, L$)
 2:     Input: FCFS requests $B_{fcfs}$, Request $R = \{r_1, r_2, ..., r_n\}$
 3:            Replica state $S$, LoRA adapters $L = \{l_1, l_2, ..., l_m\}$
 4:     Output: The batch of requests to be executed $B_{next}$
 5:     // Adaptive switching between different modes
 6:     if $S.state ==$ unmerge then
 7:         $R_{merge} = \arg\max_{l_i \in L} |\{r_i \in R \mid r_i.type == l_i\}|$
 8:         if $|R_{merge}|/|B_{fcfs}| > \alpha_{switch}$ then
 9:             $S.state$, $S.type =$ merge, $R_{merge}.type$
10:             return $B_{next} = R_{merge}[: max\_bs]$
11:         else
12:             return $B_{next} = B_{fcfs}$
13:     else
14:         $R_{merge} = \{r_i \in R \mid r_i.type == S.type\}|$
15:         if $|R_{merge}|/|B_{fcfs}| < \beta_{switch}$ then
16:             $S.state =$ unmerge
17:             return $B_{next} = B_{fcfs}$
18:         else
19:             return $B_{next} = R_{merge}[: max\_bs]$
```

Algorithm. Algorithm 1 outlines the pseudo-code. Each iteration begins by assessing the replica's state (line 6). In the unmerged state, the algorithm estimates potential performance gains from merged inference. It selects the LoRA adapter with the most requests (line 7) and evaluates its performance against the default first-come-first-serve (FCFS) order (line 8). We use the (FCFS) as the default scheduling choice. Dynamic batching is orthogonal to the underlying scheduling policy; other policies can also be used. If the size ratio exceeds the switching threshold α_{switch}, indicating a performance enhancement, the algorithm switches to merged inference (line 9) and processes the requests from the corresponding adapter type (line 10). If not, it continues with the FCFS batch (lines 11–12). In scenarios of merged inference, the algorithm first batches requests matching the active adapter, i.e., R_{merge} (line 14). If the FCFS batch size ratio to R_{merge} is smaller than the threshold β_{switch} or the active LoRA adapter is not in the set of legal adapters L (line 15), the unmerged state (i.e., processing B_{fcfs}) is deemed advantageous (lines 16–17). Otherwise, the algorithm remains the merged state (lines 19).

Adaptive threshold tuning. The switching threshold α_{switch} and β_{switch} are the key parameters of the algorithm. Figure 6

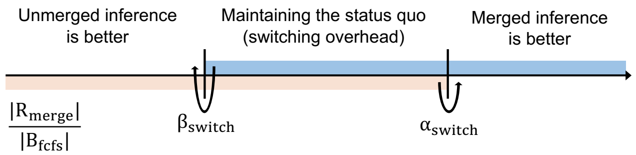

Unmerged inference is better
Maintaining the status quo (switching overhead)
Merged inference is better
$\frac{|R_{merge}|}{|B_{fcfs}|}$
β_{switch}
α_{switch}

Figure 6: Thresholds demonstration.

illustrates the two thresholds between merged and unmerged inference. The horizontal axis represents the ratio between $|R_{merge}|$ and $|B_{fcfs}|$.

Our first insight is that the switching overhead can be amortized across multiple future iterations. For example, assume that the switching overhead from unmerged to merged inference is 100 ms, and assume that merged inference of one iteration is 50 ms and unmerged inference of one iteration is 100 ms. Let the number of iterations in current processing is three, and the overall execution time is 300 ms if the status is unmerged inference. However, if we switch to merged inference at the beginning, the overall execution time is 250 ms for three iterations, i.e., switching overhead 100ms plus three iterations of 150 ms. In this case, the switching overhead is amortized across the three iterations. Our second insight is that leveraging historical retrospection is possible, despite the unavailability of future knowledge. Continuing with the example, at the beginning of the first iteration, the total number of iterations remains unknown. If requests are finished in one iteration, merging inferences offers no advantage. However, if processing spans two or more iterations, it becomes plausible to anticipate additional future iterations. Under such circumstance, switching to merged inference is still timely.

Based on the two insights, we propose an adaptive threshold tuning algorithm. The algorithm first introduces the break point on iteration granularity, which is marked by events such as replica switching, changes in R_{merge}, or after processing a set number of iterations. Upon reaching a break point, the algorithm tunes the thresholds based on the data collected from the preceding period, stretching from the current to the previous break point. The pseudo-code is outlined in Algorithm 2. In cases where the previous period's replica state is *unmerge*, we focus on tuning α_{switch}. Let the number of the iterations in the previous period be N_I and the merged requests, $R_{merged}[: max_{bs}]$, in the i_{th} iteration be B_i and B_{fcfs} in the i_{th} iteration be B_i'. The throughput of the merged inference T_{merge} and the throughput of the unmerged inference $T_{unmerge}$ are calculated in lines 4–5. The iteration time can be profiled accurately [19]. A higher T_{merge} leads to a reduction in α_{switch} by a decrement factor γ_{dec}. Otherwise, it is increased by a multiplication factor γ_{mul}. If the replica state in the previous period is *merge*, the algorithm tries to tune β_{switch}. The tuning is similar to the case of α_{switch}, and we omit the details for brevity. This method ensures the adaptability of the two thresholds to varying request characteristics.

Algorithm 2 Adaptive Threshold Tuning

```
1: Input: Candidate period $N_I$, Merged batches $B_1, B_2, ..., B_I$,
   Switching overhead $t_M$, Current switching threshold $\alpha_{switch}$
2: Output: New switching threshold $\alpha_{switch}$
3: function ADAPTIVETUNING($N_I$, $\{B_i\}$, $t_M$, $\alpha_{switch}$)
4:     $T_{merge} = \frac{\sum_{i=1}^{N_I} |B_i|}{\sum_{i=1}^{N_I} IterationTime(B_i) + t_M}$
5:     $T_{unmerge} = \frac{\sum_{i=1}^{N_I} |B_i'|}{\sum_{i=1}^{N_I} IterationTime(B_i')}$
6:     if $T_{merge} > T_{unmerge}$ then
7:         $\alpha_{switch} = \alpha_{switch} - \gamma_{dec}$
8:     else
9:         $\alpha_{switch} = \alpha_{switch} \times \gamma_{mul}$
10:    return $\alpha_{switch}$
```

Starvation prevention. Although dynamic batching accelerates LoRA LLM serving, it may cause starvation. Specifically, because dynamic batching prefers processing requests with the most loaded LoRA adapter type, it may starve other types of requests. To address this problem, we use a credit-based mechanism to prevent starvation. The basic idea involves allocating a credit to each LoRA adapter. This credit is then transferred to any preempted adapter. When the credits of certain adapters exceed a threshold, the algorithm prioritizes processing requests with these adapters. Detailed explanations are provided in §A.1.

4.3 LoRA Adapter Offloading

Besides GPU computational resources, GPU memory resources are also heavily used by LLM inference workloads. Sharing a base LLM across LoRA LLMs mitigates the GPU memory footprint, yet memory scarcity persists, especially when storing multiple LoRA adapters. For instance, a single LoRA adapter for Llama-7B requires 56 MB of GPU memory, which is comparable to the intermediate states (i.e., key-value cache) of 7 tokens. As the number of LoRA adapters increases, only a small fraction of GPU memory is available for storing intermediate states of requests. To this end, we employ a swapping mechanism that swaps LoRA adapters and intermediate request states between GPU and host memory. The compact size of LoRA adapter facilitates rapid swapping. Such process is further accelerated by overlapping the swapping with execution using prefetching techniques [19]. Regarding GPU memory distribution, dLoRA adopts a workload-aware allocation algorithm for balancing LoRA adapters and intermediate request states, as detailed in §5.

5 Dynamic Load Balancing

To address the load imbalance problem across replicas for LoRA serving, we combine proactive and reactive mechanisms. Specifically, we follow the existing approach [13, 14] to adopt a proactive mechanism that dispatches requests to replicas. The challenge of LoRA serving as discussed in §2.3 is that the variable input and output lengths of LLM requests cause load imbalance even if the proactive mechanism bal-

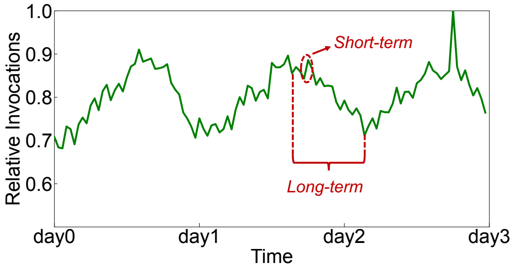


Figure 7: Workload pattern.

ances the load when dispatching requests. Therefore, we design a reactive mechanism that migrates requests and adapters between replicas to reactively address such load imbalance.

5.1 Proactive Mechanism

We first introduce the dispatcher that proactively dispatches requests to replicas and loads LoRA adapters to GPUs according to the workload patterns (i.e., the arrival pattern of requests). Figure 7 shows a production trace used by prior works on inference serving [13, 14]. The workload pattern of this trace can be examined from both long-term and short-term perspectives. In the long term, the pattern exhibits predictability and periodicity, e.g., low load at midnight and high load during daytime. On the other hand, in the short term, the pattern is marked by unpredictability and burstiness. We follow existing work [13, 14] to adopt a proactive mechanism based on these characteristics for LoRA serving.

Guided by the predictable long-term workload, it is possible to preload LoRA adapters to GPU memory to reduce the adapter loading time. During the adapter preload process, we follow the existing work [13] to maximize the minimum burst tolerance of LoRA adapters. The burst tolerance of an adapter in our scenario is defined as the ratio of the peak capacity to the average load of the adapter, where the average load can be obtained from the historical long-term workload. Different from the previous DNN serving scenarios, we find that the peak capacity of a LoRA adapter is dominated by the GPU memory allocation between LoRA adapters and requests' intermediate states. As more LoRA adapter is preloaded into the GPU memory, more replicas can serve this type of request immediately. However, it leaves less GPU memory for requests' intermediate states, which limits the peak serving throughput. Therefore, we define the peak capacity of a LoRA adapter as the number of requests that can be served by the unallocated GPU memory where the corresponding LoRA adapter resides. Based on this metric, the dispatcher of dLoRA greedily preloads the LoRA adapter with the lowest burst tolerance to a replica without this adapter until the minimum burst tolerance decreases to find an optimal placement plan.

Due to the unpredictable short-term pattern of the workload, there also exists a short-term burst of certain LoRA adapters' requests. Such short bursts may cause the burst requests to be dispatched to a small set of replicas with the corresponding LoRA adapters. This leads to severe load imbalance and long queuing delay od requests. On the contrary, if the burst requests are dispatched to other replicas without the corresponding adapter, it incurs additional loading time and consequently harms the serving performance. To address this problem, dLoRA use an adapter-aware dispatch policy with dynamic LoRA loading to consider all these factors. Specifically, dLoRA calculates an estimated pending time for each replica, which includes the time to load the LoRA adapter (if not already loaded) and the estimated queuing time on this replica. Based on this metric, dLoRA dispatches requests to the replica with the shortest estimated pending time to balance the load across replicas. This is especially effective in mitigating issues caused by sudden bursts of requests and consequent queuing delays in heavily loaded replicas.

5.2 Reactive Migration

Although proactive dispatching can mitigate the load imbalance problem, the system still suffers from load imbalance caused by variable input and output lengths of LoRA requests (§2.1). Therefore, even if the workload arrival pattern is stable and entirely predictable, the load imbalance still occurs when some requests in some replicas have longer execution times than others. The GPU memory consumption of requests also suffers from the same load imbalance problem. Some replicas may hold substantial intermediate states of requests even larger than the GPU memory capacity, which causes frequent swapping between GPU memory and host memory. Other replicas may hold a smaller amount of intermediate states of requests. Such load imbalance harms the serving performance and cannot be solved by only proactively dispatching.

Dynamic adapter-request co-migration. To address the load imbalance problem, we propose a adapter-request co-migration technique. The main idea is to migrate LoRA adapters and requests (with intermediate states) from overloaded replicas to others. Such migration reactively balances the load across replicas. As shown in Figure 8, replica 0 is overloaded with long-context requests while replica i is underloaded with short-context requests. The load imbalance can be mitigated by migrating some requests from replica 0 to replica i and loading corresponding adapters. However, it is not trivial to decide the optimal migration plan. Before introducing the algorithm in detail, we first model the reactive migration problem. The key notations are listed in Table 1.

Objective: overall running time. The goal of the migration algorithm is to minimize the overall running time among all replicas. The overall running time is determined by the placement of requests and LoRA adapters. We define two 0-1 matrices x and y to represent the placement of requests and LoRA adapters, respectively. Specifically, $x_{i,j}$ and $y_{k,j}$ are 0-1 variables that indicate whether request i or adapter k is in the replica j. Furthermore, we break x into two matrices x^G and x^H to represent the placement of requests in the GPU memory and host memory, respectively. The request with its

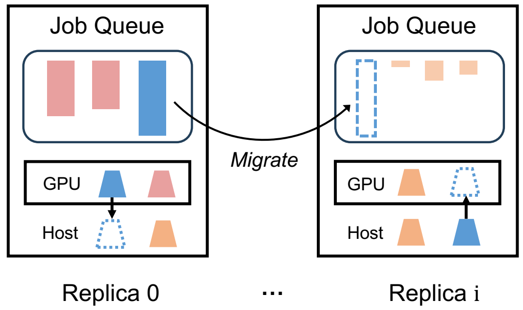


Figure 8: Dynamic migration.

Symbol	Description
$x^G_{i,j}$	Whether requests i is in the replica j's GPU memory.
$x^H_{i,j}$	Whether requests i is in the replica j's host memory.
$x_{i,j}$	Whether requests i is in the replica j.
$y_{k,j}$	Whether adapter k is in the replica j.
M^R_i	The memory consumption of request i.
M^A_k	The memory consumption of adapter k.
M^G_j	GPU memory capacity of replica j.
M^H_j	Host memory capacity of replica j.
L_i	Average exec time of requests with request i's adapter.
γ	Migration factor to control the request migration frequency.
η	Affinity factor to control the affinity between requests.

Table 1: Key notations in the migration problem.

intermediate results is either resident in the GPU memory or host memory (i.e., $x_{i,j} = x^G_{i,j} + x^H_{i,j} \leq 1$).

The overall running time of requests on each replica includes execution time and swapping overhead. On replica j, the execution time of request i is $L_i x_{i,j}$, where L_i is the average execution time of requests i. We scale the execution time by multiplying a migration factor γ to account for the migration time of request i. γ is calculated as the ratio of total executed time, including the migration time, to the total executed time without migration, inhibiting the frequent migration between replicas. As for swapping overhead, it is represented as $M^R_i x^H_{i,j}/B$, where M^R_i is the memory consumption and B is the PCIe bandwidth. As a result, the overall running time of replica j is estimated as $\sum_i \left(\gamma \times L_i x_{i,j} + M^R_i x^H_{i,j}/B\right)$. Because the dynamic batching algorithm prefers to batch requests with the same type, we also add a penalty term $\eta \sum_k y_{k,j}$ to represent the affinity among the same type of requests, where η is a hyperparameter. For all parallel replicas, the overall running time is the maximum running time of all replicas:

$$Min.\ \big(\max_j \sum_i (\gamma \times L_i x_{i,j} + M^R_i x^H_{i,j}/B) + \eta \sum_k y_{k,j}\big)$$

Adapter-request matching constraints. Different from classical load balancing problems [15–17], the adapter-request co-migration algorithm needs to consider both the migration of requests and the loading of LoRA adapters at the same time. One request can only be migrated to the replica only if there is a corresponding adapter available, which is formulated as following adapter-request matching constraints:

$$\forall i, j,\ y_{i,j} \geq x_{i,j}$$

Memory constraints. For each replica, the GPU memory consumption cannot exceed the GPU memory capacity. We define M^G_j as the GPU memory capacity and M^H_j as the host memory capacity of replica j. We get two constraints for GPU and Host memory, respectively:

$$\begin{cases} \forall j,\ \sum_i M^R_i \cdot x^G_{i,j} + \sum_k M^A_k \cdot y_{k,j} \leq M^G_j \\ \forall j,\ \sum_i M^R_i \cdot x^H_{i,j} \leq M^H_j \end{cases}$$

Existence constraints. Last but not least, the existence constraints ensure that each request is only placed in one replica, which is represented as:

$$\forall i,\ \sum_j x_{i,j} = 1$$

ILP formulation. Given the above objective and constraints, we formulate the co-migration problem as an ILP problem, where matrices x^G, x^H, and y are the variables. dLoRA solves the problem with the off-the-shelf ILP solver [28] and gets the optimal placement plan (i.e., x^G, x^H, and y). According to this plan, dLoRA migrates the intermediate states of requests and loads LoRA adapters simultaneously. dLoRA also reserves a chunk of memory in each replica, which is utilized to replicate additional LoRA adapters to maximize the burst tolerance as described in the proactive mechanism (§5.1). With this approach, dLoRA's co-migration algorithm effectively achieves load balancing across replicas with optimal performance.

Selective migration with constraints relaxation. The above ILP formulation offers an optimal migration plan, but its large decision space introduces computation overhead. As the GPU cluster and request number expand, the ILP's complexity increases exponentially. To address this, dLoRA employs two domain-specific heuristics to accelerate such solving process.

Our first insight is that migration is necessary primarily during extreme load imbalance. Therefore, dLoRA only triggers the migration algorithm when the available GPU memory of a replica is beyond a memory threshold or the queuing delay of requests in a replica is beyond a computation threshold. This approach effectively minimizes the migration frequency, thereby amortizing the migration overhead. Besides, similar to selective replication [15–17], dLoRA only considers migration between top K overloaded replicas and top K underloaded replicas. As a result, the complexity of the selective migration only depends on K rather than the cluster scale.

Our second insight is that the ILP problem can be simplified by relaxing the constraints. In practice, migration is implemented as token transfer and subsequent intermediate state reconstruction, which is much faster than direct intermediate state migration [12]. As a result, dLoRA does not differentiate the requests on the GPU memory and host memory. Instead, dLoRA only considers the total memory consumption of requests and LoRA adapters. We assume that additional memory consumption is swapped out to the host memory which is always sufficient in practice. Therefore, memory constraints are negligible, with only the swapping overhead added to the total execution time.

Last, to reduce the complexity of the ILP problem, the variable matrix x is relaxed to the real number. Given the real number matrix x, the placement of each request is determined by the largest element in the corresponding row of x. For instance, request i is placed in replica j' if $x_{i,j'} = \max_j x_{i,j}$. With these techniques, dLoRA is able to solve the optimal adapter-request co-migration plan within milliseconds.

6 Implementation

We implement a prototype of dLoRA based on vLLM [12] with about 6.2K LOC. We use vLLM to build dLoRA as vLLM is the state-of-the-art serving system with advanced features such as PagedAttention and iteration-level scheduling. The prototype includes a FastAPI [29] frontend, a global scheduler, and GPU-based execution engines. The frontend of dLoRA extends the vLLM FastAPI [29] frontend, enables client-specific LoRA adapter selection per request. dLoRA's global load balancer is responsible for dispatching requests to LLM replicas and dynamically migrating requests. It uses Ray [30] actor to interact with execution engines in the cluster and utilizes Pulp [28] to solve the ILP problem defined in §5. The execution engine also uses Ray [30] actor for key-value cache management and LoRA adapter management. To support *unmerged inference* in §4, we transform the LoRA type of each request into a one-hot vector and generate a request-type mapping matrix of the current batch. We then utilize the `einsum` function provided by PyTorch to achieve parallel matrix multiplication and integrate it into the execution engine. We add LoRA adapters to the vLLM execution engine and swap adapters between GPU and host memory asynchronously to reduce overhead. As for model executor, we implement LoRA inference for popular LLMs, including OPT [31] and Llama-2 [2]. dLoRA also accomplishes Megatron-LM [32] style tensor parallelism on LoRA adapters to support distributed execution of large models which can not fit in a single GPU, e.g., Llama-2-70B. dLoRA can be integrated with other frameworks, such as Ray Serve [33]. On the server side, similar to HuggingFace PEFT [10], the cluster administrator needs to provide additional LoRA information, including LoRA adapter weights, LoRA adapter name and its dependency on the base LLM, before launching the service. dLoRA takes charge of other things. When sending a request to dLoRA, the client side specifies the LoRA adapter name in the request. dLoRA dispatches the request to the corresponding LoRA adapter and returns the result.

Model	Size	# of Layers	# of Heads	Hidden Size
Llama-2-7B	13GB	32	32	4096
Llama-2-13B	26GB	40	40	5120
Llama-2-70B	132GB	80	64	8192

Table 2: Model configurations.

7 Evaluation

In this section, we first demonstrate the end-to-end performance improvements of dLoRA over state-of-the-art LLM serving systems under diverse workloads and models. Then, we evaluate the design choices of dLoRA and show the effectiveness of each component.

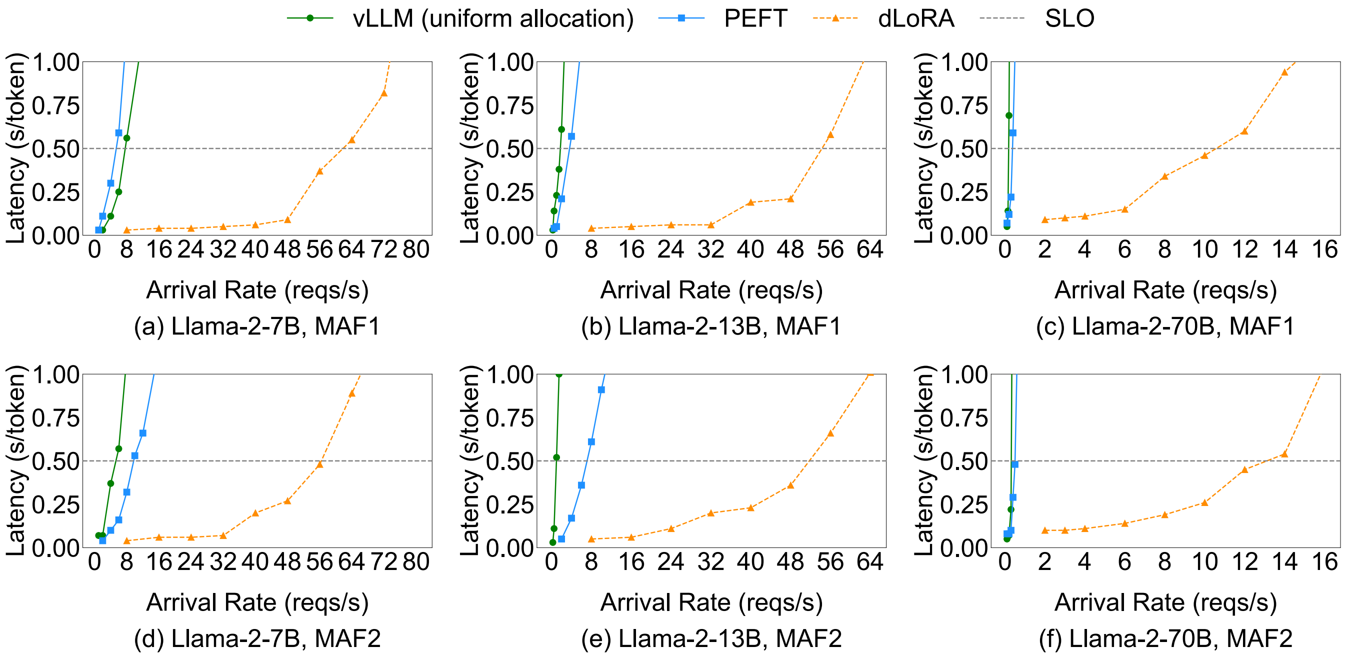
7.1 Experiment Setup

Testbed. We evaluate dLoRA on a four-node GPU cluster, each with eight NVIDIA A800 80GB GPUs, i.e., 32 GPUs in total. Each node is equipped with 128 CPUs, 2048 GB of host memory, and a 200 Gbps InfiniBand NIC. We use PyTorch 12.1.0 and NVIDIA CUDA 12.2 for our evaluation.

Models. We choose the widely-used open-sourced LLMs, Llama-2 model series [2], as the pre-trained LLMs (i.e., base LLMs) for our evaluation. We use various Llama-2 models with different sizes, including Llama-2-7B, Llama-2-13B, and Llama-2-70B. The details of these models are shown in Table 2. The rank of LoRA adapters is set to 8, as used in the evaluation of prior work [8, 9].

Workloads. Similar to prior work [12], we generate workloads based on the ShareGPT dataset [26]. ShareGPT is a dataset collected from real-world conversations with OpenAI ChatGPT shared by users. We sample the arrival pattern of requests based on the production traces, Microsoft Azure function trace 2019 (MAF1) [34] and 2021 (MAF2) [25]. Although these traces are collected from Azure serverless functions, they are also widely used as the proxy of the LLM inference traces by prior work on model serving [13, 14]. Because the number of functions is larger than that of LLMs, we sort the functions based on the function invocation frequencies and map the functions to LoRA LLMs in a round-robin manner. To demonstrate different load patterns, we define the skewness as the number of each round-robin mapping. For example, if we map the first 10 functions to the first LoRA LLM, the second 10 functions to the second LoRA LLM, and so on, the skewness is 10. The larger the skewness, the more skewed the workload. To illustrate the impact of increasingly longer requests [35], we also scale the input and output length of requests by a scale ratio factor in §7.4.

Metrics. We use the average latency as the primary metric to evaluate the performance of dLoRA. Following prior work on LLM serving [11, 12], the average latency is calculated by dividing the sum of each request's end-to-end latency by the total number of output tokens. To compare different systems,



(a) Llama-2-7B, MAF1
(b) Llama-2-13B, MAF1
(c) Llama-2-70B, MAF1
(d) Llama-2-7B, MAF2
(e) Llama-2-13B, MAF2
(f) Llama-2-70B, MAF2

Figure 9: Average latency of different serving systems with Llama-2 models.

we set a latency service level objective (SLO) and compare the maximum throughput achieved by each system under the SLO. Similar to prior work, we set the latency SLO to 10× of the latency of a single iteration in the decoding phase. Specifically, SLO is 0.5 seconds based on our profiling.

Baselines. Because there is no existing system that specifically targets LoRA LLM serving. We compare dLoRA with two state-of-the-art LLM serving systems.

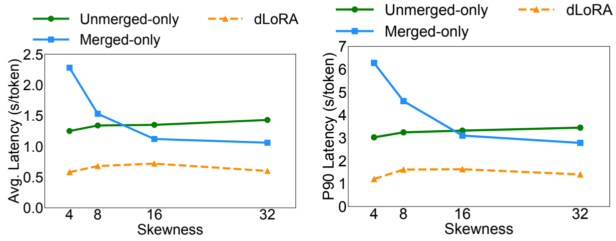
- **vLLM (uniform allocation)** [12]: vLLM is the state-of-the-art LLM serving system. It is a general-purpose LLM serving system, and ignores the multi-LLM serving scenario. To evaluate the performance of vLLM in the LoRA serving setting, we deploy multiple vLLM instances on the same cluster and uniformly allocate resources between them.
- **PEFT** [10]: PEFT is a HuggingFace library for parameter-efficient fine-tuning models. Although it can be used to serve LLMs, it does not support advanced features like selective batching [11] and PagedAttention [12]. To conduct a fair comparison, we implement these features for PEFT. PEFT batches requests based on their types and swap adapters between batches if necessary.

7.2 End-to-End Performance

We first compare end-to-end performance of dLoRA with vLLM (uniform allocation) and PEFT under MAF1 and MAF2 workload traces on Llama-2-7B, Llama-2-13B and Llama-2-70B models. The first column of Figure 9 shows the performance of these LLM serving systems when serving 128 Llama-2-7B models. Since Llama-7B is relatively small, the total GPU memory can accommodate full parameters of all models. However, vLLM (uniform allocation) and PEFT cannot dynamically batch different types of requests (i.e., unmerged inference) based on different workload patterns. In this case, dLoRA improves the throughput by up to 10.6× compared to vLLM (uniform allocation) and up to 11.5× compared to PEFT under the SLO requirement.

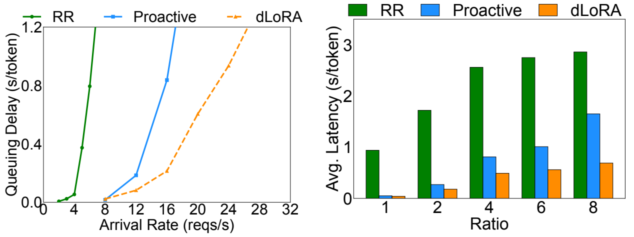
The second column of Figure 9 shows the performance of these LLM serving systems when serving 128 Llama-2-13B models. Since Llama-13B is larger than Llama-7B, the total GPU memory cannot accommodate full parameters of all models. As a result, vLLM (uniform allocation) which treats each LoRA LLM as an individual LLM has to swap the model parameters between host and GPU memory. Thus, dLoRA improves the throughput by up to 53.0× compared to vLLM (uniform allocation) under the SLO requirement. PEFT shares the base LLM between different LoRA LLMs to reduce memory footprint, but it does not have adapter-request migration. This significantly degrades the performance of PEFT. As a result, dLoRA improves the throughput by up to 15.0× compared to PEFT under the SLO requirement.

The third column of Figure 9 shows the performance improvement of dLoRA when serving 32 Llama-2-70B models in the cluster. Llama-2-70B is even larger than the GPU memory capacity of a single NVIDIA A800 80GB GPU. Therefore, the Llama-2-70B is partitioned across 4 GPUs with the tensor parallelism. As shown in Figure 9, the techniques employed by dLoRA integrate effectively with existing parallelism strategies and dLoRA improves the throughput by up



(a) Average Latency. (b) P90 Latency.

Figure 10: Effectiveness of the credit-based dynamic batching algorithm.



(a) Reduction in Queuing Delay. (b) Stability under Different Ratios.

Figure 11: Effectiveness of the adapter-request co-migration algorithm.

to 57.9× compared to vLLM (uniform allocation) and up to 26.0× compared to PEFT under the SLO requirement.

7.3 Effectiveness of Dynamic Batching

To show the effectiveness of dynamic batching with thresholds tuning, we compare the performance of dLoRA with strawman solutions described in §4. The experiments are conducted in a single NVIDIA A800 80GB GPU to serve 8 LoRA Llama-2-7B models, which avoids the effect of the adapter-request co-migration algorithm.

Figure 10(a) shows the average latency under diverse skewness. The arrival rate is set to make the average latency of dLoRA approximately equal to the SLO requirement. The first strawman solution, Merged-only, which always uses merged inference, performs poorly when the skewness is low, i.e., the type of requests is diverse, since it cannot serve different requests at the same time. Therefore, dLoRA improves the latency by up to 3.9×. The other strawman solution, Unmerged-only, which always uses unmerged inference, performs similarly no matter how the skewness changes. However, when the skewness is high, i.e., a few types of requests are dominant, Unmerged-only cannot take advantages of merged inference to avoid extra computation overhead caused by the adapter computation. Consequently, dLoRA improves the latency by up to 2.4× compared to Unmerged-only. In short, dLoRA dynamically switches between merged and unmerged inference based on the runtime workload, and always outperforms the two strawman solutions and achieves the optimal tradeoff.

In addition to the average latency, we also record the P90 latency of the three solutions with the same setting. As shown in Figure 10(b), dLoRA outperforms Merged-only and Unmerged-only by up to 5.2× and 2.5×, respectively. This consistent performance improvement indicates that although dLoRA may preempt some requests to serve other requests, the credit-based starvation prevention mechanism of dynamic batching improve performance while avoiding starvation.

7.4 Effectiveness of Dynamic Load Balancing

To show the effectiveness of the proactive and reactive dynamic load balancing, we compare the performance of dLoRA with two strawman solutions. We measure the average queuing delay of requests, an important indicator of load imbalance. Due to the limited budget, this experiment is conducted on a server with 8 NVIDIA A800 80GB GPUs. Figure 11(a) shows the experiment result. The first strawman solution is *RR* that directly dispatches requests to the corresponding LoRA LLMs preloaded by the workload-aware adapter placement in a round-robin manner without considering the bursty load imbalance (§5). Since RR does not consider the bursty load imbalance at all, dLoRA outperforms RR by 3.6× under the SLO requirement. The second strawman solution is *Proactive Dispatch* which only supports proactive mechanism in §5.1 without reactive migration. However, due to the variable and unpredictable length of requests, Proactive Dispatch cannot handle this unpredictable run-time load imbalance. In contrast, dLoRA dynamically migrates requests between replicas to balance the load and outperforms Proactive Dispatch by 1.4× under the SLO requirement.

We also evaluate the stability of the dynamic load balancing algorithm under different scale ratio factors of input and output length of requests. Figure 11(b) shows the average latency of requests under different ratios. Because RR cannot dynamically change LoRA adapter distribution across replicas as the ratio increases, the average latency of requests increases rapidly, and thus the average latency of RR is up to 23.5× higher than dLoRA. Although Proactive Dispatch Only is able to dynamically load LoRA adapters and dispatch the requests to mitigate load imbalance (i.e., it outperforms RR by up to 6.5×), it is not able to migrate requests between replicas to handle increasingly variable requests, leading to unpredictable load imbalance. As a result, the average latency of Proactive Dispatch is up to 2.39× higher than dLoRA.

7.5 Scalability and Overhead

For a cluster-scale serving system, we conduct evaluations to analyze the scalability of dLoRA. Figure 12 shows the throughput of dLoRA under different numbers of adapters. We increase the number of adapters increases while keeping the arrival rate constant. As the number of adapters increase, dLoRA achieves stable throughput. However, vLLM (uniform allocation) faces scalability challenges because it has to main-

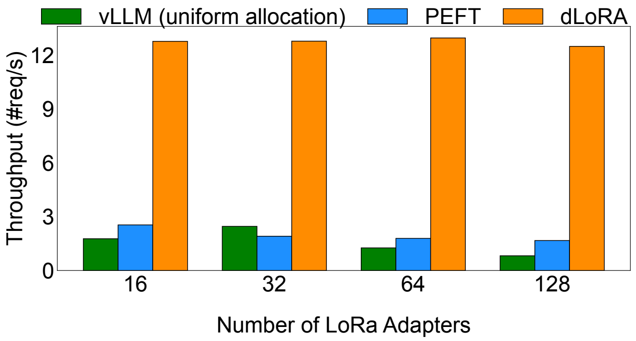


Figure 12: Scalability of dLoRA.

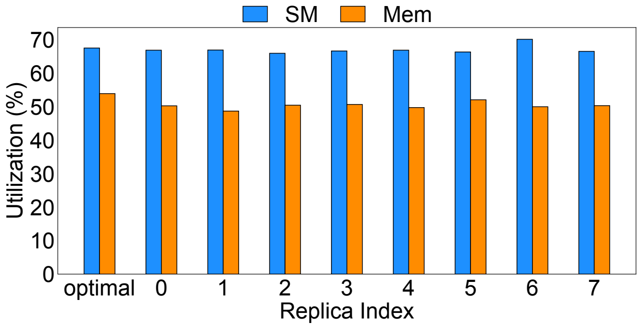


Figure 13: GPU utilization of dLoRA.

tain separate instances of full LLM weights for each LoRA LLM whcih incurs high memory footprint. PEFT struggles to scale efficiently as well because it cannot dynamically batch different types of requests. Moreover, the two baselines cannot manage the load imbalance between replicas. As a result, the throughput of vLLM (uniform allocation) and PEFT decreases by up to 3.0× and 1.5× respectively, as the number of adapters increases to 128.

We also evaluate the runtime overhead of dLoRA. Figure 15 breaks down the total latency of dLoRA's inference into three parts: the solving time of the ILP solver for the dynamic co-migration algorithm, the switching overhead of the dynamic batching algorithm, and the actual inference time. As shown in this figure, the overhead introcduced by dLoRA, i.e., ILP solving time and the switching overhead, are negligible compared to the actual inference time. The actual inference time consistently accounts for over 96.7% of the total latency, regardless of the arrival rate. The negligible runtime overhead of dLoRA mainly comes from the fact that dLoRA migrates requests or switches inference mode occasionally and the overhead is amortized by a number of iterations.

7.6 GPU Utilization

To show the GPU utilization, we measure the streaming multiprocessor (SM) utilization and GPU memory utilization of each replica in the cluster.The setup is the same as §7.4. dLoRA serves the Llama-2-13B model. The request rate is set to make the average latency of dLoRA approximately equal to the SLO requirement. We also measure the optimal SM utilization and GPU memory utilization when all requests belong to the same type, and vLLM always uses merged inference. As shown in Figure 13, although requests belong to different LoRA types, dLoRA achieves nearly the same SM utilization and GPU memory utilization compared to the optimal utilization, which indicates that dLoRA effectively utilizes the GPU resources and avoids resource waste as shown in Figure 2(a). Besides, the SM utilization and GPU memory utilization among replicas are balanced, which shows the effectiveness of the dynamic load balancing algorithm.

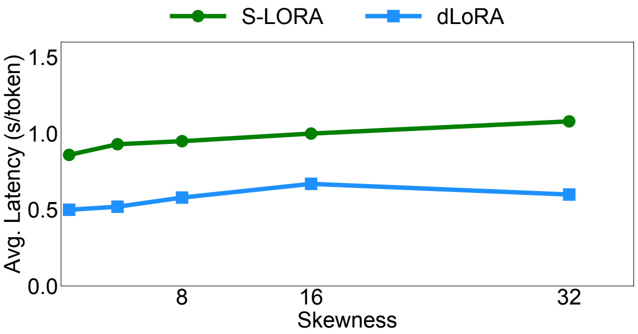


Figure 14: Comparison with SLoRA.

7.7 Comparison with Concurrent Work

S-LoRA [18] is a concurrent work for serving multiple LoRA LLMs. S-LORA also tries to batch requests destined for different LoRA adapters. However, S-LoRA does not consider the load imbalance between replicas and is not able to dynamically switch between merged and unmerged inference. The parallelism strategy proposed by S-LoRA is also orthogonal to dLoRA. Figure 14 compares dLoRA with S-LoRA using the same setting as §7.3. As shown in the figure, no matter how the skewness changes, dLoRA consistently outperforms S-LoRA by up to 1.8× in terms of average latency. The reason is that S-LoRA statically serves requests by unmerged-only inference, which ignores the opportunity of using merged inference to reduce the computation overhead. In contrast, dLoRA exploits this opportunity and is more effective in handling diverse requests with different LoRA types.

8 Related Work

LLM serving systems. Recently, many works have been proposed for LLM serving. Orca [11] proposes iteration-level scheduling to continuously batch requests with different lengths at the iteration level. vLLM [12] proposes a PagedAttention operator and a block-based KV cache management mechanism to reduce GPU memory fragmentation. FastServe [19], DeepSpeed-FastGen [36], and SARATHI [37] leverage the characteristics of LLM serving to schedule requests with different lengths. Nvidia FasterTransformer [38],

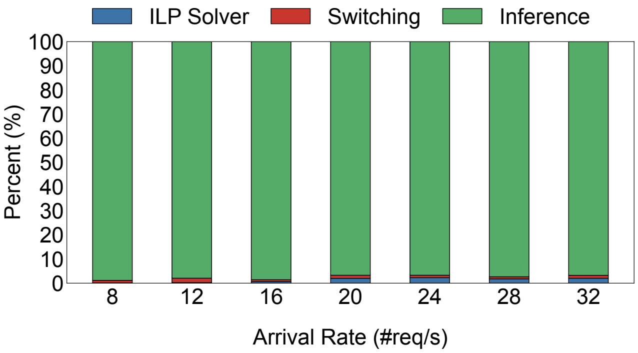


Figure 15: Overhead of dLoRA.

DeepSpeed Inference [39], and the Google serving system for PaLM [40] optimize the GPU/TPU implementation and parallelism specifically for LLM inference. FlexGen [41] uses offloading to improve the throughput of LLM serving. SpecInfer [42] uses speculative decoding to reduce the latency of LLM serving systems. SpotServe [43] tries to leverage spot instances to reduce the cost of LLM serving. These works focus on the optimization of a single LLM. In contrast, dLoRA is a serving system for multiple LoRA LLMs in a GPU cluster.

S-LoRA [18] and Punica [44] are concurrent works that also propose to serve multiple LoRA LLMs by batching requests destined for different adapters. S-LoRA proposes a new parallelism strategy, which is orthogonal to dLoRA. Punica uses an ad-hoc migration strategy to reduce the number of used GPUs, which is also different from dLoRA. dLoRA proposes dynamic load balancing algorithms to balance the load among GPUs, which is not considered in Punica and S-LoRA. They also overlook the opportunity of using merged inference to further reduce the latency of LLM serving by only adopting a merged-only inference strategy.

Traditional DNN serving systems. Many production-ready DNN serving systems have been developed, such as Tensorflow Serving [45] and Triton Inference Server [46], but they do not have LLM-specific optimizations. Some recent DNN serving systems, such as Clipper [47], ClockWork [48], SHEPHERD [13], Tabi [49], and Paella [50] serves multiple DNN models in cluster wide, but they mainly focus on small DNN models, such as ResNet and BERT. AlpaServe [14] leverages diverse parallelism strategies to accelerate serving multiple large DNN models in a GPU cluster, but it does not target autoregressive LLM serving and LoRA models. PetS [24] considers the scenario of serving multiple parameter-efficient DNN models in a GPU server, but it does not consider serving autoregressive LLMs in a GPU cluster and the unique system characteristics of LoRA adapters. DVABatch [51] uses multi-entry multi-exit batching to serve diverse models simultaneously, but it does not target either LLMs or LoRA.

Load balancing. Load balancing has been studied in many research fields, such as networking and cloud computing. Many load balancers, such as Ananta [52], Beamer [53] and Maglev [54], try to improve the performance of dispatching packets, but as discussed above, it is not sufficient to solve load imbalance in our scenario. Other kinds of load balancers, such as Pegasus [15], Scarlett [16] and E-Store [55], adopt selective replication or migration to balance the load. However, they are not suitable for our scenario, and the dependency between requests and adapters makes the load imbalance in our scenario even more complicated.

9 Conclusion

We present dLoRA, an inference serving system for LoRA models. dLoRA dynamically orchestrates requests and adapters for efficient LLM serving in each worker replica and across worker replicas. For each replica, dLoRA employs a dynamic batching technique to leverage both merged and unmerged inference to improve the efficiency of LLM serving. For the cluster, dLoRA employs a dynamic load balancing technique to migrate both requests and adapters to balance the load among worker replicas. Based on these techniques, we build a system prototype of dLoRA. Evaluation on production traces shows that dLoRA achieves up to 57.9× and 26.0× higher throughput than vLLM and HugginFace PEFT. dLoRA also achieves up to 1.8× lower average latency than the concurrent work S-LoRA.

Acknowledgments. We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback. This work was supported by the National Natural Science Foundation of China under the grant number 62325201, 62172008, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xuanzhe Liu is the corresponding author. Bingyang Wu, Ruidong Zhu, Zili Zhang, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] OpenAI, "GPT-4 technical report," 2023.

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv*, 2023.

[3] B. Roziére, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," *arXiv*, 2023.

[4] "OpenAI Documentation: Fine-tuning." https://platform.openai.com/docs/guides/fine-tuning, 2023.

[5] "Announcing Anyscale Private Endpoints and Anyscale Endpoints Fine-tuning." https://www.anyscale.com/blog/announcing-anyscale-private-endpoints-and-anyscale-endpoints-fine-tuning, 2023.

[6] "Customize a model with Azure OpenAI Service." https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning?tabs=turbo%2Cpython&pivots=programming-language-studio, 2023.

[7] "Fine-Tuning Llama 2 with Together.ai: A Step-by-Step Guide." https://blog.gopenai.com/fine-tuning-llama-2-with-together-ai-a-step-by-step-guide-cf2f3cce659d, 2023.

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," in *ICLR*, 2022.

[9] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," *arXiv*, 2023.

[10] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods." https://github.com/huggingface/peft, 2022.

[11] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *USENIX OSDI*, 2022.

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *ACM SOSP*, 2023.

[13] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the Wild," in *USENIX NSDI*, 2023.

[14] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving," in *USENIX OSDI*, 2023.

[15] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories," in *USENIX OSDI*, 2020.

[16] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *EuroSys*, 2011.

[17] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *USENIX NSDI*, 2013.

[18] S. Ying, C. Shiyi, L. Dacheng, H. Coleman, L. Nicholas, Y. Shuo, C. Christopher, Z. Banghua, Z. Lianmin, K. Kurt, *et al.*, "S-LoRA: Serving thousands of concurrent lora adapters," *Conference on Machine Learning and Systems*, 2023.

[19] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models," in *arXiv*, 2023.

[20] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li, "LightSeq: A High Performance Inference Library for Transformers," in *NAACL*, 2021.

[21] A. Chavan, Z. Liu, D. Gupta, E. Xing, and Z. Shen, "One-for-All: Generalized LoRA for Parameter-Efficient Fine-tuning," *arXiv*, 2023.

[22] Y. Chen, S. Qian, H. Tang, X. Lai, Z. Liu, S. Han, and J. Jia, "LongLoRA: Efficient fine-tuning of long-context large language models," *arXiv*, 2023.

[23] T. Gong, C. Lyu, S. Zhang, Y. Wang, M. Zheng, Q. Zhao, K. Liu, W. Zhang, P. Luo, and K. Chen, "Multimodal-GPT: A vision and language model for dialogue with humans," *arXiv*, 2023.

[24] Z. Zhou, X. Wei, J. Zhang, and G. Sun, "PetS: A unified framework for Parameter-Efficient transformers serving," in *USENIX ATC*, 2022.

[25] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *ACM SOSP*, 2021.

[26] "ShareGPT Teams." https://sharegpt.com/, 2023.

[27] "Introducing ChatGPT." https://openai.com/blog/chatgpt, 2022.

[28] "PuLP: A Python Linear Programming API.." https://github.com/coin-or/pulp, 2009.

[29] "FastAPI Documentation.." https://fastapi.tiangolo.com/, 2018.

[30] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *USENIX OSDI*, 2018.

[31] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," *arXiv*, 2022.

[32] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv*, 2020.

[33] "Ray serve: Scalable and programmable serving." https://docs.ray.io/en/latest/serve/index.html, 2023.

[34] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *USENIX ATC*, 2020.

[35] "Introducing Claude 2.1." https://www.anthropic.com/index/claude-2-1, 2023.

[36] "DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference." https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen, 2023.

[37] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," 2023.

[38] N. Corporation, "FasterTransformer." https://github.com/NVIDIA/FasterTransformer, 2019.

[39] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale," in *SC*, 2022.

[40] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Conference on Machine Learning and Systems*, 2023.

[41] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput Generative Inference of Large Language Models with a Single GPU," *International Conference on Machine Learning (ICML)*, 2023.

[42] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi, C. Shi, Z. Chen, D. Arfeen, R. Abhyankar, and Z. Jia, "SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification," 2023.

[43] P. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "SpotServe: Serving Generative Large Language Models on Preemptible Instances," *ACM ASPLOS*, 2024.

[44] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-Tenant LoRA Serving," *Conference on Machine Learning and Systems*, 2023.

[45] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv*, 2017.

[46] N. Corporation, "Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.." https://github.com/triton-inference-server/server, 2019.

[47] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System.," in *USENIX NSDI*, 2017.

[48] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *USENIX OSDI*, 2020.

[49] Y. Wang, K. Chen, H. Tan, and K. Guo, "Tabi: An Efficient Multi-Level Inference System for Large Language Models," in *EuroSys*, 2023.

[50] K. K. W. Ng, H. M. Demoulin, and V. Liu, "Paella: Low-Latency Model Serving with Software-Defined GPU Scheduling," in *ACM SOSP*, 2023.

[51] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs," in *USENIX ATC*, 2022.

[52] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud Scale Load Balancing," in *ACM SIGCOMM*, 2013.

[53] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless Datacenter Load-balancing with Beamer," in *USENIX NSDI*, 2018.

[54] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu,

B. Cheyney, W. Shang, and J. D. Hosein, “Maglev: A Fast and Reliable Software Network Load Balancer,” in *USENIX NSDI*, 2016.

[55] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, “E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems,” *Proceedings of the VLDB Endowment*, 2014.

A Appendix

A.1 Starvation Prevention of Algorithm 1

Algorithm 3 shows the pseudo-code of the credit-based dynamic batching algorithm to prevent starvation. Initially, we assign the same credit to each LoRA adapter, which guarantees the fairness in the beginning. Whenever a request is not served in the FCFS order, the credit of the corresponding LoRA adapter needs to be transferred to the requests that orignially should be served in the FCFS order (line 14 and line 21). In this, way, if a LoRA adapter is not served for a long time, it will accumulate enough credit and our algorithm always serves them first (lines 11–15). We also prevent the LoRA adapter to merge into the base LLM weights when it does not have sufficient credit (line 18). To mitigate the oscillation of starvation prevention between different LoRA adapters, we set a threshold T_{starve} to decide the starvation (lines 6–10). This parameter can be tuned to make a tradeoff between performance and fairness.

Algorithm 3 Credit-based Dynamic Batching

```
 1: function CREDITBATCHING($B_{fcfs}, R, S, L$)
 2:     Input: FCFS requests $B_{fcfs}$, Request $R = \{r_1, r_2, ..., r_n\}$
 3:            Replica state $S$, LoRA adapters $L = \{l_1, l_2, ..., l_m\}$
 4:     Output: The batch of requests to be executed $B_{next}$
 5:     // Stavation Prevention
 6:     for $l \in L$ do
 7:         if $l_i.state == S_{starve} \wedge l_i.credit < T_{normal}$ then
 8:             $l_i.state = S_{normal}$
 9:         else if $l_i.credit > T_{starve}$ then
10:             $l_i.state = S_{starve}$
11:     $L_{starve} = \{l_i \in L \mid l_i.state == S_{starve}\}$
12:     $B_{starve} = \{r_i \in R \mid r_i.type \in L_{starve}\}[: max\_bs]$
13:     if $|B_{starve}| > 0$ then
14:         $transfer\_credit(B_{starve}, B_{fcfs})$
15:         return $B_{next} = B_{starve}$
16:
17:     // Adaptive switching between different modes
18:     $L_{eligible} = \{l_i \mid l_i.creidt \geq credit(\{r_i \mid r_i.type == l_i\}, B_{fcfs})\}$
19:     $RET$ =DYNAMICBATCHING($B_{fcfs}, R, S, L_{eligible}$)
20:     if $RET\ != \ B_{fcfs}$ then
21:         $transfer\_credit(RET, B_{fcfs})$
22:     return $B_{next} = RET$
```

Parrot: Efficient Serving of LLM-based Applications with Semantic Variable

Chaofan Lin[1]*, Zhenhua Han[2], Chengruidong Zhang[2], Yuqing Yang[2]
Fan Yang[2], Chen Chen[1]*, Lili Qiu[2]
[1]*Shanghai Jiao Tong University,* [2]*Microsoft Research*

Abstract

The rise of large language models (LLMs) has enabled LLM-based applications (a.k.a. AI agents or co-pilots), a new software paradigm that combines the strength of LLM and conventional software. Diverse LLM applications from different tenants could design complex workflows using multiple LLM requests to accomplish one task. However, they have to use the over-simplified request-level API provided by today's public LLM services, losing essential application-level information. Public LLM services have to blindly optimize individual LLM requests, leading to sub-optimal end-to-end performance of LLM applications.

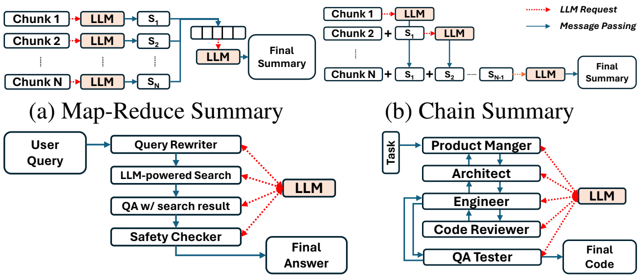
This paper introduces Parrot, an LLM service system that focuses on the end-to-end experience of LLM-based applications. Parrot proposes *Semantic Variable*, a unified abstraction to expose application-level knowledge to public LLM services. A Semantic Variable annotates an input/output variable in the prompt of a request, and creates the data pipeline when connecting multiple LLM requests, providing a natural way to program LLM applications. Exposing Semantic Variables to the public LLM service allows it to perform conventional data flow analysis to uncover the correlation across multiple LLM requests. This correlation opens a brand-new optimization space for the end-to-end performance of LLM-based applications. Extensive evaluations demonstrate that Parrot can achieve up to an order-of-magnitude improvement for popular and practical use cases of LLM applications.

1 Introduction

Large language models (LLMs) have demonstrated a remarkable language understanding capability [7, 41]. This enables a paradigm shift in application development. In this new paradigm, one or multiple application entities, known as AI agents or co-pilots, communicate with LLMs via natural language, known as "prompts", to accomplish a task collaboratively. For example, Meeting applications like Microsoft Teams or Google Meet can summarize meeting discussions through LLMs [33]. Search engines like Google and Bing can be enhanced with Chat ability through LLMs [14, 34]. It is believed such LLM-based applications will become the mainstream applications in the near future [13].

To accomplish a task, LLM-based applications typically require multiple rounds of conversation. The conversation, implemented through multiple API calls to LLM, demonstrates complex workflow patterns. Figure 1 illustrates several popular conversation patterns. For example, a meeting summary application [8, 33] often divides a lengthy document into multiple shorter sections, each satisfying the length constraint of the LLM conversation and thus can be summarized and combined into the final summary through the Map-Reduce or chaining summary patterns. Chat-based applications, e.g., Bing Copilot [34], call LLM APIs multiple times to generate answers based on user queries. Multiple agents, each representing a different role played by different LLM calls, can collaborate to achieve a task [22, 47, 54].

Public LLM service providers have to face diverse tenants and applications, each with different workflows and performance preference. However, existing API design for LLM service provision is still request-centric. Public LLM services only observe tons of individual requests, without knowing any



(a) Map-Reduce Summary (b) Chain Summary

(c) LLM-Powered Search (d) Multi-agent Coding

Figure 1: The workflow of popular LLM-based applications. The final result requires multiple LLM requests.

*This work is partially done while Chaofan Lin's internship and Dr. Chen Chen's visting scholar in Microsoft Research.

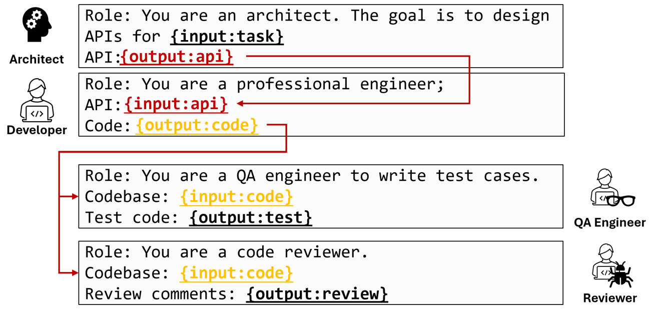
application-level information, e.g., which requests belong to the same application, how different requests are connected, or whether there are any similarities. The lost application-level information makes public LLM service blindly optimize the performance of individual requests, leading to sub-optimal *end-to-end* performance of LLM applications. In this paper, we observe there exist significant opportunities to improve the *end-to-end* experience of LLM applications by exploiting the application-level information, especially the *correlation* of multiple LLM requests.

First, multiple consecutive LLM requests may be dependent: the result of one request could be the direct input of the next request. Therefore, it is desirable to colocate those requests together and execute them consecutively on the LLM service side. However, unaware of their dependencies, these requests have to be executed interactively between the client side of LLM-based applications and the public LLM services. These clients, often located on the other end of the Internet, can only issue the second request after they receive the result of the first request. This unnecessarily incurs *extra overhead of consecutive requests* on network latency as well as losing the opportunity of co-scheduling these consecutive requests (§3).

Second, LLM requests may have *diverse scheduling preference*, even within a single application. For example, in Figure 1a, to reduce the end-to-end latency, the requests representing multiple Map tasks should be batched more aggressively to increase the throughput of the Map tasks; while the Reduce task, due to its scarcity, should be optimized for latency. Unfortunately, public LLM services cannot discriminate the difference between the two types of tasks. As a result, the current practice is to blindly optimize the latency for individual requests, which might not be desirable for the end-to-end experience.

Third, there exists a high degree of *commonality* across LLM requests. Popular LLM applications (e.g., Bing Copilot [32], GPTs [42]) use a long system prompt, including task definitions, examples, and safety rules, to guide the behavior of LLM applications. The long system prompt is usually static and common for all users. As existing public LLM services treat each request individually, these common prefix prompts are provided repeatedly in each request, leading to a great waste of storage, computation, and memory bandwidth. Our analysis of a production LLM-based search engine shows that over 94% of tokens in the requests are repeated across different users.

Although we have seen some emerging engine-level techniques [25,56,63] proposed to optimize the above three cases, they all work based on certain application-level knowledge, which is lost in nowadays public LLM services. In a nutshell, due to the lack of understanding of the correlations of LLM requests, existing LLM services cannot leverage the three opportunities, leading to high end-to-end service latency and reduced throughput. Based on the above facts and insights, we introduce Parrot, an LLM service system that treats LLM applications as first-class citizens. Parrot retains most of application-level information by a simple abstraction Semantic Variable, achieving a perfect balance between increasing system complexity and bringing new information for optimization. A Semantic Variable is a text region in the prompt with a specific semantic purpose, such as a task instruction, a list of few-shot examples, an input, or an output. A Semantic Variable can also work as the data pipeline that connects multiple LLM requests. Semantic Variable naturally exposes the information of prompt structures and correlations of requests to LLM services. By inspecting Semantic Variable at runtime, Parrot can perform conventional data flow analysis to derive the data dependency between LLM requests just-in-time.



```
Role: You are an architect. The goal is to design
APIs for {input:task}
API:{output:api}
```

```
Role: You are a professional engineer;
API:{input:api}
Code:{output:code}
```

```
Role: You are a QA engineer to write test cases.
Codebase: {input:code}
Test code: {output:test}
```

```
Role: You are a code reviewer.
Codebase: {input:code}
Review comments: {output:review}
```

Figure 2: The communication of consecutive LLM requests in multi-agent applications.

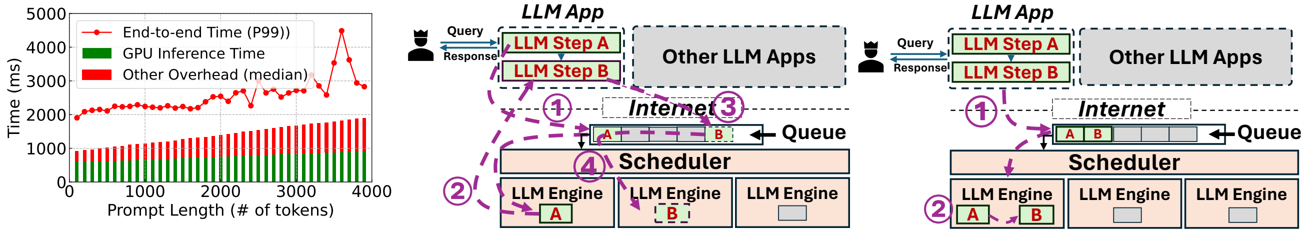
By analyzing the application-level information, Parrot's unified abstraction naturally enables joint optimizations, which bring better global optimality. The same data pipeline built by Semantic Variables can enable multiple optimizations simultaneously, including hiding data pipeline's latency, objective deduction for a better scheduling and commonality analysis to perform de-duplication. Parrot's scheduling also takes different opportunities into accounts under the unified abstraction. Our extensive evaluation of Parrot on popular LLM-based applications, including the production and open-source projects, shows Parrot achieves up to 11.7× speedup or 12× higher throughput compared with the state-of-the-art solutions. Parrot is open-sourced at `https://github.com/microsoft/ParrotServe`, including the code for artifact evaluations to reproduce our experiment results.

2 Background

LLM Service. Most LLM services are provisioned as a conditional generation service via a text completion API.

$$Completion(\text{prompt}:\text{str}) \rightarrow \text{generated\_text}:\text{str}.$$

The application client provides a text prompt, and the LLM service responds with the generated text. Behind the API, an LLM service provider runs one or multiple clusters of LLM inference engines. A request scheduler dispatches LLM



(a) Latency Breakdown

(b) Current LLM Services

(c) Our system: Parrot

Figure 3: The end-to-end latency breakdown of current LLM services. The source of the overhead comes from network and queuing due to chatty interaction between LLM application and LLM services, which is eliminated in our system Parrot.

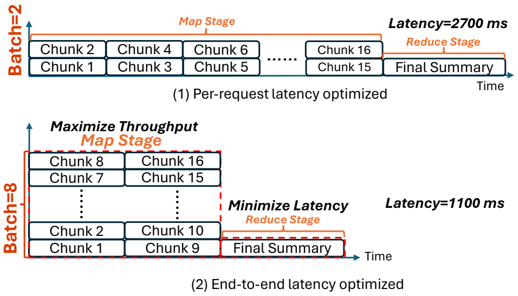
requests from a queue to an LLM inference engine, which uses a set of GPUs to conduct the LLM inference.

LLM-based Applications. Figure 1 highlights the representative workflows of how LLM is used in the applications. Due to the limited context window of LLMs (e.g., 4,096 for GPT-3.5-Turbo [40]), data analytics on long documents follow a *map-reduce style* (Figure 1a) or *chain style* (Figure 1b) workflow to generate the final results. It splits the long transcript into chunks, uses multiple requests to generate partial results for each chunk (the Map task), and combines them altogether (a Reduce task) or incrementally (the chain style) to generate the final result. Chat-based search engine in Figure 1c may use consecutive LLM requests to discern query intention, enrich the query with supplementary information, retrieve related data, undergo a safety check, and finally generate the response. Multi-agent in Figure 1d and Figure 2 is another type of workflow using multiple LLM requests, each with a designated role. Different roles work collaboratively on the same task, e.g., AutoGen [54] and MetaGPT [22] use the roles like product manager, architect, engineer, and QA tester. They communicate with each other on a software project. Each role is supported by one or multiple LLM requests to act as the designed role to generate their responses.

3 Problems of Serving LLM Applications

Although LLM's text completion API provides a flexible way of building LLM applications, it loses the application-level information to public LLM services, leading to the following challenges.

Excessive Overhead of Consecutive Requests. As demonstrated in Figure 1, LLM applications frequently make multiple LLM calls to complete a single task. Due to the request-centric design of existing public LLM services, which generate responses for each request individually, developers have to parse the output of an LLM request and compose the prompts for subsequent LLM requests on the client side. Figure 3a shows our empirical study of the latency breakdown of the LLM calls from a popular LLM application in our production, which uses a chain-style workflow. The prompt lengths range from 150 to 4000 tokens and the output length is around 50 tokens. We find there is a significant portion of the latency of LLM API call originates outside the LLM engine (30 ∼ 50% on average and over 70% in the worst cases). The overhead increases with the growing length of prompts. The high latency can sometimes result in API timeouts and resubmissions.



(1) Per-request latency optimized

(2) End-to-end latency optimized

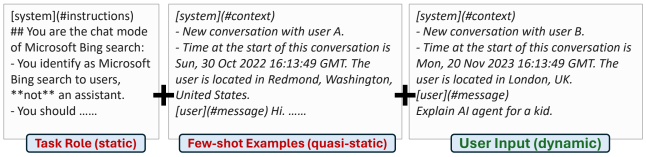
Figure 4: Request-centric scheduling v.s. application-centric scheduling for the map-reduce style document summary task.

Such overhead is due to the chatty interaction between LLM services and clients. Figure 3b illustrates the overhead of a simple two-step LLM application (e.g., *chain-style* summary of two text chunks). Existing LLM services are unaware of the dependency among such requests, where the output of the previous request may be the direct input of the next one. For such consecutive and dependent requests, the client has to wait for the arrival of the response to the first LLM request (②) before submitting the next LLM request (③). This unnecessarily incurs heavy network latency because clients and LLM services are typically in different data centers. Moreover, the next LLM request has to suffer extra queuing delays (④), because requests from other applications may arrive between the consecutive LLM requests.

LLM-based App.	# Calls	Tokens	Repeated (%)*
Long Doc. Analytics	$2 \sim 40$	$3.5k \sim 80k$	3%
Chat Search	$2 \sim 10$	$5k$	94%
MetaGPT [22]	14	$17k$	72%
AutoGen [54]	17	$57k$	99%

*We count a paragraph as repeated if it appears in at least two LLM requests.

Table 1: Statistics of LLM calls of LLM applications.



```
[system](#instructions)
## You are the chat mode
of Microsoft Bing search:
- You identify as Microsoft
Bing search to users,
**not** an assistant.
- You should ......
```

```
[system](#context)
- New conversation with user A.
- Time at the start of this conversation is
Sun, 30 Oct 2022 16:13:49 GMT. The
user is located in Redmond, Washington,
United States.
[user](#message) Hi. ......
```

```
[system](#context)
- New conversation with user B.
- Time at the start of this conversation is
Mon, 20 Nov 2023 16:13:49 GMT. The
user is located in London, UK.
[user](#message)
Explain AI agent for a kid.
```

Figure 5: The prompt structure of search copilot shows a long prompt reused by different user queries.

In Table 1, we evaluated four popular LLM applications. The first two are from our production, and the last two are popular open-source projects. They all require tens of LLM calls to complete a single task, which results in high user-perceived latency. Our evaluation in §8.2 shows LLM services that treat requests individually could slow down the end-to-end latency by over 2×. An LLM service can eliminate the overhead if it can handle consecutive requests in a batch. Parrot adopts such an approach. As shown in Figure 3c, the two steps of the same application are scheduled together, thus allowing the output of Step A to be fed directly into Step B—with the network and queuing overhead bypassed.

Misaligned Scheduling Objectives. Due to the lost application information (workflow and application performance objective), existing public LLM services have to blindly use a universal treatment for all requests, e.g., optimizing per-request latency [44]. However, LLM-based applications are more concerned about the end-to-end experience, rather than individual requests. This misaligned optimization objectives may negatively impact end-to-end performance. Considering the map-reduce document summary in Figure 1a, the system should minimize the end-to-end time it takes to receive the final summary, rather than the latency of individual requests. The LLM services optimized for individual requests are not optimal for end-to-end latency.

As depicted in Figure 4, current LLM services must limit the number of concurrent requests running on each LLM engine to control the latency of individual requests. However, there is a trade-off between latency and throughput in LLM inference. Increasing the batch size can bring up to 8.2× higher throughput but lead to 95% higher latency [9]. Yet, if we understand the application-level performance objective, which in this case is the end-to-end latency, we can determine that the ideal scheduling strategy should maximize the throughput (using higher batch sizes) during the map stage and minimize request latency during the reduce stage. This strategy reduces end-to-end latency by 2.4×. Moreover, it uncovers the potential to enhance cluster throughput without compromising the end-to-end latency of LLM applications. This insight is essential for addressing the conflict between rising demand and limited hardware resources. It underscores the necessity of scheduling LLM requests from the perspective of LLM applications, but it also presents the challenge of managing diverse LLM requests with varying performance objectives.

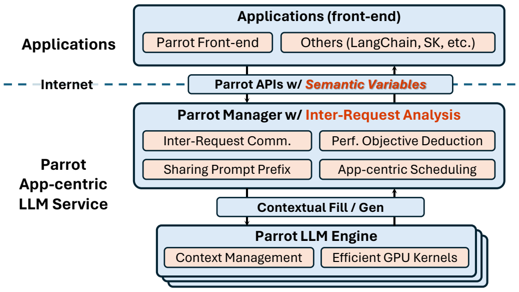


Figure 6: Parrot system overview.

Redundant Computations. Currently, most LLM-based applications exhibit a high degree of redundancy in the prompts of their requests. For instance, Bing Chat [32] has handled more than 1 billion chat prompts. These prompts share the same system prompts that defines the functionality of Bing Chat. OpenAI introduces GPTs [42] to let users customize a ChatGPT for a specific purpose whose prompt template is the same across users. The commonality in prompts is crucial as it delineates the functionality and restrictions of LLM-based applications. The prompt structure in Figure 5 [52] includes a role definition, several examples to enhance the precision of LLM's behaviors and user query details. While the user input is dynamic, the task role is always fixed, and the few-shot examples could be quasi-static in that the same type of tasks use the same examples. This is why more than 94% of prefix tokens could be repetitively used across LLM requests for various users (Table 1). Such commonality also exists in multi-agent applications. For example, MetaGPT [22] and AutoGen [54] recurrently incorporate conversation history into the prompt over several rounds of LLM requests, leading to 72% and 99% redundancy respectively. These redundant sections excessively utilize GPU memory bandwidth and are computed for multiple times. Earlier results have proposed optimizations in LLM engines to avoid redundant GPU memory of shared prompt [25]. However, it is hard for public LLM services to swiftly detect and co-locate the prompt-sharing requests, which be dynamically generated, from tons of diverse requests from diverse applications. Without knowledge about the prompt structure, extensive token-by-token matching for every LLM request is expensive at the cluster level. Hence, if the cluster scheduler of public LLM service cannot dispatch prompt-sharing requests to the same engine, the engine-level redundancy avoidance optimizations would be hard to take effect.

4 Parrot Design

Figure 6 depicts the overview of Parrot's design. Parrot provides a natural way of programming LLM applications with Semantic Variable annotations (§4.1), which is compatible of existing LLM orchestration frameworks, e.g., LangChain [8]. Centering on this abstraction, Parrot Manager is designed

```
import Parrot as P
from Parrot.PerformanceCriteria import LATENCY

@P.SemanticFunction
def WritePythonCode(task: P.SemanticVariable):
""" You are an expert software engineer.
    Write python code of {{input:task}}.
    Code: {{output:code}}
"""

@P.SemanticFunction
def WriteTestCode(
    task: P.SemanticVariable,
    code: P.SemanticVariable):
""" You are an experienced QA engineer.
    You write test code for {{input:task}}.
    Code: {{input:code}}.
    Your test code: {{output:test}}
"""

def WriteSnakeGame():
  task = P.SemanticVariable("a snake game")
  code = WritePythonCode(task)
  test = WriteTestCode(task, code)
  return code.get(perf=LATENCY), test.get(perf=LATENCY)
```

Figure 7: Example: a multi-agent application in Parrot.

to schedule LLM requests at a cluster-level, by deriving the application-level knowledge (§4.2) and optimizing end-to-end performance of application (§5). The manager will schedule the LLM requests to LLM `Engine`, which is formed by a GPU server (or a group of servers) in the cluster that can serve LLM requests independently.

4.1 Semantic Variable

Parrot treats an LLM request as a semantic function[1] implemented using natural language and executed by LLMs. A Semantic Variable is defined as a input or output variable of a semantic function, which is referred as a placeholder in the prompt. Figure 7 shows a simplified example of multi-agent application like MetaGPT [22]. It contains two `SemanticFunctions`, one for the software engineer to write code and one for the QA engineer to write test code. It has three Semantic Variables: `task`, `code`, and `test`, for task description, the code to be developed by the software engineer, and the test code to be developed by the QA engineer, respectively. Although existing LLM orchestration frameworks (e.g., LangChain [8]) also allow placeholders in a prompt, however, the placeholders are rendered with real data before the submission, hence public LLM services cannot detect such a structure. Instead, Parrot relies on Semantic Variables to preserve the prompt structure for further inter-request analysis in public LLM services side.

In addition to the semantic functions, LLM application developers can further define orchestration functions that connect multiple semantic functions (e.g., `WriteSnakeGame` in Figure 7). The Semantic Variables connecting multiple semantic functions form the data pipeline of multiple LLM requests in the public LLM service. A simple data flow analysis of the semantic functions can be done to reveals the connections of multiple LLM requests. E.g., in Figure 7, the `code` variable connects the two LLM requests originating from `WritePythonCode` and `WriteTestCode`, showing their sequential dependency. Different from traditional completion API, Parrot splits a completion request to `submit` operation and `get` operation (§7). A function calling of `SemanticFunction` will trigger the `submit` API to submit a LLM request with its prompt and input Semantic Variables. The execution of a `SemanticFunction` is asynchronous thus it returns the futures of the output Semantic Variables. Through the `get` API, applications can fetch the value of an output Semantic Variable from the public LLM service in an on-demand manner. This asynchronous design allows Parrot-powered LLM service to receive all LLM requests not blocked by native functions and analyze their relationships just-in-time.

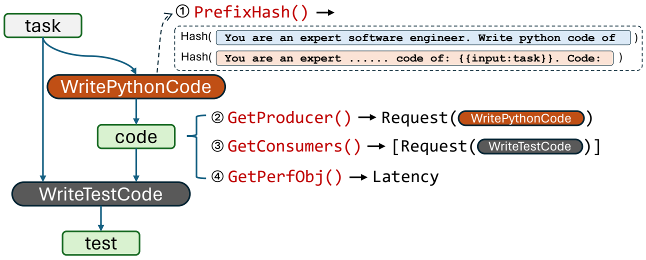

task
① PrefixHash() →
Hash(You are an expert software engineer. Write python code of)
Hash(You are an expert ...... code of: {{input:task}}. Code:)
WritePythonCode
code
② GetProducer() → Request(WritePythonCode)
③ GetConsumers() → [Request(WriteTestCode)]
④ GetPerfObj() → Latency
WriteTestCode
test

Figure 8: Primitives (selected) for Inter-Request Analysis.

The `get` operation supports annotation of performance criteria, showing the end-to-end performance requirement of an application, which can be end-to-end latency or throughput (extensible to more criteria like per-token latency when streaming, and time-to-first-token). For example, the final outputs, `code` and `test` in Figure 7, are fetched using `get` with an objective of end-to-end latency. Criteria of middle variables will be automatically deduced and propagated from final outputs (§5.2). After propagation, each variable is attached to a criterion, which finally works by serving as a hint to Parrot's scheduler (§5.4).

4.2 Primitives of Inter-Request Analysis

In general, Parrot perform inter-request analysis mainly by two types of application-level information deduced from Semantic Variable: DAG of requests and prompt structure. Figure 8 illustrates the DAG workflow of the example shown in Figure 7 and the primitives used for inter-request analysis and optimizations.

DAG-based analysis. As requests, or `SemanticFunctions`, are submitted beforehand, Parrot can receive them all at once and analyze their correlations just-in-time on the service side. Parrot maintains a DAG-like data structure in each user's

[1] The term *semantic function* is borrowed from Semantic Kernel [36].

registered session. Each node is either a request or a Semantic Variable that connects different requests. When a request comes, Parrot inserts it to DAG by linking edges with Semantic Variables it refers through placeholders in the prompts. Parrot can perform conventional dataflow analysis [1, 38] using the primitives to get the producer and consumers of Semantic Variables (i.e., `GetProducer` and `GetConsumers`) to recover dependency of LLM requests. Using the request DAG and the annotated performance criteria (via `GetPerfObj`) of final output Semantic Variables, Parrot can deduct the request-level scheduling preference by analyzing the DAG and the performance objective of final outputs (§5.2).

Prompt structure-based analysis. Based on the prompt structure declared by Semantic Variables, Parrot supports extracting the hash values of an LLM request at multiple positions split by Semantic Variables (i.e., `PrefixHash`). For example, the prompt of `WritePythonCode` has two potential sharing prefix: the text before `{{input:task}}` and the text before `{{output:code}}`, thus there will be two prefix hash values generated. The prefix hashes of LLM requests will be used by swift detection of commonality across multiple requests, supporting both static and dynamically generated contents, as well as within the same type of application or even across applications (§5.3).

5 Optimizations with Semantic Variable

5.1 Serving Dependent Requests

To avoid the unnecessary client-side execution, it requires the dependency of requests at the application level, which is lost in today's public LLM services. With the DAG and primitives illustrated in §4.2, Parrot serves dependent requests efficiently through a graph-based executor. The executor polls constantly and sends it to corresponding engine once ready (i.e. producer requests are all finished), which allows instant execution and maximizes batching opportunities. For consecutive execution of dependent requests, materialized value is transmitted through a message queue allocated for corresponding Semantic Variable, avoiding unnecessary chatty communication between clients and LLM services.

The value of a Semantic Variable in a request may require transformation before being exchanged, e.g., the value of a Semantic Variable is extracted from the JSON-formatted output of an LLM request, which is then fed into consecutive LLM requests. Similar to existing message queue systems that support message transformation (e.g., Kafka [5]), Parrot also supports string transformation to manipulate Semantic Variables during value exchanging among LLM requests. Parrot supports most output parsing methods of LangChain [8], which covers most use cases of LLM applications.

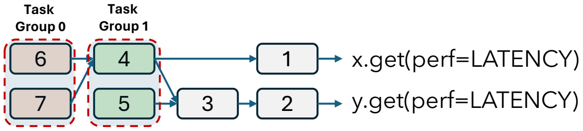


Figure 9: Performance deduction for an LLM-based application generating two latency-sensitive Semantic Variable.

5.2 Performance Objective Deduction

To optimize the end-to-end performance of applications, we need to know the application-level performance criteria. To help deriving the request-level scheduling preference from the end-to-end application's performance requirement, we need to understand the workflow of the LLM application, which is the DAG of LLM requests derived by Parrot's primitives.

When an application annotates a Semantic Variable to prefer higher throughput, all requests generating this Semantic Variable (both directly or indirectly) will be marked as throughput-preferred when scheduling. This scheduling preference is usually beneficial for offline data processing, such as bulk document analysis.

Handling latency-sensitive applications is more intricate. As demonstrated in Figure 4, achieving low end-to-end latency may sometimes require prioritizing throughput at the Mapping stage. The latency of individual requests can sacrificed so as to reduce the completion time of the entire DAG of requests. Parrot analyzes LLM requests in reverse topological order, beginning with those linked to latency-critical Semantic Variable, as depicted in Figure 9. With the extracted DAG, LLM requests that directly result in latency-critical Semantic Variables are labeled as latency-sensitive (Request 1 and 2), as are their immediate predecessors (Request 3). Parallel LLM requests at the same stage are grouped into a *task group* (Task Groups 0 and 1). The scheduler should minimize the latency of the entire task group, often leading to a higher batch capacity for higher throughput of token generation.

5.3 Sharing Prompt Prefix

When an LLM request is scheduled to an LLM engine, a context on the engine is created to store the state of the model execution for this request (mainly KV cache). Existing works have proposed to share the KV cache of common prefix of prompts in LLM engines to save the GPU memory. However, as we have explained in §3, today's public LLM service face diverse applications and requests, which is hard to identify the commonality at the cluster level. Token-by-token comparison is impractical due to high time complexity, especially for very long context with massive requests. In Parrot, by exposing Semantic Variables to LLM service, we can understand the prompt structure to automatically detect the commonality more efficiently at the granularity of Semantic Variables. Using Parrot's primitive of `PrefixHash`, Parrot only needs to check the hash value at positions after each Semantic Variable in a request's prompt. Parrot maintains a key-value store, where each entry maps a (hashed) prefix of tokens to a list of requests, thus the scheduler can quickly check the opportunity in an online manner, supporting both static and dynamically-generated prompt within one application or even across different applications.

Furthermore, we propose better GPU kernel for the attention computation of the requests with a common prefix. We first leverage vLLM's paged memory management [25] to save the redundent GPU memory. But vLLM's kernel still suffers from redundant computation and memory loading of the shared tokens. Therefore, we design a new Attention decoding algorithm by combining FlashAttenation [12] and PagedAttention [25] that treat the shared and non-shared token separately. This significantly accelerates the attention of shared contexts (implementation details in §7).

5.4 Application-Centric Scheduling

Parrot's scheduling is a problem that matches LLM requests to LLM engines, i.e. the cluster-level scheduling, while the engine-level scheduling will be covered in the implementation details of the engine in §7. To fix the problem of existing public LLM service that blindly optimize diverse individual requests, Parrot's scheduling policy leverages the application-level knowledge to optimize the end-to-end performance. Specifically, the primary goal of Parrot's scheduler is to meet the varied performance goals of LLM applications while optimizing GPU cluster utilization. As explained in §3, a conflict arises when combining throughput and latency oriented requests: large batch sizes increase throughput and GPU efficiency but degrade latency, and vice versa. Transformer-based LLM inference is largely memory-bound, with latency influenced by the count of concurrent tokens within the engine. To meet performance targets of LLM applications, particularly latency, an LLM engine must regulate the token count below a specified threshold, which is determined by the LLM request with the most strict latency constraint. Therefore, Parrot's scheduling principles are twofold: (1) group LLM requests with similar performance requirements to circumvent the conflict, and (2) maximize opportunities for sharing across requests.

Algorithm 1 outlines the scheduling process of Parrot. With the extracted DAG, the system arranges the LLM requests according to their topological order (line 1). Parrot tends to schedule requests belonging to the same application together to avoid the slowing down of interleaved scheduling (§8.2). For requests identified as part of a task group through Parrot's performance objective deduction, the scheduler attempts to allocate the entire task group together (line 4-line 5). Additionally, if Parrot detects other queued requests or running contexts with a common prefix, it tries to assign them to the same LLM engine (line 3, line 6-line 9), to utilize Parrot's context fork to reduce the redundant computation and GPU memory transactions. For an LLM request without the above opportunity, Parrot schedules the request independently (line 10-line 11). Due to limited space, we omit the details of how Parrot chooses LLM engines (i.e., `FindEngine`). Briefly, Parrot finds the engine that satisfies the scheduling preference of a request while minimizing the negative impacts. For instance, if a latency-sensitive request is scheduled to an LLM engine that can run up to 64,000 tokens of throughput-driven requests, its capacity will be significantly reduced to 2,000 to satisfy its strict latency requirement. But, if it is scheduled to an engine that has already been running a latency-sensitive request, the capacity reduction is negligible.

Algorithm 1: Parrot's Request Scheduling.

```
   Data: Q: the request queue
1  Q.sort() ; /* Topological order                */
2  for $r \in Q$ do
3      SharedReqsInQueue, CtxInEngine =
         FindSharedPrefix($r$);
4      if $r.TaskGroup \neq \varnothing$ then
5          $r^*$ = FindEngine(r.TaskGroup);
6      else if $SharedReqsInQueue \neq \varnothing$ then
7          $r^*$ = FindEngine(SharedReqsInQueue);
8      else if $CtxInEngine \neq \varnothing$ then
9          $r^*$ = FindEngine(r, filter=CtxInEngine);
10     if $r^* = \varnothing$ then
11         $r^*$ = FindEngine($r$);
12     Q.remove($r^*$);
```

6 Discussion

Dynamic Applications and Function Calling. Currently, Parrot only supports cloud-side orchestration of LLM requests without involving dynamic control flow and native functions (e.g., Python Code). They still require client-side execution. We intentionally disable the offloading of these functions to public LLM services to minimize the security risks of malicious injection. For private LLM services whose LLM applications are trusted or there is a trusted zone to execute these functions, Parrot's APIs can be easily extended with conditional connections and native code submission. Moreover, these extensions further enable new optimizations, e.g., we can speculatively pre-launch high-probability branches in dynamic applications based on past profiles. This also proves the potential of Parrot's design when facing new types of applications. We leave these extensions as future works.

Other Applications of Inter-Request Analysis. The inter-request analysis in Parrot enables a new optimization space not limited to the ones we introduced in §5. A large-scale service has more scheduling features to consider, including handling outliers [3], job failures [58], delay scheduling [57], fairness [15, 61], starvation [17], or supporting heterogeneous clusters [24, 37], which have been widely studied in other systems. Parrot provides a new view from the perspective of LLM-based applications: we need to understand the interconnection and commonality of LLM requests to optimize applications' end-to-end performance. These features can be revisited in the LLM service system by considering the new characteristics of LLM applications. In this paper, we focus on Parrot's mechanisms and a few use cases, leaving other optimizations as promising future works.

Parrot with LLM Orchestration Frameworks. There have been several frameworks for developers to build LLM-based applications, e.g., LangChain [8], SemanticKernel [36], and PromptFlow [35]. The key function of these frameworks is to "glue" different LLM calls to accomplish a complex task (aka. LLM orchestration). Parrot can be integrated with these frameworks by extending their calling of LLM service APIs with Semantic Variables. Most of these frameworks have already used a template-based approach in which developers can design a template with placeholders, and render the placeholders at runtime. These placeholders naturally have the same concept as Parrot's Semantic Variable. However, because these frameworks will render the template prompt before the submission, LLM services lose the information on the prompt structure. To make these frameworks compatible with Parrot, both the template itself and the variables to render the template (using Semantic Variable in Parrot) need to be wrapped as a `SemanticFunction` so the necessary information is exposed to Parrot's LLM service.

7 Implementation

Parrot is an end-to-end LLM service for LLM applications, implemented on Python with about 14,000 lines of code. Its front-end provides the abstraction of Semantic Variable, and `SemanticFunction`, which is transformed into Parrot's APIs (implemented with FastAPI [48]) to be submitted as LLM requests. A centralized Parrot manager handles the management of LLM requests, including Semantic Variables, communication, and scheduling. We also build an LLM engine based on efficient kernels from vLLM [25], xFormers [26], and ourselves. The engine supports advanced features for LLM serving, including paged memory management [25] and continues batching [56]. Parrot's front-end and manager are implemented in 1,600 and 3,200 lines of Python, respectively. Parrot's LLM engine is implemented in 5,400 lines of Python and 1,600 lines of CUDA. We have implemented OPT [60] and LLaMA [51] with PyTorch [45] and Transformers [53].

APIs. Applications programmed by `SemanticFunctions` or other frontends are finally lowered to requests to universal APIs through different adapters. Parrot provides OpenAI-like APIs with the extension of Semantic Variables. The request body of two operations mentioned in §4.1 is shown as follows:

```
(submit) {"prompt": str, "placeholders": [{"name":
↪  str, "in_out": bool, "semantic_var_id": str,
↪  "transforms": str}, ...], "session_id": str}

(get) {"semantic_var_id": str, "criteria": str,
↪  "session_id": str}
```

In addition to the static string prompt, Parrot preserves the input and output placeholders. A placeholder is associated with a semantic variable either for rendering the input or parsing the output. As introduced in §5.1. Parrot supports transformations before the input or after the output. Parrot also supports other APIs for setting and fetching the value of Semantic Variables. The error message will be returned when fetching an Semantic Variable, whose intermediate steps fail (including engine, communication, and string transformation).

Kernel Optimization. vLLM's GPU kernel, while capable of reusing results cached in GPU memory for shared prefix tokens in a prompt, sometimes excessively reloads these tokens from global to shared memory, impeding attention score computations. Using OpenAI Triton [43] and CUDA, we have developed a novel GPU kernel, integrating concepts from PagedAttention [25] and FlashAttention [11, 12], to accelerate attention decoding computation involving shared prefixes. This kernel retains PagedAttention's approach of storing the key-value (KV) cache in disparate memory segments and utilizes a page table per request to monitor block status and placement. Furthermore, employing FlashAttention principles, the kernel maximizes data reuse within shared memory. Unlike reloading tiles repeatedly in the PagedAttention's implementation, it loads KV cache tiles for the shared prefix to shared memory only once, diminishing memory transactions between the L2 Cache and Shared Memory. The kernel initially calculates interim attention metrics (including attention scores, `qk_max`, `exp_sum`) for the shared prefix using the loaded tiles and records these back to HBM. Subsequently, it processes the new tokens' partial attention beyond the prefix, amalgamating this with the prefix's interim results to derive the ultimate attention output.

Universal Engine Abstraction. Parrot's cluster manager controls multiple engines running various models, tokenizers, KV cache layouts, etc. To enable Parrot's optimizations, LLM engines need to support (1) stateful generation (e.g., guidance [18]) and (2) sharing KV cache states across different requests. Hence we propose a universal abstraction to describe the minimal capability required to LLM engines to be integrated into Parrot.

```
def Fill(token_ids: List[int], context_id: int,
 ↪  parent_context_id: int)
def Generate(sampling_configs: Dict, context_id:
 ↪  int, parent_context_id: int)
def FreeContext(context_id: int)
```

These three methods not only cover the basic completion functionality of LLM inference engine, but also provide a flexible context management interface. The `Fill` method processes the initial prompt tokens, calculates and fills the KV cache into corresponding context. The `Generate` method produces tokens via generative decoding that produces one token per iteration until it reaches the length limit, user-defined termination character or EOS (end-of-sequence) token, under certain sampling configurations (e.g. temperature). `Fills` and `Generates` are scheduled and batched by engine's scheduler per iteration using continuous batching [56]. Creating and forking contexts can also be realized with these two methods by setting `context_id` and `parent_context_id`, respectively. The `FreeContext` method explicitly frees a context (i.e. free its KV cache in GPU memory). Separating `Fill` and `Generate` not only fits Semantic Variable naturally: constant text and input values are processed by `Fill`; the output values are generated by `Generate`, but also breaks the request-level dependency into a finer granularity, enabling more parallel execution opportunities [2, 21, 46, 64].

8 Evaluation

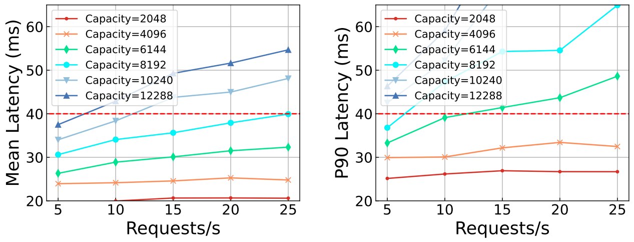
8.1 Experimental Setup

Testbed. We evaluate Parrot with two separate setups for single-GPU and multi-GPU experiments. The single-GPU evaluations use a server with a 24-core AMD-EPYC-7V13 CPUs equipped with one NVIDIA A100 (80GB) GPU. The multi-GPU evaluations use a server with 64-core EPYC AMD CPU and four NVIDIA A6000 (48GB) GPUs. Both servers run CUDA 12.1 and cuDNN 8.9.2.

Workloads. Our evaluations are performed to run four representative LLM applications. Each LLM engine uses one GPU and runs a LLaMA 13B or LLaMA 7B model [51] . For LLM-based data analytics on long documents, we use the Arxiv dataset [27], executing chain and map-reduce summarizations on an extensive collection of academic papers. To investigate the sharing opportunities of LLM-based applications with many users, we run the prompts from Bing Copilot and GPTs [42] with synthesized user queries. For multi-agent applications, we build a multi-agent programming application using MetaGPT [22], which contains a system architect to design APIs, multiple programmers to write code for different files, reviewers to share review comments. The programmers will also revise the code based on comments. For chat service workloads, we derived scenarios from the ShareGPT dataset [50], which mirrors real LLM chat conversations. According to the distribution of our measurement, we introduced a random delay of 200 $\sim$ 300 ms to LLM requests to emulate typical network overhead seen over the Internet. To create realistic workloads, we documented the LLM responses using GPT-4 [41], ensuring the LLaMA models generated text of similar length for system performance analysis. Table 2 presents the workloads and their optimizations in Parrot.

Workload	Serving Dependent Requests.	Perf. Obj. Deduction	Sharing Prompt	App-centric Scheduling
Data Analytics	✓	✓		✓
Serving Popular LLM Applications			✓	✓
Multi-agent App.	✓	✓	✓	✓
Mixed Workloads	✓	✓		✓

Table 2: The workloads and the optimizations taking effect.

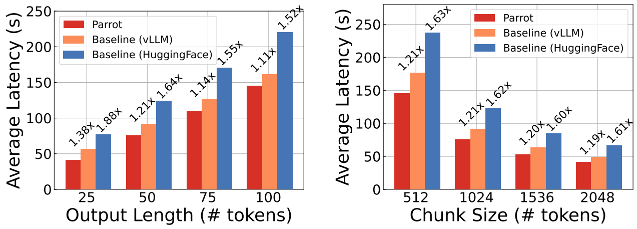


(a) Mean Latency

(b) P90 Latency

Figure 10: Latency (per output token) of vLLM with varying token capacities and request rates. Requests are sampled from ShareGPT [50] and their arrival time follows Poisson distributions.

Baseline. We benchmark Parrot against sate-of-the-art solutions for building LLM applications and serving LLM requests. The majority of LLM applications used in our baseline comparisons are developed using LangChain [8], which is the predominant framework for LLM application development. The LLM applications in baselines leverage OpenAI-style chat completion APIs as provided by FastChat [62]. FastChat is a widely recognized open-source LLM serving system with over 30,000 stars on its repository. Incoming requests to FastChat are allocated to LLM engines that run either HuggingFace's Transformers library [53] or vLLM [25], both of which incorporate cutting-edge enhancements for LLM execution, such as FlashAttention [12], PagedAttention [25], and continuous batching techniques [56]. The default scheduling strategy employed by FastChat assigns incoming requests to the LLM engine with the smallest current queue. Since existing LLM services typically expose their functionality through "chat" completion APIs, baseline assessments treat all requests as independent and assume a high sensitivity to latency. To manage token generation response times, each LLM engine is subject to a capacity threshold, which is the



(a) Output lengths

(b) Chunk sizes

Figure 11: Average E2E latency of chain summarization with varying output lengths and chunk sizes.

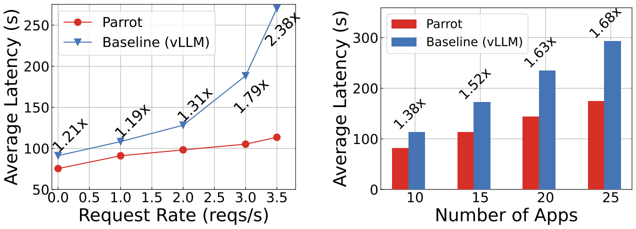
aggregate token count from all active requests on the engine.

Since existing LLM token generation is usually bound by memory bandwidth, the per-token generation latency of an engine is mainly affected by the number of running tokens in a batch. As depicted in Figure 10, our experiments indicate that the latency per output token, i.e. TPOT (Time-per-output-token) for vLLM, with continuous batching enabled, experiences a notable uptick when the engine's workload using a batch capacity beyond 6144. In our evaluation, we use the setting that an LLM engine can keep its generation latency under 40 ms/s for latency-sensitive requests, consistent with our experience of OpenAI's LLM services. When all LLM engines hit their maximum capacity, any additional LLM requests are queued in a FIFO (First In, First Out) manner, awaiting the completion and release of resources by ongoing tasks. Serving longer context (e.g., 32k or even 1M tokens) within a satisfactory latency require either more GPUs using tensor-parallel [49] or sequence-parallel [6] approaches, or approximate attention (e.g., StreamingLLM [55]), which is beyond the scope of this paper.

8.2 Data Analytics on Long Documents

Our experimental analysis within data analytics randomly picks ten long documents from the Arxiv-March dataset [27], using chain-summary and map-reduce summary. Each document has over 20,000 tokens. The results measures the mean end-to-end latency across all documents.

Chain-style Applications. Our evaluation demonstrates how Parrot enhances chain summarization by mitigating the excessive communication overhead stemming from client interactions. Figure 11 presents the average end-to-end latency for summarizing a single document using one LLM engine (A100, LLaMA 13B) . We adjust the chunk size (the count of tokens per chunk) and the output length, with results shown in Figure 11a and Figure 11b, respectively. Parrot achieves a reduction in end-to-end latency by as much as 1.38× and 1.88× compared to the baselines employing vLLM and HuggingFace, respectively. The efficiency of Parrot primarily stems from the decreased network latency, which is a consequence of reduced client interaction. As the output length increases, the time spent on generation becomes more significant, leading to a diminishing advantage for Parrot over the baseline. By increasing the chunk size, we decrease the number of chunks, yet the extent of the speedup is contingent upon the network latency savings for each chunk. Given that token generation is substantially more time-consuming than prompt processing, we observe a consistent speedup with variable chunk sizes and a fixed output length (1.2× and 1.66× relative to vLLM and HuggingFace, respectively). This indicates that Parrot's optimization for dependent LLM requests is particularly beneficial for shorter outputs, which are prevalent in various LLM applications such as summarization, short answer generation, scoring, and choice provision. Due to HuggingFace's slower performance relative to vLLM, subsequent evaluations focus solely on the comparison between Parrot and vLLM.



(a) With background requests

(b) Multiple summary apps.

Figure 12: Average E2E latency of chain-summary with background requests or other chain-summary applications.

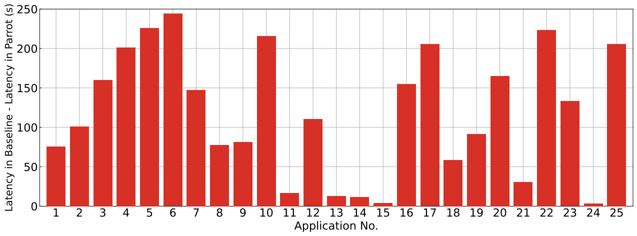
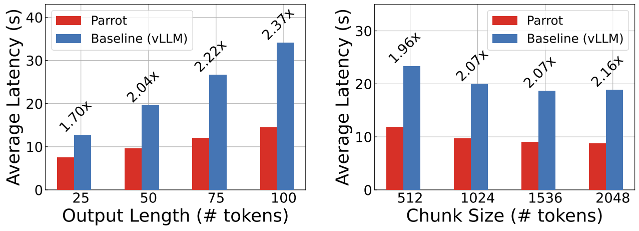


Figure 13: The difference in E2E latency of the 25 chain-summary application between Baseline and Parrot. All applications finish earlier in Parrot.

Figure 12a extends the evaluation by introducing background LLM requests at varying rates to examine the capability of Parrot in mitigating additional queuing delays for dependent requests. Parrot slashes the end-to-end latency by a factor of 2.38× in comparison to the baseline (vLLM). With Parrot, as soon as the summary for the first chunk is completed, the subsequent chunk is processed immediately by incorporating the summaries of previous chunks into the prompt, which aids in generating the summary for the next chunk. In contrast, the baseline treats all LLM requests individually. As a result, in addition to the network latency from client interactions, subsequent requests must re-enter the queue, leading



(a) Output lengths

(b) Chunk sizes

Figure 14: Average E2E latency of Map-Reduce document summary with varying output lengths and chunk sizes.

to added queuing delays. Figure 12b further illustrates the end-to-end latency when multiple chain-summary applications are submitted concurrently, with each application tasked with generating a summary for a separate document. Parrot manages to reduce the average end-to-end latency for all applications by 1.68× without slowing down any applications compared to the baseline according to Figure 13. The baseline, by interleaving the execution of different applications, exacerbates the slowdown of the end-to-end latency for all applications. These experiments validate that recognizing the interconnections of LLM requests can significantly enhance end-to-end performance, as opposed to processing requests in isolation.

Map-Reduce Applications. An alternative implementation of the document summarization application follows the map-reduce paradigm as depicted in Figure 1a. This approach consists of multiple parallel mapping LLM requests, where each request summarizes a distinct segment of the document, followed by a reducing LLM request that aggregates these individual summaries into a final summary. As shown in Figure 14, Parrot realizes a 2.37× acceleration over the baseline with one LLM engine (A100, LLaMA 13B). Since the mapping LLM requests are independent, they are dispatched concurrently by both Parrot and the baseline. The primary advantage of Parrot stems from its deduction of a performance objective that identifies the mapping tasks as a task group. By recognizing this relationship, Parrot is capable of optimizing the latency of the entire task group through larger batch sizes, which in turn enhances throughput. In contrast, the baseline processes each LLM request in isolation, operating under the presumption that they are all sensitive to latency. This constrains the baseline to utilize a limited token capacity (4096 tokens) on the LLM engine to achieve optimal latency for individual tasks, which is detrimental to the end-to-end performance of applications. It underscores the necessity for LLM services to distinguish LLM requests to optimize the end-to-end performance of varied LLM applications.

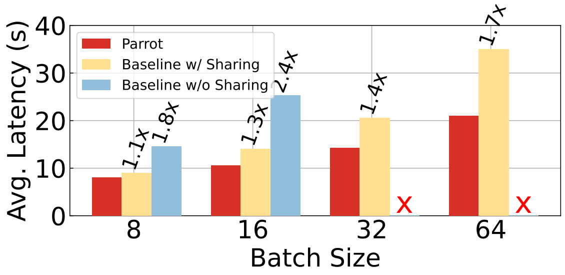
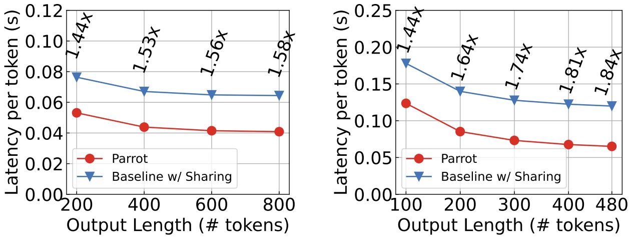


Figure 15: Latency of Bing Copilot with varying batch sizes.

8.3 Serving Popular LLM Applications

Production applications need to face massive users. As explained in Figure 5, developers often need to use a very long system prompt to define the behavior of LLMs. Therefore, users of the same LLM application often use the shared prompt, which can benefit from Parrot's context fork mechanism and Parrot's scheduling policy that co-locates LLM requests sharing a long prompt prefix. Because we do not have access to the intermediate steps of Bing Copilot, we only evaluate the final request generating the response to users. We synthesized 64 requests from the length distribution we measured using Bing Copilot. The system prompt length is about 6000 tokens. The output lengths ranges from 180 to 800 tokens. Figure 15 shows the average request latency of Bing Copilot of Parrot and the baselines. Because the LLM service in the baseline system does not know the prompt structure, it is hard to infer the shared prompt from massive LLM requests. Compared to the baseline without sharing prompt, Parrot achieves 1.8× $\sim$ 2.4× speedup for batch sizes of 8 and 16. Further increasing the batch size leads to out-of-memory due to the massive KV cache of shared system prompt. We also build an advanced baseline using vLLM's paged attention to support sharing the prompt with a static prefix. Both Parrot and vLLM use the paged memory management [25], thus both systems can hold the same number of tokens in an LLM engine (A100, LLaMA 7B). Parrot further achieves 1.1× $\sim$ 1.7× speedup over vLLM because of the better GPU kernel. Although vLLM can save extra memory usage of the shared prompt, its GPU kernel still has to reload the tokens repeatedly. Given that the token generation of LLMs is bound by memory bandwidth, such redundant memory loading slows down the end-to-end inference. By combining FlashAttention and PagedAttention, Parrot only needs to load the tokens of the shared prompt once, when computing the attention from the diverged tokens of different users. Parrot's speedup of shared prompt mainly comes from the token generation, thus the longer output length leads to higher improvement. Figure 16 shows Parrot achieves 1.58× and 1.84× speedup compared to vLLM using paged attention, showing 40 *ms* per-output-token latency at a batch size of 32.

In Figure 17, we further evaluated the serving of multiple GPTs applications [42], each of which has multiple users, in



(a) Batch Size = 32

(b) Batch Size = 64

Figure 16: Latency per output token of Bing Copilot.

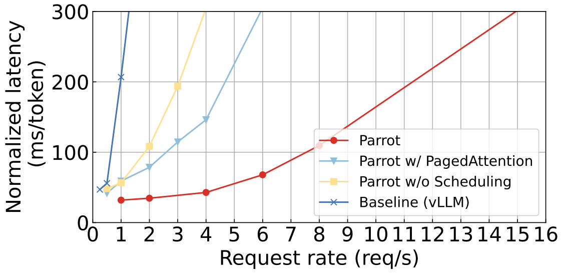
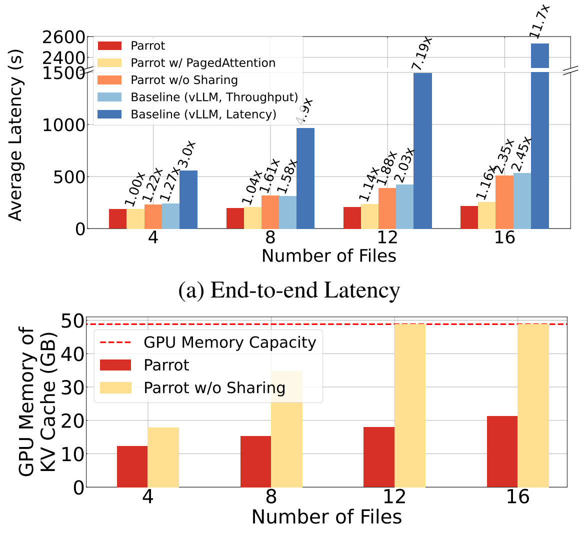


Figure 17: Serving multiple GPTs applications.

a multi-GPU cluster. Four A6000 (48GB) GPUs are deployed with four LLM engines (LLaMA 7B). We select four GPTs applications in four popular categories including productivity, programming, image generation, and data analysis. The LLM requests are randomly generated from the four categories with equal probability. LLM requests arrive at fixed rates following Poisson distribution. Parrot can sustain 12× higher request rates compared to the baseline without sharing. Because the baseline's scheduling policy is not aware of the shared prompt within each LLM application, the requests are mixed in all LLM engines making it impossible to reuse the common prompt prefix. Parrot's scheduling policy co-locates LLM requests of the same applications to maximize the sharing opportunity, achieving both lower inference latency and higher cluster throughput. After turning off such affinity scheduling policy, Parrot only exhibits 3× higher request rates compared to the baseline, because the requests with shared prefix are often dispatched to different engines thus reduced the sharing opportunities. Moreover, Parrot's attention kernel helps Parrot to achieve 2.4× higher rate compared to Parrot using vLLM's PagedAttention, by avoiding the redundant memory loading for attention of shared prompts.

8.4 Multi-agent Applications

We assess the performance of multi-agent systems utilizing MetaGPT [22] within Parrot. A workflow is constructed with three distinct roles. Initially, the Architect outlines the project's file structures and specifies APIs within each file for a given task. Subsequently, multiple Coders undertake the project implementation, with each focusing on a specific file. Following the integration of the code from all files, several Reviewers engage in the process, each examining and commenting on a single file. The Coders then revise their code based on these comments. This review-and-revision cycle is iterated three times to produce the final code. Figure 18 illustrates the latency and memory consumption of Parrot compared to baseline systems on one A100 running LLaMA 13B. Parrot achieves a speedup of up to 11.7× compared with the latency-centric baseline. The primary improvement is attributed to Parrot's capability to deduct the performance objectives for LLM requests based on the end-to-end performance criteria. For this specific multi-agent scenario, the goal is to minimize the time taken to deliver the final code. Parrot identifies multiple task groups within the parallel processes of coding, reviewing, and revising, facilitating larger batch sizes to enhance throughput and reduce the completion time of task groups. We also contrast Parrot with an throughput-centric baseline that uses larger batch on purpose to optimize cluster throughput, which also shows higher concurrency and better completion time than the latency-centric baseline.



(a) End-to-end Latency

(b) GPU Memory of KV Cache

Figure 18: The latency and memory usage for multi-agent programming, with varying number of files to program.

Even when compared to the throughput-centric baseline, Parrot demonstrates superiority, being faster by up to 2.45×. This enhancement mainly stems from Parrot's ability to decrease redundancy through its prompt structure analysis, which contributes a 2.35× acceleration. Given the interactive nature of the roles in MetaGPT, there is considerable overlap in the context among different roles, which Parrot capitalizes on by sharing this common context as a prompt prefix. The static prefix sharing mechanism from vLLM does not work in this dynamic scenario. Without a grasp of the prompt's structure, it cannot identify dynamically generated Semantic

Variables that could also be shared during runtime. As depicted in Figure 18b, Parrot without this sharing capability would hit the GPU memory ceiling. Additionally, Parrot's specialized GPU kernel for processing the shared prefix achieves a further 1.2× speedup when there are 16 files, compared to using vLLM's PagedAttention, due to the reduced memory transactions.

8.5 Scheduling of Mixed Workloads

To assess the performance of Parrot on a multi-GPU setup, we configure a cluster with four A6000 (48GB) GPUs, each hosting a separate LLM engine (LLaMA 7B), resulting in a total of four LLM engines. We emulate a real-world scenario where LLM services encounter a variety of demands by injecting a mix of requests from chat applications at a rate of 1 req/s and from data analytic tasks (i.e., map-reduce applications) previously analyzed in §8.2. Requests from the chat applications are characterized by their need for low latency, whereas the map-reduce applications prioritize high throughput, creating a challenge when they are concurrently processed by the same LLM engine. We benchmark Parrot against two reference implementations: one tailored for latency, limiting engine capacity to reduce decoding time, and another for throughput, utilizing full engine capacity to maximize GPU utilization.

The results depicted in Figure 19 demonstrate that Parrot attains a 5.5× and 1.23× improvement in normalized latency (measured as request latency per number of output tokens) [25, 56] for chat applications in comparison to the latency-focused and throughput-focused baselines, respectively. In terms of token generation speed for chat applications, Parrot delivers performance on par with the latency-centric baseline and outperforms the throughput-centric baseline by 1.72×. For map-reduce applications, Parrot reaches a 3.7× speedup over the latency-centric baseline and is 1.05× more efficient than the throughput-centric baseline. Parrot excels by providing both low latency for chat applications and high throughput for map-reduce applications. It mitigates the contention between chat and map-reduce workloads by intelligently scheduling them on separate engines. These findings underscore the significance of specialized handling for diverse requests to enhance the overall performance of LLM services.

9 Related Works

Deep Learning Serving Systems. The field of model serving has seen a surge of research activity in recent years, with many systems developed to address the different challenges of deep learning model deployment. The systems include Clipper [10], TensorFlow Serving [39], Clockwork [19], REEF [20], AlpaServe [28], which have explored many aspects including batching, caching, placement, scheduling, model parallelism for the serving of single or multiple models.

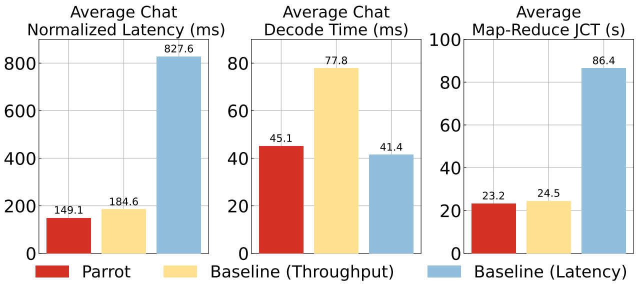


Figure 19: The mixture of chat and map-reduce applications.

These systems were proposed for serving general deep learning models, which have less consideration about the unique requirements of large language models, e.g., autoregressive decoding. Orca [56] proposed a fine-grained scheduling mechanism that can batch multiple LLM requests at the iteration level, which is also known as continuous batching. vLLM proposes PagedAttention [25] allows the batching of LLM requests with different lengths using non-contiguous memory, increasing memory utilization. These systems for LLM serving still treat LLM requests separately, missing the opportunities to understand the interconnections within an application and exploit the commonality of different requests. Parrot is orthogonal to them. With more application-level knowledge exposed by Semantic Variables, Parrot can do data flow analysis on LLM requests, which enables a brand new optimization space with the final goal of optimizing the end-to-end performance of applications, rather than individual requests.

LLM Orchestrator Frameworks. LLM orchestration frameworks help developers create and manage applications powered by LLMs. They simplify the process of prompt design, and orchestration of multiple LLM requests, which enable developers to interact with LLMs easily. LangChain [8] is a Python framework that provides many workflow patterns, e.g., chain, map-reduce so that developers can easily customize their own LLM applications. Semantic Kernel [36] introduces Planners are semantic agents that can automatically generate plans based on the needs of the users. PromptFlow [35] supports chains of native and semantic functions and visualizes them as a graph. LlamaIndex [29] allows developers to use natural language queries to retrieve relevant documents. Parrot is orthogonal to these frameworks and can be easily integrated with these frameworks to support Parrot's APIs with Semantic Variable abstraction, as discussed in §6.

DAG-aware System Optimizations. Dependency graphs or DAGs (Directed Acyclic Graphs) widely exist in many kinds of systems, and many optimizations have been proposed to optimize the systems by exploiting the DAG information. Tez [4], Dryad [23], and Graphene [16] use the task dependency to optimize the scheduling and packing of parallel data analytic workloads. SONIC [30], Caerus [59], and Orion [31] optimize serverless functions from the aspects of communication, latency, and cost. Parrot learns from the previous system works and realizes the importance of correlations of LLM requests to optimize the end-to-end performance of LLM applications. This motivates Parrot to build APIs for exposing such dependency information. Moreover, it is unique to LLM applications to understand the prompt structure in addition to request-level dependency, which is necessary for communication and identifying commonality across LLM requests. This motivates us to propose the Semantic Variable abstraction, instead of just using a DAG of requests.

10 Conclusion

This paper proposes Parrot that treats LLM applications as first-class citizens and targets to optimize the end-to-end performance of LLM applications, instead of only optimizing individual LLM requests. We propose Semantic Variable as the key abstraction that exposes the dependency and commonality of LLM requests, enabling a new optimization space. Our evaluation shows Parrot can optimize LLM-based applications by up to 11.7×. We envision this new angle of efficiency improvement of LLM applications brings a broad future direction to study other scheduling features like the fairness of *end-to-end* performance of LLM applications.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their constructive feedback and suggestions. Zhenhua Han, Yuqing Yang and Chen Chen are the corresponding authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[4] Apache. Tez. https://tez.apache.org/, November 2019.

[5] Apache. Kafka. https://kafka.apache.org/, October 2023.

[6] Zhengda Bian, Hongxin Liu, Boxiang Wang, Haichen Huang, Yongbin Li, Chuanrui Wang, Fan Cui, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. *CoRR*, abs/2110.14883, 2021.

[7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[8] Harrison Chase. LangChain. https://github.com/langchain-ai/langchain, October 2022.

[9] Lequn Chen. Dissecting batching effects in gpt inference. https://le.qun.ch/en/blog/2023/05/13/transformer-batching/, May 2023.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[13] Bill Gates. Ai is about to completely change how you use computers and upend the software industry. https://www.gatesnotes.com/AI-agents, Nov 2023.

[14] Google. Google bard. https://bard.google.com/, Nov 2023.

[15] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[16] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware scheduling for Data-Parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[18] guidance ai. Guidance. https://github.com/guidance-ai/guidance, November 2023.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[21] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[22] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 59–72, New York, NY, USA, 2007. Association for Computing Machinery.

[24] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 642–657, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[27] Yucheng Li. Unlocking context constraints of llms: Enhancing context efficiency of llms with self-information-based content filtering, 2023.

[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[29] Jerry Liu. LlamaIndex, November 2022.

[30] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[31] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 22), pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[32] Microsoft. Bing chat. https://www.bing.com/chat, Nov 2023.

[33] Microsoft. Meeting recap in microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/premium, May 2023.

[34] Microsoft. Microsoft 365 copilot. https://www.microsoft.com/en-us/microsoft-365/enterprise/microsoft-365-copilot, Mar 2023.

[35] Microsoft. PromptFlow. https://github.com/microsoft/promptflow, November 2023.

[36] Microsoft. Semantic Kernel. https://github.com/microsoft/semantic-kernel, November 2023.

[37] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[38] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[39] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[40] OpenAI. Chatgpt. https://chat.openai.com/, Nov 2023.

[41] OpenAI. Gpt-4 technical report, 2023.

[42] OpenAI. Introducing gpts. https://openai.com/blog/introducing-gpts, Nov 2023.

[43] OpenAI. OpenAI Triton. https://github.com/openai/triton, November 2023.

[44] OpenAI. Production best practices - openai api. https://platform.openai.com/docs/guides/production-best-practices/improving-latencies, Nov 2023.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[46] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[47] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[48] Sebastián Ramírez. FastAPI. https://github.com/tiangolo/fastapi.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[50] ShareGPT Team. Sharegpt dataset. https://sharegpt.com/, Nov 2023.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[52] Unknown. Prompt of bing chat. https://www.make-safe-ai.com/is-bing-chat-safe/Prompts_Conversations.txt, Nov 2023.

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020.

[54] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[55] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv*, 2023.

[56] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[57] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 265–278, New York, NY, USA, 2010. Association for Computing Machinery.

[58] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, 2008.

[59] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021.

[60] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[61] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[62] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[63] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

[64] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

USHER: Holistic Interference Avoidance for Resource Optimized ML Inference

Sudipta Saha Shubha
University of Virginia

Haiying Shen
University of Virginia

Anand Iyer
Georgia Institute of Technology

Abstract

Minimizing monetary cost and maximizing the goodput of inference serving systems are increasingly important with the ever-increasing popularity of deep learning models. While it is desirable to spatially multiplex GPU resources to improve utilization, existing techniques suffer from inter-model interference, which prevents them from achieving both high computation and memory utilizations. We present USHER, a system that maximizes resource utilization in a holistic fashion while being interference-aware. USHER consists of three key components: 1) a cost-efficient and fast GPU kernel-based model resource requirement estimator, 2) a lightweight heuristic-based interference-aware resource utilization-maximizing scheduler that decides the batch size, model replication degree, and model placement to minimize monetary cost while satisfying latency SLOs or maximize the goodput, and 3) a novel operator graph merger to merge multiple models to minimize interference in GPU cache. Large-scale experiments using production workloads show that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency compared to existing methods, while scaling to thousands of GPUs.

1 INTRODUCTION

Driven by the breakthroughs achieved by Deep Learning (DL) models in a wide variety of domains [1–3], machine learning (ML) inference has emerged as the dominant workload that underpins many real-world applications. Our quest for improving the capability and accuracy of DL models has led to models growing in size rapidly [4]. While the success of DL models has been celebrated, it has come with a significant monetary cost: the increase in model sizes and popularity of ML model-based applications demand the use of expensive and power-hungry GPUs, leading to ML inference accounting for more than 90% of production costs [5]. Forecasts paint a gloomy picture: annual data center infrastructure and operating costs are projected to increase to over $76 billion by 2028 due to the rapid increase of the number of GPUs in the data centers, which is more than twice the estimated annual operating cost of Amazon AWS [6].

The exorbitant operating cost requirement has led to several systems innovations; state-of-the-art ML inference systems incorporate several optimizations that increase the *utilization* of GPUs. The fundamental technique to improve the utilization of a GPU is to use *batching*, where multiple inputs are combined and passed together through the model. Batching inputs together results in an increase in the compute requirements and thus improves the utilization of the GPU, albeit at the expense of increased latency. Unfortunately, batching is insufficient to optimally utilize a GPU because it is a *single knob* that influences two GPU resources: memory and compute, and thus is unable to saturate both resources at the same time. Moreover, since real-world batch sizes are not continuous in nature, there is no fine-grained control over the resources—while one batch size may severely underutilize the GPU in terms of memory or compute, the next possible batch size may not fit in the given resources or violate the strict latency Service-Level-Objective (SLO) (§2.1). When combined with the fact that request rates vary over time [3], real-world deployments have reported low GPU utilization averaging between 25% to below 50%, which has become a thorny pain point in reducing the total operational cost of large GPU clusters [7–10].

A natural solution to this problem is to place and simultaneously execute multiple models in a GPU. Unfortunately, previous research works have shown that this could result in *interference* between models due to resource contention which could introduce significant increase in inference latency, thus leading to SLO violation [11, 12]. An alternative is to leverage virtualization technologies that can divide the GPU resources; sadly GPU virtualization technologies available today are rudimentary and inflexible at best. NVIDIA Multi-Process Service (MPS) [13] facilitates simultaneous execution of multiple spatially multiplexed models by logically partitioning the computation space of the GPU among the models. Several scheduling systems [4, 14] have been proposed that leverage MPS and decide how much GPU computation space to allocate to each model to satisfy the latency SLO of the model requests based on offline profiling and place the models to the GPUs in such a manner that maximizes the utilization of the computation space of the GPUs. However, these works solely focus on compute utilization and largely overlook the maximization of memory space utilization. NVIDIA multi-instance GPU (MIG) addresses the problem of inter-model interference by physically partitioning the computation and memory spaces of the GPU [15]. However, MIG only provides rigid partitioning, leading to GPU overprovisioning, and hence underutilization, and is only available in the latest generation of GPUs.

In this paper, we propose USHER, an end-to-end inference serving system that optimizes both GPU computation and memory utilizations by spatially multiplexing its resources in an interference-aware fashion. We design USHER from first-principles, based on a systematic analysis of performance and interference characteristics of the state-of-the-art solutions on real-world data traces (§2). Our analysis reveals several key observations. First, we may need to divide the workload of a model into multiple GPUs even when one GPU is enough to complete the workload within the latency SLO. Also, we need to perform this workload division holistically for all models. Second, not only the model parameter size, but also the intricate relationship between batch size, batch size-dependent resource requirements, and SLO contributes to making a model computation-heavy or memory-heavy. USHER leverages such observations in designing its three key components.

To accurately estimate the computation and memory requirements of a new model without incurring the prohibitive cost of offline profiling, USHER proposes a novel low-level GPU kernel analysis-based approach that first estimates which GPU kernels of the model will be executed concurrently and then sums up the resource requirements of those kernels. Finally, it takes the maximum sum across all sets of concurrent kernels as the highest resource requirement of the model (§3.2).

Based on the estimation, USHER needs to decide on the placement of each model that maximizes resource utilization without interference in both computation and memory spaces. Towards this, USHER incorporates a novel variant of a multi-dimensional bin packing scheduler [16–20]. To address the exponential complexity of holistic workload division, the scheduler first creates moderate-sized groups of models to maximize the probability of spatially multiplexing computation-heavy models with memory-heavy models within a group. Then, the scheduler decides the optimal workload division, batch size, and GPU placement decisions holistically for all models within a group by a heuristic algorithm (§3.3).

Finally, to minimize the cache interference among multiple spatially multiplexed models, USHER proposes a novel method that merges the computation graphs of multiple DL models to maximize the usage of GPU cache contents. Existing works [21–23] on computation graph merging reduce memory requirements by sharing weights across multiple models, which cannot maximize GPU cache usage. To this end, USHER merges the graphs in a manner that when the weight submatrix that is similar across different models is present in the GPU cache, the matrix multiplications of the different models associated with the submatrix are performed at the same time (§3.4).

We implemented USHER on Tensorflow (§4) and evaluated it on a wide variety of models and workloads using both real testbed and simulations. Our evaluation shows that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency against Shepherd [3], GPUlet [14], and AlpaServe [4], three representative state-of-the-art baselines (§5).

Overall, we make the following contributions in this paper:

1. We systematically analyze the underutilization of resources and inter-model interference in the state-of-the-art inference serving systems.
2. We propose USHER, a system that spatially multiplexes the inference serving of multiple DL models in an interference-aware and resource utilization-maximizing manner.
3. We evaluate USHER against the state-of-the-art baselines and show that it significantly outperforms them.

Table 1: DL models used in experiments.

Task & Domain	Model Name	Number of Parameters	Latency SLO	Dataset
Object detection (CNN-based vision models)	YOLO-v3 [24]	8.8M	197ms	COCO [25]
	R-CNN [26]	42M	284ms	
	MobileNetSSD-v2 [27]	15M	93ms	
Object recognition (CNN-based vision models)	ResNet-50 [28]	24M	108ms	ImageNet [29]
	ResNet-101 [28]	44M	198ms	
	ResNeXt-50 [30]	25M	116ms	
	ResNeXt-101 [30]	89M	407ms	
	SqueezeNet [31]	0.42M	14ms	
	ShuffleNet-v2 [32]	2M	40ms	
	MobileNet-v2 [33]	3.4M	64ms	
	DenseNet-121 [34]	7.6M	202ms	
	DenseNet-201 [34]	14.1M	405ms	
	Inception-ResNet-v2 [35]	56M	439ms	
	Inception-v3 [36]	25M	116ms	
	Inception-v4 [35]	43M	204ms	
	EfficientNet-B7 [37]	66M	217ms	
Language translation (Transformer-based language model)	GNMT [38]	278M	66ms	WMT 2019 [39]
Text classification (Transformer-based language model)	BERT [40]	110M	35ms	IMDB Movie Review [41]
Text generation (Transformer-based language models)	GPT-2 [42]	1.5B	140ms	WikiText [43]
	Llama-2 [44] (Large model)	13B	834ms	

2 Experimental Analysis

We use *Cuti* and *Muti* to denote GPU computation and memory utilization, respectively, and use *Creq* and *Mreq* to denote their requirement from a model. Creq (or Mreq) of a model is the highest percentage of the total computation (or memory) space of a GPU consumed by the model at any point during its execution. We further use *Rreq* to denote the sum of *Mreq* and *Creq*. We use *C-heavy* and *M-heavy* to denote computation-heavy and memory-heavy, respectively. We use

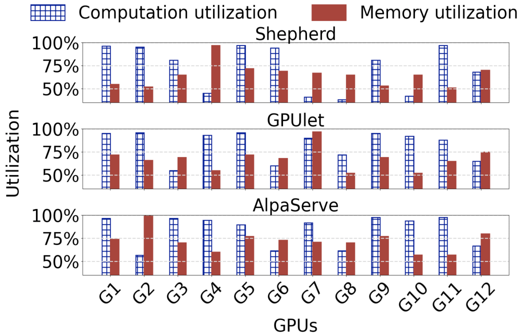


Figure 1: Performance of existing systems.

GPU# to denote the total number of GPUs required to satisfy the SLOs, and use *model#* to denote the number of models in a GPU. We use *C-space* and *M-space* to denote the available capacity of a GPU in computation and memory, respectively.

As [3, 14], we conducted analytical experiments using a mix of convolutional neural network (CNN) and Transformer models, which are typically the most widely used DL models in production deployments [3]. The 20 models used in our experiments are shown in Table 1. We got the trained models from HuggingFace repository [45]. The inference requests are also taken from the datasets. As [4, 14], the SLO of a model is taken as double the average inference latency of a single request in a Nvidia V100 GPU. As [3], the experiments were conducted on a GPU cluster formed by 12 Amazon EC2 servers of type p3.2xlarge. Each server has one Nvidia V100 GPU with 5760 computation cores, 16 GB GPU memory, one 2.3 GHz processor with 8 CPU cores, and 61 GB host memory. For the large model Llama-2, we utilized the DeepSpeed library [46] to partition the model into multiple partitions, allowing each partition to be loaded onto a GPU. Throughout the remainder of the paper, when referring to a model, it denotes the entire model for small models and a model partition for large models. Also, multiplexing refers to spatial multiplexing. As [3,4], we used the Microsoft Azure Function trace 2019 [47] for the inference request rates and assigned the 46,000 function streams from the trace to the 20 models in a round-robin manner.

2.1 Performance of Existing Systems

We used Shepherd [3] to represent systems that do not allow spatial multiplexing and use GPUlet [14] and AlpaServe [4] to represent systems that allow spatial multiplexing. They aim to maximize GPU computation utilization and goodput. Goodput is defined as the number of inference requests completed within their latency SLOs per unit time.

Fig. 1 shows the average Cuti and Muti of each GPU. Shepherd achieves 41%-97% and 51%-97% Cuti and Muti, respectively. Though Shepherd uses batching to increase utilization (Ruti), it also increases the inference latency and memory requirement, which may become a bottleneck and limit the Cuti. GPUlet and AlpaServe increase the Cuti and Muti to 55%-97% and 54%-99%, respectively, due to their spatial multiplexing. However, there is still room for improvement. Also, Shepherd, GPUlet, and AlpaServe produce 14.2%-52%, 7%-40.1%, and 5.3%-44.9% $|Cuti - Muti|$ values, respectively. This is because the Creq and Mreq of a model are not necessarily correlated and hence, maximizing Cuti does not necessarily maximize Muti. Next, to study interference among models, we measured the goodput of each model with and without multiplexing in GPUlet and AlpaServe, and calculated their ratios. Fig. 2 shows the CDF of models versus the ratio. We see that 87% and 100% of the models have a ratio $\leq$ 0.55 in GPUlet and AlpaServe, respectively, meaning their goodputs are decreased by almost half due to the interference. GPUlet tries not to place the models in one GPU if their interference estimated by a regressor is high. However, it does not capture the interference among three or more models, and also does not address the cache interference problem.

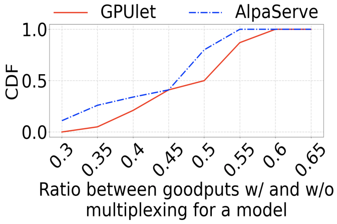


Figure 2: Impact of inter-model interference on goodput.

Observation 1. *The existing inference serving systems cannot maximize Cuti or Muti, and their model multiplexing significantly reduces the goodput due to model interference. In addition, maximizing Cuti does not necessarily maximize Muti.*

2.2 Opportunity of Workload Division

By equally dividing the workload (i.e., number of incoming requests per unit time) of a model into multiple GPUs, we essentially replicate the whole model in those GPUs. We use replication degree (RD) to denote the number of replicas of a model. In the example in Fig. 3a, models M1, M2, and M3 have 70%, 65%, and 40% Mreq (with 10%, 5%, and 10% for parameters, and the rest for intermediate data), respectively, to meet their SLOs with BS (batch size) = 8. Hence, the average Muti equals $\frac{70\%+65\%+40\%}{3} = 58.33\%$. Though the three GPUs have 30%, 35%, and 60% of the GPU memory unused, they are not enough to host any other model. Now, due to the strict latency SLO requirement, it is not possible to increase the BS of any model any further to increase the memory utilization since increasing the BS also increases the per-batch inference latency. Reducing the BS of M1 by half essentially conducts a workload division and lowers the intermediate data amount by half in each GPU where M1 is hosted, resulting in 40% Mreq in such a GPU. Fig. 3b shows a multiplexing schedule. The total number of GPUs (GPU#) is still 3 with 61.66% average Muti. Fig. 3c shows another multiplexing schedule, which performs workload division also for M2 and M3, and results in 2 GPU# and 100% Muti. This example shows that to increase Ruti in multiplexing, we may need to divide the workload to more GPUs than the minimum required and we need to decide the optimal workload division schedule *holistically* (instead of independently) for all models.

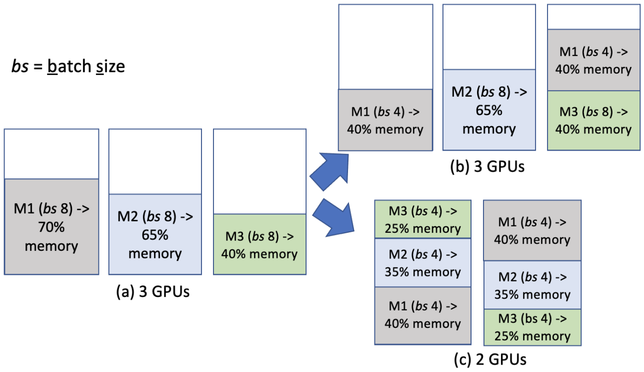
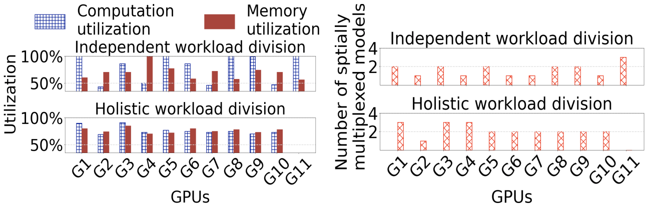


Figure 3: Performing workload division holistically for all models increases resource utilization.



(a) Resource utilization of each GPU by workload divison.

(b) Number of spatially multiplexed models in each GPU using workload division.

Figure 4: Effectiveness of holistic workload division.

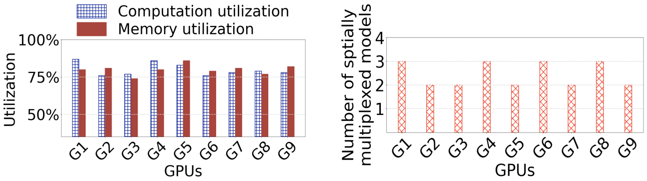
We then experimented to test the impact of workload division. We first used Shepherd to decide the BS (b_i^s) and the minimum RD (r_i^d) of each model i to complete its workload within the SLO. Then, for each model i independently, we created $2r_i^d$ replicas of each model with BS=$\frac{b_i^s}{2}$ and placed them to randomly selected GPUs with enough available C-spaces and M-spaces to host the model replicas. In our next experiment, to perform holistic workload division, we created all possible configurations, i.e., {(BS, RD) for each model i within range $[b_i^s, \frac{b_i^s}{2}]$ and $[r_i^d, 2r_i^d]$}. Then, for each configuration, we placed the model replicas to randomly selected GPUs having enough resources. Finally, we chose the best configuration that resulted in the lowest GPU#. Fig. 4 shows the average Cuti and Muti, and model# in each GPU for both experiments. Independent workload division increases the average Cuti and Muti by 3.5% and 3.8%, respectively, compared to Shepherd shown in Fig. 1 and the GPU# is decreased from 12 to 11. The holistic workload division further increases the average Cuti and Muti by 4.7% and 5.1%, respectively, leading to another decrease of GPU# by 1, even with a simple strategy of random GPU placement.

Observation 2. *In spatial multiplexing-based inference serving, unlike existing systems, we may need to divide the workload of a model even when one GPU is enough to complete the workload within SLO in order to increase the overall resource utilization.*

Observation 3. *Optimal workload division should not be decided independently for each model. Instead, a holistic approach that considers all models simultaneously is essential.*

2.3 Study on C-heavy and M-heavy Models

In this experiment, we did the same holistic workload division experiment described above, except that, for each configuration, we first ordered the models in descending order of Creq+Mreq and followed this order of models to place the model replicas to randomly selected GPUs with enough spare resources. Fig. 5 shows the results. The average Cuti and Muti increase by 5.3% and 4.9%, respectively, compared to the holistic approach in Fig. 4a, leading to a further decrease of GPU# by 1. Also, the average model# in a GPU increases by 0.24 compared to the holistic approach in Fig. 4b. Due to the ordering of the models, less spare resources remain in the GPUs after the model placement, leading to less resource fragmentation and better utilization.



(a) Resource utilization of each GPU by sorting the models according to total resource requirements.

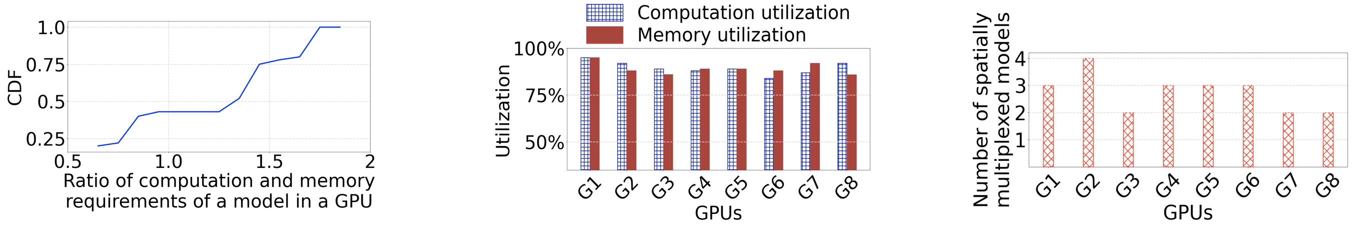
(b) Number of spatially multiplexed models in each GPU using the sorting approach.

Figure 5: Considering a model's total resource requirement in model replica placement.

The general notion assumes large models have high Creq and Mreq, while small models have low Creq and Mreq. However, this distinction overlooks the impact of BS in inference serving. Increasing BS may boost resource use but risks latency violations and memory overflow. This highlights the delicate balance of Cuti and Muti in workload scheduling.

For example, when we executed LlaMA-2 (with 13 billion parameters) with BS=4 in a Nvidia H100 GPU, it takes up almost all GPU memory, but has 45% Cuti unused. Increasing BS any further overflows the memory. Hence, it is M-heavy instead of C-heavy, despite being a large model. On the other hand, MobileNetV2 (only 3.4 million parameters) with BS=128 reaches up to 93% Cuti but has 30% Muti. Increasing BS any further violates its 64ms SLO. Hence, it is a C-heavy model instead of M-heavy model.

Fig. 6a shows the CDF of models versus the ratio $Creq/Mreq$ of a model. We see that 22% of the models have ratios $\leq$0.75, indicating they are M-heavy. Also, 28% of the models have ratios in (1.35,1.65], indicating they are C-heavy. Llama-2 is an M-heavy model (i.e., Mreq/Creq$\geq$1.2). Among the other small models, 39% are M-heavy, 2% have comparable Creq and Mreq, and the rest are C-heavy (i.e., Creq/Mreq$\geq$1.2).

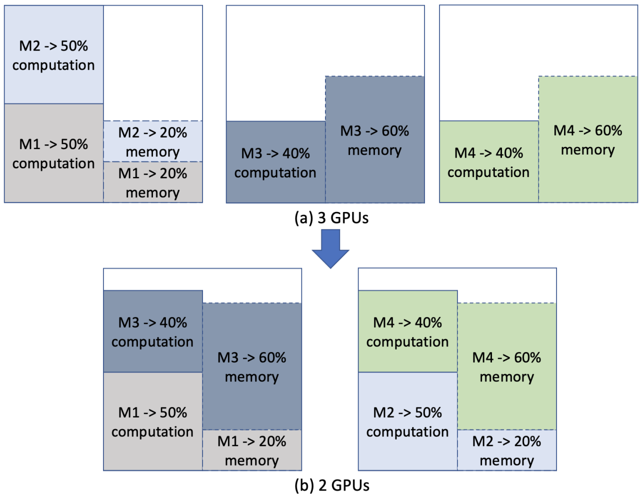


(a) CDF of models vs. Creq/Mreq.

(b) Resource utilization by multiplexing C-heavy and M-heavy models.

(c) Model# by multiplexing C-heavy and M-heavy models.

Figure 6: Computation-heavy and memory-heavy models.



(a) 3 GPUs

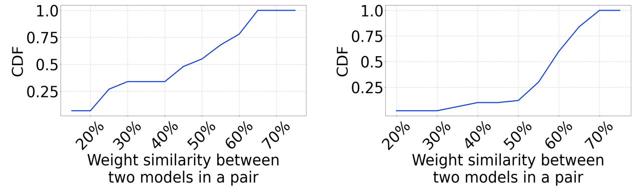
(b) 2 GPUs

Figure 7: Spatially multiplexing a computation-heavy model with a memory-heavy model increases resource utilization.

> **Observation 4.** *Unlike common belief, model parameter size alone cannot dictate whether a model is C-heavy or M-heavy. Even a small model can surpass a larger one in Creq, driven by the intricate relationship between BS, BS-dependent resource needs, and SLO.*

We next investigate whether multiplexing a C-heavy model with an M-heavy model improves Ruti. Let us consider 4 models M1, M2, M3, and M4 in Fig. 7 with BS=32. Their Creq are 50%, 20%, 40%, and 40%, and their Mreq are 20%, 20%, 60%, and 60% respectively. Hence, M1 and M2 are C-heavy and M3 and M4 are M-heavy. First, we multiplex M1 and M2 in the same GPU as shown in Fig. 7a, which results in 100% Cuti but 40% Muti. M3 and M4 cannot be multiplexed due to lack of memory, resulting in 3 GPU#, 60% average Cuti and 53.33% average Muti. Alternatively, if we multiplex M1 with M3 and M2 with M4 as shown in Fig. 7b, it results in 2 GPU#, 90% average Cuti and 80% average Muti. Hence, multiplexing a C-heavy model with a M-heavy model maximizes resource utilization.

To experimentally validate this, we did the same experiment described in §2.3, except that we interleave C-heavy models with M-heavy models in the ordered list. Figs. 6b and 6c show the results. The average Cuti and Muti increase by 12.1% and 11.8%, respectively, compared to the holistic approach in Fig. 4a, Also, the average model# increases by 0.6 compared to the holistic approach in Fig. 4b, and hence, GPU# is decreased by 2.



(a) Weight similarity across CNN models.

(b) Weight similarity across Transformer models.

Figure 8: Weight similarity across models.

> **Observation 5.** *Multiplexing C-heavy model with an M-heavy model increases both Cuti and Muti of a GPU.*

2.4 Models' Weight Overlapping

Previous studies [48, 49] have indicated that CNN models have significant weight overlapping (i.e., similar parameter values) in earlier layers because the first few convolutional layers function as feature-extractors [48] and are task-agnostic. Also, to reduce training cost, the task-specific transformer models are typically trained not from scratch, but from pre-trained task-agnostic foundation models using transfer learning. Additionally, for both models, people generally customize a task-specific pretrained model for their own datasets by fine-tuning only the last few layers. These indicate the potential for weight similarity across models. Motivated by these, we evaluated the weight similarity across models. Specifically, for each pair of CNN models and Transformer models, we measured the weight similarity between the two models in the pair.

Given CNN models A and B, we first find the weight similarity between every possible pair of convolutional layers across the two models. Then, we take the average weight similarity of all pairs of layers as the weight similarity between the two models. To find the weight similarity between layer $i \in L_A$ and layer $j \in L_B$, where L_x denotes the set of all convolutional layers in model x, we calculated $\frac{2\times|\max(W_A^i \cap W_B^j)|}{|W_A^i|+|W_B^j|}$, where $\max(W_A^i \cap W_B^j)$ denotes the longest common submatrix between weight matrices W_A^i and W_B^j. We consider two weight values as the same if their absolute difference is very low (i.e.,

$\leq 10^{-7}$). We did the same experiment for the attention layers of the Transformer models.

Fig. 8 shows the results for randomly chosen 1k CNN and 1k Transformer models from HuggingFace trained model repository [45]. For the CNN models, 52% model pairs have weight similarity within $(45\%, 65\%]$. For the Transformer models, 70% model pairs have weight similarity within $(55\%, 70\%]$ in between themselves.

Observation 6. *There are significant weight overlappings between different CNN models, and also between different Transformer models.*

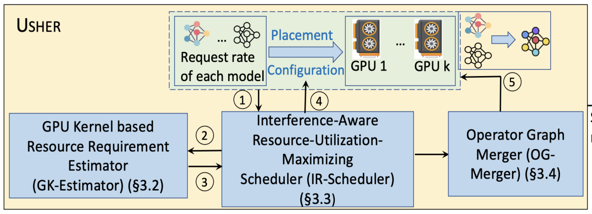

USHER
Request rate of each model
Placement
Configuration
GPU 1
...
GPU k
1
2
3
4
5
6
GPU Kernel based Resource Requirement Estimator (GK-Estimator) (§3.2)
Interference-Aware Resource-Utilization-Maximizing Scheduler (IR-Scheduler) (§3.3)
Operator Graph Merger (OG-Merger) (§3.4)
Serving requests

Figure 9: System overview of USHER.

3 System Design of Usher

3.1 Overview

Observation(O)1 motivates us to propose a new system to maximize both computation and memory utilizations of GPUs in an interference-aware manner to minimize the inference serving cost. We design USHER based on O2-O6.

Given models with request rates and a cluster of GPUs (that may be heterogeneous), USHER decides the *schedule* that includes the configuration (BS,RD), GPU allocation, and placement of the model replicas of each model. We consider two scenarios with different goals in this paper: (i) non-fixed cluster, where USHER aims to minimize the monetary cost while satisfying the SLOs of all models [12, 14, 50], and (ii) fixed-cluster, where USHER aims to maximize goodput [3, 4].

USHER has following major methods as shown in Fig. 9.

(1) **GPU Kernel based Resource Requirement Estimator (GK-Estimator)**(§3.2). The estimator quickly and correctly calculates the Creq and Mreq of a model in a GPU type based on a given configuration.

(2) **Interference-Aware Resource-Utilization-Maximizing Scheduler (IR-Scheduler)**(§3.3). Instead of solving an optimization problem, which has high complexity, USHER provides a lightweight heuristic to quickly derive the schedule. It first groups the models in a manner that maximizes the opportunity of multiplexing C-heavy and M-heavy models within a group. Then, within each group, it chooses the configuration that results in the placement with the best performance regarding the specific goal (by leveraging O2-5). The IR-Scheduler ensures each model replica gets its Creq and Mreq in the GPU where it is placed, thus ensuring there is no inter-model interference in C-space and M-space.

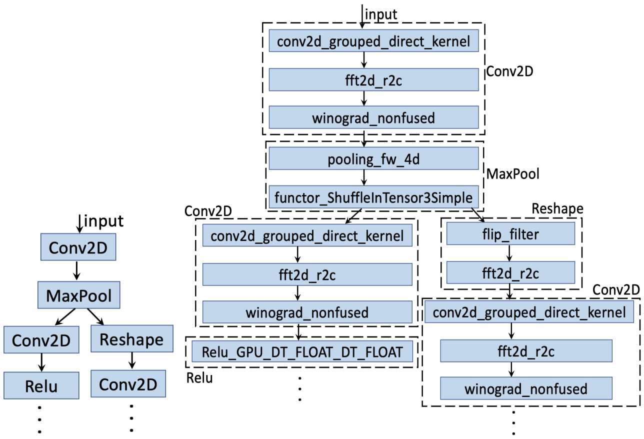
(3) **Operator Graph Merger (OG-Merger)**(§3.4). After deciding the placement, OG-Merger merges as many operator graphs of the models assigned to a GPU as possible to minimize inter-model interference in the GPU cache. After the merging, the merged graph is allocated the sum of the resources allocated to the models (decided by the IR-Scheduler) whose graphs have been merged. Note that the IR-Scheduler does not satisfy the cache requirements of the models, which we found to be almost 100% in the setup in §2. Hence, satisfying the cache requirement results in underutilization of C-space and M-space. That is why USHER addresses cache interference separately in OG-Merger.

In USHER, the input to the IR-Scheduler includes the request rate (i.e., workload) of each model and the types of GPUs in the system (①). During decision making, IR-Scheduler uses GK-Estimator (②) to estimate Creq and Mreq given a configuration (③), and finally outputs the optimal schedule (④). Then, OG-Merger merges the models placed to the same GPU (⑤). Then, the system starts serving the inference requests. As [14, 50], when a model's workload changes significantly, i.e., by 0.5k requests/second, (which may happen after 45s-300s as shown in §5.4), USHER is used again to adapt to the new workload pattern.

3.2 Kernel-based Resource Requirement Estimator

Offline profiling is a common approach for estimating the Creq and Mreq of each new model [3, 4, 12, 14, 50]. However, it is costly and time-consuming. To address this challenge, we propose the GK-Estimator that estimates the resource requirement of each model independently by analyzing its low-level GPU kernels without actually running the model in a GPU. We use Mreq as an example to explain how GK-Estimator works. Every model can be treated as a computational graph, in which a node is an operator and an edge is a tensor (i.e, multi-dimensional matrix) denoting model input or intermediate data generated by a model layer. Internally, each operator execution involves sequentially calling one or more GPU kernel APIs defined in the GPU programming framework (e.g., CUDA for Nvidia GPUs, ROCm for AMD GPUs). For each operator defined in ONNX [51], we found which GPU kernels are called by an ML framework during the execution of the operator by using Nvidia Profiling tool [52]. We noticed that 1, 2, 3, and 4 GPU kernels are called for 2%, 8%, 56%, and 34% of the operators, respectively.

The operators of a new model are usually from a pre-known set of operators [53, 54]. Therefore, GK-Estimator uses a regression model that quickly computes the memory required by the intermediate data generated by an operator based on the sizes of the input tensors and mathematical operations of the operator. Then, for a sequential DL model, its Mreq is the sum of the model parameter size and the highest memory required by an operator. The memory requirement for a model with parallel branches (e.g., Inception model, illustrated in Fig. 10a), the Mreq of the intermediate data is the sum of



(a) Operator-level computation graph **(b) Kernel-level computation graph**

Figure 10: Conversion of an operator-level computation graph for a CNN to its kernel-level computation graph.

Mreqs of the intermediate data from kernels that are executed concurrently. Then, as shown in Fig. 10, first, GK-Estimator converts the operator-level computation graph to a kernel-level computation graph by replacing each operator with the sequence of GPU kernels it calls. This sequence is found offline for each operator in ONNX. Then, it finds each set of kernels that will be executed concurrently. Next, for each set, it estimates the Mreq of the intermediate data generated by each kernel of the set by a regression model (called Mreq-Regressor) and then sums up the Mreqs of all the kernels in the set. Finally, it takes the sum of the model parameter size and the maximum memory requirement from all sets as the Mreq of the model.

The start time of the first kernel of the model (i.e., the kernel that directly gets the model input) is 0^{th}ms. To identify which kernels will be executed concurrently, GK-Estimator first finds the start time of each kernel. It uses another regression model to estimate the execution time duration of each kernel (called Time-Regressor). The kernels with a start time difference of no more than τms (e.g., 0.001ms) are considered to be potentially running concurrently.

To build Mreq-Regressor and Time-Regressor, we use a stacked model (a combination of lasso regression, kernel ridge regression, gradient boost regression, and XGBoost regression models) for higher accuracy [55]. The inputs to both the regressors include the following features that impact the resource requirement and execution time duration of a kernel: batch size, sizes of the kernel's parameter weight tensor and its input intermediate data tensor, the number of floating-point operations of the kernel, and GPU type. Note that none of the features depend on the kernels of the other models that the IR-Scheduler may potentially place to the same GPU. We trained the regression models offline using all the GPU kernels for all the operators defined in ONNX. The training takes 1.3hr in a V100 GPU. We conducted an experiment and found that this kernel analysis approach achieves 99.98% accuracy in estimating the Creq and Mreq of the models in Table 1 with BS 2. We also measured the accuracy of each of the constituting regression models in the stacked model and found that they provide 23%-40% less accuracy compared to the stacked model.

3.3 Interference-aware and Resource Utilization-maximizing Scheduler

USHER first groups the models such that the models inside a group are highly probable to be multiplexed in an interference-aware and resource utilization-maximizing manner. Then, inside each group, based on O3, USHER decides the GPU allocation and placement decisions *holistically* for all models of the group. Below, we first describe the grouping process (§3.3.1) and then the scheduling process (§3.3.2).

3.3.1 Model Grouping

Based on O4 and O5, multiplexing a C-heavy model with an M-heavy model maximizes Ruti. Such multiplexing makes the Cuti and Muti of a GPU comparable. If a GPU's C-space is much lower than its M-space or vice versa, it may not be able to host an additional model. Based on this, we follow one principle during multiplexing models. That is, the sum of the Creqs of the models is nearly equal to the sum of the Mreqs of the models in the GPU (i.e., $\sum_i Creq_i \approx \sum_i Mreq_i$). Based on this, USHER groups the models such that $\sum_i Creq_i \approx \sum_i Mreq_i$ for the models in each group.

Before conducting the grouping, USHER first finds the Creq and Mreq of each model. USHER calculates the average Rreq across all possible BS and GPU type combinations using GK-Estiamtor (described in §3.2). For a GPU type, USHER stops at the BS for which the Creq or Mreq exceeds the maximum C-space or M-space of the type, respectively.

Next, USHER performs the grouping using a variant of k-means clustering [56]. The algorithm clusters a set of elements into nearly equal-sized groups so that the sum of the distances between elements within each group is minimized, while the sum of the distances between groups is maximized.

At the beginning, each group consists of a single model. USHER calculates the distance between every two models as $D = |\sum_i Creq_i - \sum_i Mreq_i|$. Then, it uses the k-means algorithm to group the models into several groups, where each group consists of two models such that D is minimized, i.e., $\sum_i Creq_i \approx \sum_i Mreq_i$ for each group. Next, considering each group as an element, USHER executes another pass of the algorithm. This process merges two groups created by the previous pass to one and increases the number of models in each group by two times. As a result, if we decide to have at most 2^p (i.e., 4) models in each group, we need to perform p passes of the algorithm. Finally, the models are grouped into several groups and the set of the groups is denoted by $\mathbf{G} = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_n\}$.

3.3.2 Scheduling: Deciding Configuration and Placement

After grouping the models, for every model in each model group $\mathcal{G}_i$, IR-Scheduler decides the schedule. Below, we use

Algorithm 1 Interference-aware and resource utilization-maximizing scheduler for **G**.

```
1: for each $\mathcal{G}_i \in$ G do
2:     Generate all possible configurations={(BS, RD) for each
       model $M \in \mathcal{G}_i$}.
3:     for each configuration do:
4:         cost, total_goodput = PLACEMENT(configuration)
5:     Schedule as per the configuration for which all of the requests are
       completed within their latency SLOs, i.e., total_goodput =
       total_workload and the cost is minimum.
```

the non-fixed cluster as an example to present the method and then extend it for the fixed cluster. Based on O2, workload division (i.e., RD>1) may increase the utilization even when one model replica is enough to complete all of the inference requests within the latency SLO. During the scheduling, first, USHER takes all possible configurations, i.e., {(BS, RD) for each model in the group $\mathcal{G}_i$} (Algorithm 1). For each configuration, it finds the placement to minimize the cost (Algorithm 2). Finally, USHER chooses the configuration and placement that result in the minimum cost.

The placement algorithm tries to assign one group of models $\mathcal{G}_i$ to the same GPU set to maximize Cuti and Muti. Therefore, to assign the models in $\mathcal{G}_i$, the algorithm prioritizes the GPUs that are already assigned with the models in $\mathcal{G}_i$, and initializes a new GPU only when no used GPU can host a model. To avoid resource fragmentation and increase Ruti, it prioritizes the models $\mathcal{G}_i$ that have high Creq and Mreq and aims to place it to a GPU that leaves the lowest C-space and M-space after hosting it. Additionally, based on O5, USHER takes placement decisions alternatively between C-heavy and M-heavy models. The scheduling algorithm is shown in Algorithm 1. The algorithm finds the best placement decision for each possible configuration by calling the PLACEMENT() algorithm (Lines 1-4). The set of possible values of BS of a model is: {4, 8, 16, 32, 64, 128}, and the set of possible values of RD of a model is: $\{m \cdot c_M^l\}$, $m = 1,2,...,6$, where c_M^l is the minimum possible value of RD of model M to satisfy its SLO. It is calculated as the minimum number of GPUs of the highest GPU type required to complete all of the model M's requests within the SLO considering the highest possible BS in each GPU. The highest possible BS is taken as the minimum value between 128 and the BS beyond which its Mreq exceeds the memory capacity ($\max M$) of the highest capacity GPU type. This way, as the maximum possible value of RD of a model can be as much as 6 times the minimum possible value, the scheduling algorithm conducts workload division based on O2.

Algorithm 2 Placement algorithm for model group $\mathcal{G}_i$.

```
1: procedure PLACEMENT (configuration)
2:     $\mathcal{G}_{iGPU} \leftarrow$ GPU group for $\mathcal{G}_i$, initially empty
3:     for each $M \in \mathcal{G}_i$ do
4:         Calculate its $Creq$ and $Mreq$ in each type of GPU
5:         if $Creq > \max C$ or $Mreq > \max M$ (highest-capacity GPU) then
6:             return Infeasible_configuration
7:     Group the models into C-heavy and M-heavy models
8:     Sort two groups in descending order of $Creq + Mreq$:
       $\{M_1, M_2, \ldots, M_n\}$ and $\{M'_1, M'_2, \ldots, M'_m\}$
9:     final_model_list $\leftarrow \{(M_1, M'_1), (M_2, M'_2), \ldots, (M_n, M'_m)\}$
10:    for each $M \in$ final_model_list do
11:        MODEL_REPLICA_PLACEMENT_WITHIN_$\mathcal{G}_{iGPU}$()
12:        MODEL_REPLICA_PLACEMENT_OUTSIDE_$\mathcal{G}_{iGPU}$()
13:        for each model replica of $M$ do
14:            NEW_LOWEST_COST_GPU_INITIALIZATION()
15:            Assign the new GPU to $\mathcal{G}_{iGPU}$.
16:    for each $M \in \mathcal{G}_i$ do
17:        goodput$_M$ = min (achieved_goodput$_M$, workload$_M$)
18:    total_goodput = $\sum_M$goodput$_M$
19:    return additional costs for initializing new GPUs and
       total_goodput for the taken placement decision.
```

The placement algorithm is shown in Algorithm 2. USHER first calculates the Creq and Mreq for each model in each GPU type according to the configuration given as input using GK-Estimator (Lines 3-4). Then it further groups the models in the model group $\mathcal{G}_i$ into C-heavy and M-heavy models (Line 7). A model is C-heavy if its average C-req/M-req ≥ 1.2, and is M-heavy if M-req/C-req ≥ 1.2. Next, USHER sorts each of the two sub-groups in the descending order of the Creq+Mreq (Line 8). After that, USHER pairs up each two models from the two sub-groups to create final_model_list (Line 9). Finally, USHER inserts the models that are neither C-heavy nor M-heavy into the list while maintaining the descending order of Creq+Mreq.

Then, USHER picks up a pair or a model one by one from the list to be assigned to a GPU. Specifically, USHER calls the MODEL_REPLICA_PLACEMENT_WITHIN_$\mathcal{G}_{iGPU}$ function (Line 11). The function places as many model replicas of M as possible to the mode's GPU group (denoted by $\mathcal{G}_{iGPU}$). The GPU group of a model group is defined as the group of GPUs that hosts most of the model replicas of the model group. Basically, when USHER initializes a new GPU for any model replica of model group $\mathcal{G}_i$, the new GPU is added to the GPU group $\mathcal{G}_{iGPU}$. This way, USHER tries to place the model replicas in the same model group to the GPUs of the same GPU group. If multiple GPUs are available for a model or a pair, USHER chooses the one that leaves the lowest C-space+M-space after hosting it to avoid resource fragmentation.

After that, USHER calls MODEL_REPLICA_PLACEMENT_OUTSIDE_G_{iGPU} that places as many remaining model replicas of M as possible to the GPUs of the other GPU groups (Line 12). Finally, for each model replica that is not placed to any GPU yet, USHER calls NEW_LOWEST_COST_GPU_INITIALIZATION that initializes a new GPU of the GPU type that can host the model or pair with the minimum cost and assigns the GPU to G_{iGPU} (Lines 13-15). At last, the placement algorithm returns to the scheduling algorithm the additional costs for initializing the new GPUs and the total goodput across all the models (Line 19). As [14], goodput of a model is taken as the ratio between the batch size and the expected time (including in-queue wait time) to complete a batch. The execution time of a batch is calculated using the Time-Regressor in §3.2.

For the fixed cluster setup, the modifications are as follows. First, the maximum value of model replication degree is capped by the total number of GPUs in the cluster. Second, in Algorithm 1, USHER schedules as per the configuration for which the `total_goodput` is maximum.

3.4 Operator Graph Merging to Minimize Cache Interference

To minimize interference in GPU cache, among the models assigned to one GPU, USHER merges as many operator graphs of the models as possible into a single graph. To perform maximal operator graph merging, USHER first groups the models to sub-groups based on their architectural similarity (i.e., the structure and the constituting operators) (§3.4.1). Then, for each sub-group, motivated by O6, USHER decides which operators across multiple graphs to merge based on their weight similarity (§3.4.2). Finally, during the merging process, USHER extracts away the largest common weight submatrix between the operators that need to be merged and ensures that the matrix multiplications associated with the submatrix for different inputs of different models are processed at the same time, while the submatrix is in the GPU cache (§3.4.3).

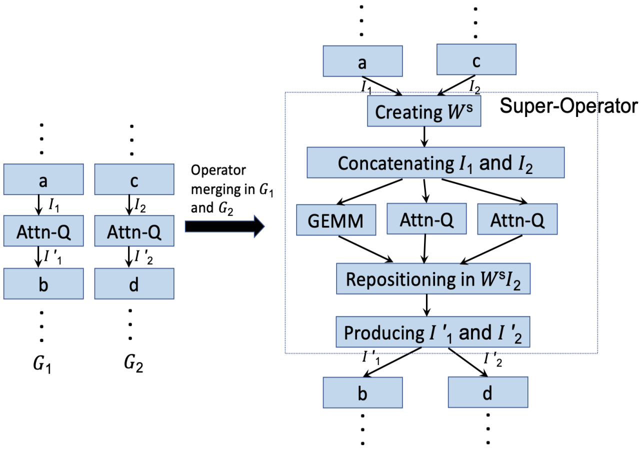
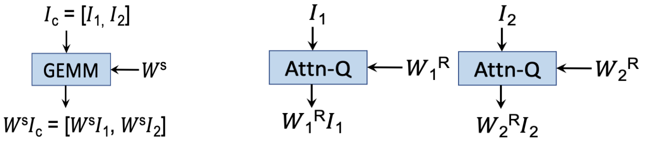


Figure 11: Operator merging in G_1 and G_2 to maximize GPU cache usage. Attn-Q refers to the Attention Query operator.



(a) Executing on W^s at the same time.

(b) Executing the remaining operations using separate operators.

Figure 12: Creating a new operator GEMM.

3.4.1 *Grouping Architectural-Similar Operator Graphs*

The models with architectural-similar operator graphs are more likely to have high similarity in their weights [54]. Based on this, USHER uses DBSCAN algorithm [57] to group the models assigned to a GPU based on architectural similarity. Given a set of elements, DBSCAN algorithm can cluster the elements with very little distance (i.e., 10^{-7}) between themselves within the same group, without requiring any predetermined number of groups or number of elements within a group. Inside DBSCAN algorithm, USHER uses graph edit distance [58] to calculate the distance to measure the architectural similarity between two operator graphs. Basically, the edit distance algorithm finds how many addition/deletion/replacement of operators need to be performed to make the two operator graphs identical. A shorter distance means higher architectural similarity.

3.4.2 *Graph Matching in an Architectural-Similar Group*

Among the models in an architectural-similar group. USHER first randomly takes two models. Then, it generates a bipartite graph $\mathcal{B}$ containing the operators of both models. An edge e exists between two vertices from the two graphs if and only if all of the following conditions are satisfied: (i) same type (i.e., either convolutional operator or attention operator), (ii) same starting time (explained in §3.2), (iii) the weight similarity between the operators (described in §2.4) is no less than ω (e.g., 40%). It is assigned as the edge weight.

After generating $\mathcal{B}$, USHER finds the maximal weighted matching using Hungarian algorithm [59] in $\mathcal{B}$, which chooses a set of independent edges (i.e., that do not share any common vertex) such that the sum of weights is maximized. The two endpoint operators of each of the chosen edges are matched and will be merged (explained in §3.4.2). Then, USHER randomly takes another model from the remaining models and generates a new bipartite graph $\mathcal{B}'$ using $\mathcal{B}$ and the other model by repeating the same procedure. This process repeats until no more models in the group can be merged.

3.4.3 *Operator Merging Process*

As an example of operator merging, we describe the process for two *Query* operators in the attention layers of two Transformer models. It is a matrix multiplication operation: $I'_1 = W_1 I_1$, where I_1 is the input and W_1 is the Query weight matrix. Now, we explain how USHER modifies this operation for operator merging. If I_1 and I_2, as well as W_1 and W_2, do not have the same size, we apply zero padding to make them the same size.

Fig. 11 shows the overall process of merging two Query operators of two graphs. USHER creates a Super-Operator to merge operators, which consists of several individual operators, each of which performs a specific task as described below. First, it extracts out the largest common submatrix between W_1 and W_2, denoted by A_{W_1,W_2} using template matching [60]. It creates a matrix W^s, which contains A_{W_1,W_2} at the same position as W_1 (or W_2), and zeros in other entries. It also creates another matrix W_1^R, which is the same as W_1, except that the entries associated with A_{W_1,W_2} are all zero, and creates matrix W_2^R from W_2 similarly. Second, as illustrated in Fig. 12a, USHER concatenates the inputs I_1 and I_2 into a matrix as

$I_c = [I_1, I_2]$ and creates a new general matrix multiplication (GEMM) operator that performs $W^s I_c = [W^s I_1, W^s I_2]$. This way, the new GEMM operator executes the matrix operations of the original two operators corresponding to A_{W_1,W_2} at the same time, while W^s is still loaded in the GPU cache, thus maximizing its usage.

Also, as illustrated in Fig. 12b, for each of the original operators, USHER creates a Query operator to perform the remaining matrix multiplication operations that are not associated with W^s: $W_1^R I_1$ and $W_2^R I_2$. Third, I_1' can be calculated simply as $I_1' = W^s I_1 + W_1^R I_1$. However, the entries in $W^s I_2$ need to be repositioned. This is because A_{W_1,W_2} is positioned in W^s according to its position in W_1. This repositioned $W^s I_2$ would be generated if A_{W_1,W_2} were positioned in W^s according to its position in W_2. Finally, after the repositioning, I_2' is calculated as $I_2' = W^s I_2 + W_2^R I_2$.

While merging a Super-Operator with another Query operator O_q, USHER merges each Query operator inside the Super-Operator with O_q.

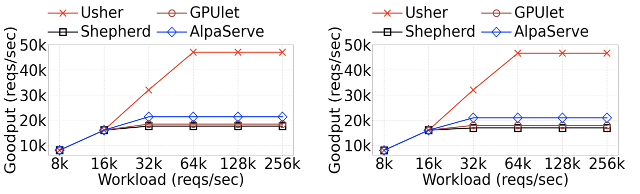
4 Implementation Details

We developed USHER using Python and the implementation is available at [61]. We used Tensorflow for the inference executions of the models, but note that our techniques are general and can be incorporated on other serving platforms. After the XLA operator graph optimization [62] is performed on the operator graph in TensorFlow, we converted it into the framework-independent ONNX format to ensure USHER works for other ML frameworks (e.g., PyTorch, MXNet) as well. We found the GPU kernels called by an ML framework during the execution of an operator by profiling the operator using Nvidia Nsight [52] with the `print-gpu-trace` option turned on. We used `achieved_occupancy` and `dram_utilization` options in Nsight to measure the Creq and Mreq of a kernel, respectively. In each GPU, we used Nvidia MPS [13] to divide the GPU computation resource among the models based on their requirements. After merging the operator graphs in ONNX format, we applied the TVM optimization [63] from Nvidia TensorRT [64] to further optimize the merged graph and executed the merged graph as a single `CUDA Context` in MPS, which was allocated the sum of `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`s assigned to the models whose operator graphs were merged.

5 Performance Evaluation

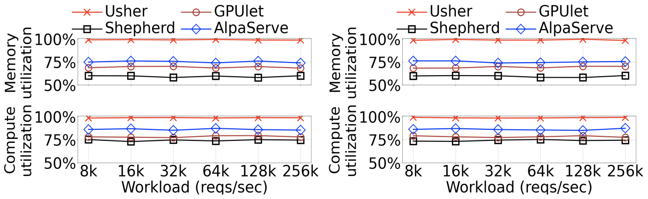
5.1 Experimental Setup

Unless otherwise specified, the experiment settings are the same as those in §2. In addition to the models described in Table 1, we also used two multi-model applications that include multiple DL models [2, 50]: video surveillance (SLO: 500ms) [2] and social media (SLO: 750ms) [65]. In addition to the Microsoft Azure Function trace 2019 (MAF1) with steady and dense request arrival rates, we also experimented with MAF trace 2021 (MAF2) with bursty arrival rates [66].



(a) MAF1 (b) MAF2

Figure 13: Goodput comparison of different methods in real testbed for a fixed cluster.



(a) MAF1 (b) MAF2

Figure 14: GPU computation and memory utilization comparison of different methods in real testbed for a fixed cluster.

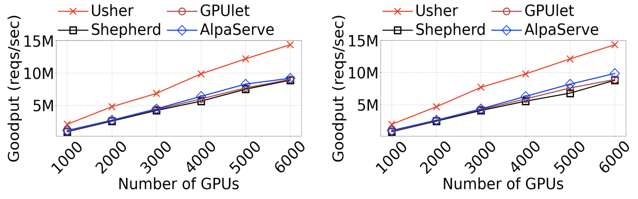
Testbed. We conducted both real testbed and simulation experiments. The real testbed is a cluster with 6 AWS EC2 p3.8xlarge servers, each of which consists of 4 V100 Nvidia GPUs and the GPUs are inter-connected via NVLinks. In the simulation, we increased the number of GPUs up to 6000 to simulate a large enterprise-grade GPU cluster [67]. As it is prohibitively expensive to actually execute the models using these many GPUs, we report the result directly from the scheduler decision, without actually running the models. In simulation, we experimented with tremendously large workloads reaching up to 15M requests/second to simulate the large workloads in enterprise-grade clusters. We tested for both fixed and non-fixed cluster setups. We compared USHER with Shepherd [3], GPUlet [14], and AlpaServe [4].

5.2 Comparison Results

Our key results include: USHER (i) achieves up to 2.6× higher goodput in a fixed cluster and (ii) requires up to 3.5× lower cost in a non-fixed cluster.

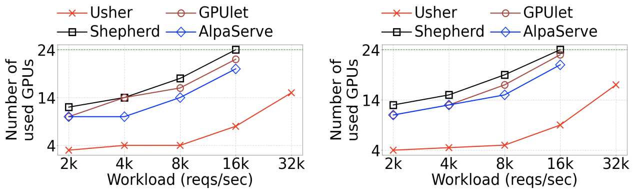
5.2.1 Fixed Cluster

Fig. 13 shows the average goodput in the real testbed with varying average workloads (averaged across the total duration of 2 weeks for a trace). Fig. 14 shows the average Cuti and Muti per GPU. USHER achieves 1×-2.6× higher goodput, 22%-24.2% higher Cuti and 38.9%-40.1% higher Muti compared to Shepherd. This is because USHER performs



(a) MAF1 (b) MAF2

Figure 15: Goodput comparison of different methods in simulation for a fixed cluster.



(a) MAF1 **(b) MAF2**

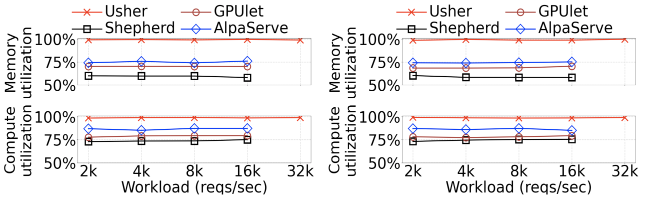
Figure 16: Number of used GPUs in real testbed for a homogeneous non-fixed cluster. The maximum number of GPUs was 24.

model multiplexing in each GPU in an interference-aware manner, whereas Shepherd allows only one model in each GPU. We observed that the models in each of the following sets were multiplexed in USHER for at least half the total duration of the experiment: {YOLO-v3, ResNeXt-101, Inception-v4, GNMT, BERT}, {ResNet-101, EfficientNet-B7, BERT, GPT-2}, {SqueezeNet, Llama-2}, and {R-CNN, ShuffleNet-v2, GNMT, GPT-2}.

USHER achieves 1×-2.2× higher goodput, 19%-23% higher Cuti and 25.1%-32.2% higher Muti compared to GPUlet and AlpaServe. GPUlet and AlpaServe only try to optimize the GPU computation use, while USHER addresses the interference between models not only in the computation space but also in the memory and cache spaces. Also, USHER has several strategies in its IR-scheduler such as multiplexing C-heavy models with M-heavy models and holistic workload division to maximize the utilizations of both computation and memory spaces. USHER performs consistently for both traces, indicating its resilience to different request arrival patterns. The minimal rescheduling overhead of USHER (i.e., 0.51s from Table 2) enables it to quickly adapt to the bursty workload of MAF2. Nonetheless, if the workload exhibits very frequent burstiness, then the benefits of USHER may be reduced. However, such extreme workloads are uncommon in real-world settings [3, 4] (e.g., the production workload illustrated in §5.4, where burstiness occurs after every 45s-300s) and thus the rescheduling overhead of USHER is reasonable enough to not have any adverse impact on its performance.

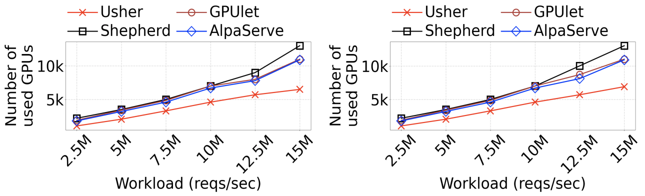
Goodput of USHER becomes stable at around 47k reqs/s, whereas Shepherd, GPUlet, AlpaServe become stable at much less goodput values. At this point, the method has used up all the GPU resources in the fixed cluster. Fig. 14 establishes that merely increasing the workload cannot saturate the GPUs. This is because it is the batch size that mainly determines the resource consumption of a model and a system cannot choose a larger batch size that leads to a latency exceeding the SLO. To cope up with the increased workload, a system scales out, i.e., increases the number of used GPUs.

Fig. 15 shows the goodput in simulation with varying number of V100 GPUs. USHER achieves 1.5×-1.9× higher goodput compared to the comparison methods due to the same reasons described above. With the increase in the number of GPUs, the goodput of USHER increases 1.6×-2.1× faster than other methods, indicating the higher scalability of USHER.



(a) MAF1 **(b) MAF2**

Figure 17: GPU computation and memory utilization of different methods in real testbed for a homogeneous non-fixed cluster.



(a) MAF1 **(b) MAF2**

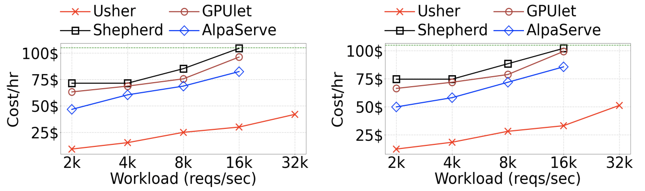
Figure 18: Number of used GPUs of different methods in simulation for a homogeneous non-fixed cluster.

5.2.2 *Non-fixed Cluster*

For the non-fixed cluster, we first considered homogeneous V100 GPUs. Then, the cost is proportional to the number of used GPUs. We report the average number of GPUs required by a method at a second. Next, we considered heterogeneous GPUs: K80, V100, A100, and H100. The GPU types have varying costs following AWS on-demand pricing [68].

Homogeneous GPUs. Fig. 16 shows the number of used GPUs in a homogeneous real testbed to complete all inference requests within their SLOs. Fig. 17 shows the average Cuti and Muti per GPU. USHER requires 2.5×-3× fewer GPUs, and achieves 19.3%-24.4% and 24.9%-40% higher Cuti and Muti, respectively, compared to Shepherd, GPUlet, and AlpaServe. As the comparison methods either do not employ multiplexing or suffer from increased latency due to inter-model interference, they require more GPUs.

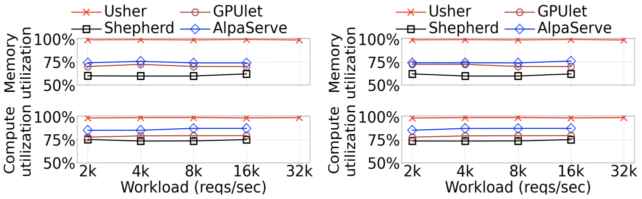
Fig. 18 shows the number of used GPUs in simulation. USHER uses 1.7×-2.1× fewer GPUs due to the same reasons explained above. In both real testbed and simulation, with the increase of the workloads, the increase rate in the number of GPUs of USHER is slower than other methods especially when the workload is very high. These results show the cost-effectiveness of USHER even for large workloads.



(a) MAF1 **(b) MAF2**

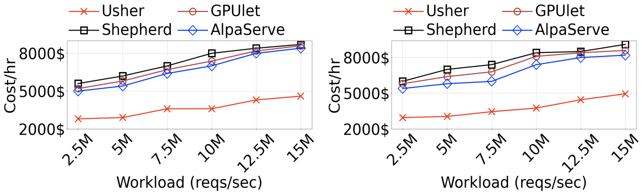
Figure 19: Cost comparison in real testbed for a heterogeneous non-fixed cluster. The maximum cost/hr was 105$.

Heterogeneous GPUs. Fig. 19 shows the cost per hour with varying workloads in a heterogeneous real testbed. Fig. 20



(a) MAF1 **(b) MAF2**

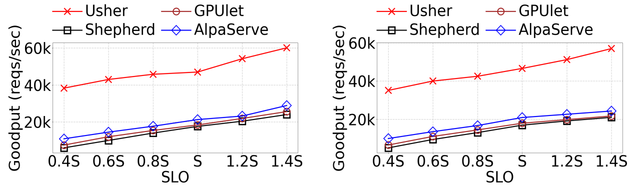
Figure 20: GPU computation and memory utilization comparison in real testbed for a heterogeneous non-fixed cluster.



(a) MAF1 **(b) MAF2**

Figure 21: Cost comparison of different methods in simulation for a heterogeneous non-fixed cluster.

shows the average Cuti and Muti per GPU. USHER requires 2.8×-3.5× lower cost, and achieves 19%-24.1% and 25.2%-40.3% higher Cuti and Muti, respectively, due to the same reasons explained above. USHER performs slightly better for heterogeneous GPUs compared to homogeneous GPUs as USHER can choose the optimal GPUs from varying GPU types depending on their cost-performance trade-offs.

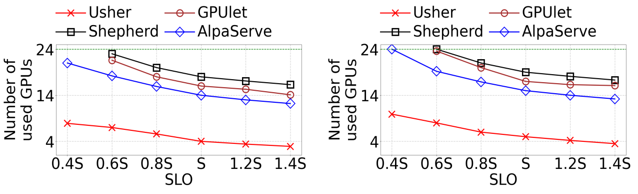


(a) MAF1 **(b) MAF2**

Figure 22: Goodput comparison with varying SLO values in real testbed for a fixed cluster. S denotes the default SLO (Table 1).

Fig. 21 shows the cost per hour in simulation. USHER incurs 1.9×-2.2× lower cost for the same reasons described above. With the increase in workload, its cost increases slower than other methods, especially when the workload is very high.

Overheads. Table 2 presents the average time overhead and average impact on accuracy of different methods. USHER's grouping of models takes only 0.1s and it needs to be updated only when there is addition or deletion of models in the system. The decision-making times of the scheduling in different methods are comparable. This is because all the methods employ time-efficient heuristics to accelerate the scheduling process in order to excel in autoscalability when workload changes (§5.4). USHER's operator graph merging takes slightly more time than its scheduling. Hence, after the scheduling, the requests are executed using individual model graphs. After the graphs are merged, the requests are then executed on the merged graph. For the models whose GPU placement has changed during scheduling, each method takes 0.69s-0.72s to load the models from host memory to GPU memory. Table 3 presents the average model loading time of USHER for each model shown in Table 1. Rescheduling does not happen so frequently [3,4], e.g., after every 45s-300s from Fig. 24. As a result, considering the scheduling time required in USHER, there is enough time left before another scheduling occurs to realize the benefits of USHER. The accuracy loss is only 0.0003% per request batch of a model due to the operator graph merging. This is because, while finding the longest common submatrix in the merging process, USHER takes two weight values as the same only when their absolute difference is very low (i.e., $\leq 10^{-7}$) (§2.4). Table 3 shows the average accuracy loss per request batch for each model in USHER.



(a) MAF1 **(b) MAF2**

Figure 23: Number of used GPUs with varying SLO values in real testbed for a homogeneous non-fixed cluster. The maximum number of GPUs was 24. S denotes the default SLO (Table 1).

Table 2: Overheads of the methods.

Methods	Grouping of models	Scheduling algorithm		Operator graph merging based on weight similarity	Impact on model accuracy due to operator graph merging
		Decision making	Model loading		
USHER	0.1s	0.51s	0.63s	2.3s	-0.0003%
Shepherd	0	0.5s	0.62s	0	0
GPUlet	0	0.46s	0.64s	0	0
AlpaServe	0	0.63s	0.65s	0	0

5.3 Performance on Varying SLOs

5.3.1 Fixed Cluster

Fig. 22 shows the goodput with varying SLO values in the same real testbed setup of Fig. 13 with workload=256k reqs/sec. In the figure, S denotes the default SLO of each model as defined in §2, and $m_f S$ denotes that the SLO is multiplied by $m_f \in \{0.4, 0.6, 0.8, 1, 1.2, 1.4\}$. USHER achieves 2.2×-2.7× higher goodput than the comparison methods for the same reason described in §5.2.1. As SLO decreases, goodput also decreases for each method. This is because, in the fixed GPU resources of the cluster, more requests miss their SLO deadlines as the SLO becomes stricter. The result shows that USHER retains its higher goodput compared to the existing methods even when the SLO is ultra-low.

5.3.2 Non-fixed Cluster

Fig. 23 shows the number of used GPUs with varying SLO values in the same real testbed setup of Fig. 16 to complete all inference requests within their SLOs with workload=8k reqs/sec. The result shows that USHER achieves 3.2×-4.3× fewer GPUs than the comparison methods for the same reason

Table 3: Model loading and accuracy overheads in USHER.

Model name	Model loading	Impact on model accuracy due to operator graph merging
YOLO-v3	0.44s	-0.0002%
R-CNN	0.57s	-0.0003%
MobileNetSSD-v2	0.495s	-0.0003%
ResNet-50	0.51s	-0.0002%
ResNet-101	0.586s	-0.0004%
ResNeXt-50	0.531s	-0.0004%
ResNeXt-101	0.7s	-0.0003%
SqueezeNet	0.31s	-0.0001%
ShuffleNet-v2	0.39s	-0.0002%
MobileNet-v2	0.395s	-0.0002%
DenseNet-121	0.4s	-0.0002%
DenseNet-201	0.49s	-0.0002%
Inception-ResNet-v2	0.6s	-0.0002%
Inception-v3	0.53s	-0.0003%
Inception-v4	0.58s	-0.0003%
EfficientNet-B7	0.64s	-0.0004%
GNMT	0.89s	-0.0002%
BERT	0.78s	-0.0004%
GPT-2	1.18s	-0.0004%
Llama-2	1.59s	-0.0003%

described in §5.2.2. As SLO decreases, the number of used GPUs increases for each method. This is because each method needs to create more replicas of a model to execute more requests in parallel as the SLO becomes stricter. Otherwise, the requests would have to wait longer for the required GPU resource, leading to SLO violation. The result shows that USHER retains its superior cost-efficiency compared to the existing methods even when the SLO is very strict.

5.4 Microanalysis on Autoscalability

To better evaluate the system autoscalability during rescheduling, we measured the number of used GPUs and the in-queue wait time of a request at each second during a randomly chosen window of 1000 seconds. We used the MAF2 trace in the homogeneous non-fixed cluster real testbed. Fig. 24 shows the results. Each method increases the number of GPUs when the workload surges up and decreases it when the workload surges down. USHER requires 1.6×-3.9× fewer GPUs compared to the comparison methods, due to its interference-minimizing and resource utilization-maximizing design. The in-queue wait time is almost the same for all the methods and also varies by only 0.8ms-1.4ms as the workload surges up or down. This means that all the methods excel in autoscalability, but USHER requires much fewer GPUs.

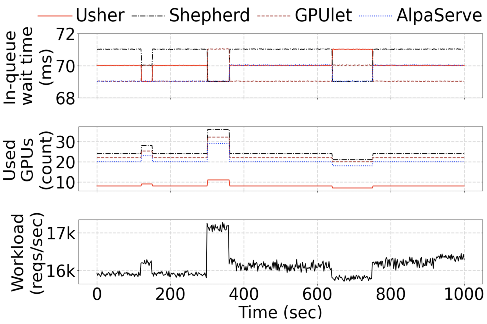
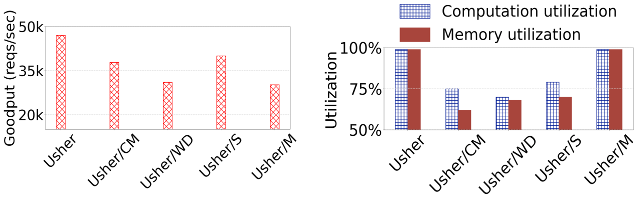


Figure 24: Microanalysis on autoscalability.

Table 4: Performance of USHER's GK-Estimator.

Methods	Accuracy of computation requirement calculation	Accuracy of memory requirement calculation	Cost	Time
USHER's GK-Estimator	99.98%	99.98%	**0**	**31.6ms**
Profiling	**100%**	**100%**	42.7$	4.8hr



(a) Goodput **(b) GPU utilization**

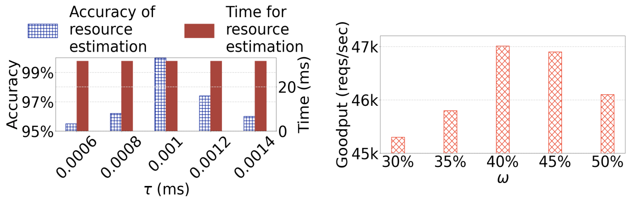
Figure 25: Performance of different variants of USHER.

5.5 Ablation Study

In this section, we evaluate the effectiveness of each proposed strategy of USHER. We first evaluated the cost- and time-efficiency of USHER's GK-Estimator compared to the existing profiling approaches in estimating the resource requirements of a new model. Table 4 shows the results averaged across the models in Table 1. The GK-Estimator takes 100% less cost and time with comparable accuracy. This is because the profiling approach needs to actually execute the model in the GPUs for different BSs and GPU types, whereas USHER's GK-Estimator only needs to analyze the kernel-level computation graph.

To measure the effectiveness of the other methods in USHER, we created several variants of USHER as follows. 1) USHER/CM skips the step to classify the models in a group to C-heavy and M-heavy. 2) USHER/WD does not conduct workload division between multiple GPUs if one GPU can support the workload to satisfy the SLO. 3) USHER/S does not sort the models based on their computation and memory requirements. 4) USHER/M does not have the OG-Merge.

Fig. 25 shows the goodput and GPU utilization performance of USHER and its different variants in the same setup as Fig. 13a with workload=256k reqs/s. The results show that USHER achieves 24.3%-51.6% higher goodput than USHER/CM, USHER/WD, and USHER/S. This is because, by skipping a method in each of these variants, the Cuti and Muti decrease by 24%-41.4% and 42%-59.6%, respectively, resulting in much lower goodput. USHER achieves 55.7% higher goodput than USHER/M because OG-Merge reduces the interference in GPU cache, resulting in lower latency.



(a) Impact of τ **(b) Impact of ω**

Figure 26: Sensitivity of USHER on its parameters.

5.6 Sensitivity on Parameters

We did the following experiments in the fixed cluster real testbed setup using MAF1 trace with 256k reqs/s workload.

Impact of τ (in §3.2). Fig. 26a shows the accuracy and time for resource estimation of a model with varying values of τ. All τ values lead to the same time for resource estimation. However, we chose $\tau = 0.001$ms as it leads to the highest accuracy of resource estimation since it can correctly capture which GPU kernels will be executed concurrently.

Impact of ω (in §3.4.2) Fig. 26b shows the goodput of USHER for different values of ω. We chose $\omega = 40\%$ as it provides the highest goodput. When $\omega > 40\%$, the number of operators that can be merged is reduced, leading to higher cache interference. When $\omega < 40\%$, more operators can be merged. However, the overhead of additional operations in Fig. 11 outweighs the benefit.

6 Limitations and Discussion

Precision Quantization. The current version of USHER implements FP16 quantization of weights. In the future, we will explore the impact of various precision quantizations (e.g., FP8, FP32) on various factors such as accuracy, interference, and resource utilization and extend USHER to adaptively select the most suitable precision quantization for each model based on the above factors.

Model Parallelism. USHER supports model parallelism out of the box for large models. We assume that model parallelism is enabled by the underlying framework (e.g., DeepSpeed decides how to do model parallelism on Llama-2 in our experiments (§2)), and USHER simply uses it. An interesting future work would be to jointly optimize the model parallelism and placement strategies of USHER to further enhance the resource utilization.

7 Related Work

Inference Serving Systems without Spatial Multiplexing. Many of the systems avoid spatial multiplexing of models within a GPU to prevent inter-model interference [3, 7, 11, 65, 69–81]. Shepherd [3] aggregates request streams into similar-sized groups for high computation utilization and schedules placement to maximize goodput within each group. Several methods [7, 65, 71–76] rely on profiling to find the optimal request batch size for each model, aiming for high GPU utilization and goodput. However, these methods suffer from low resource utilization due to lack of spatial multiplexing (§2.1). Additionally, offline profiling to calculate the resource requirements of a model is time-consuming and costly.

Inference Serving Systems with Spatial Multiplexing. A set of systems adopt spatial multiplexing to enhance GPU utilization while maximizing goodput [1, 2, 4, 12, 14, 21, 23, 50, 82–85]. GPUlet [14] proposes a heuristic for placing models in GPUs to maximize computation space utilization. AlpaServe [4] explores the best placement scheduling by leveraging model parallelism. However, due to inter-model interference in spatial multiplexing, these systems may suffer from longer inference latency (§2.1). Additionally, these systems fail to maximize both GPU computation and memory utilizations (§2.1). Orion [86] is a recent work that maximizes resource utilization by spatially multiplexing the best-effort jobs (e.g., training), while avoiding multiplexing the high-priority jobs (e.g., inference) so that they are not impacted by inter-model interference. However, in our scenario where all the jobs are high-priority inference, Orion will fail to maximize utilization due to the lack of multiplexing. A group of methods [21–23] propose merging layers and sharing parameter weights across multiple models to reduce the memory requirement. However, the merging processes cannot solve the interference in GPU cache as they do not maximize the usage of cache contents.

8 Conclusion

Spatial multiplexing has the potential to increase resource utilization of the GPUs to design a cost-efficient inference serving system. However, it requires careful system design to address the challenges of spatial multiplexing, i.e., maximizing the utilizations of both computation and memory spaces, while minimizing inter-model interference. To this end, we propose USHER. USHER has a lightweight interference-aware scheduler that schedules the models to jointly maximize GPU computation and memory utilizations. During the scheduling, USHER uses a novel lightweight GPU kernel-based estimator to compute the resource requirement of each model. Finally, USHER has a novel operator graph merging approach to minimize interference in GPU cache. Experimental results on both real testbed and large-scale simulations show that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency compared to existing systems.

Acknowledgments

We sincerely thank the anonymous reviewers of OSDI and our shepherd for their invaluable feedback. We are grateful to Kevin Skadron for the discussion on GPU architecture. This research was supported in part by U.S. NSF grants NSF-1827674, NSF-2206522, NSF-1822965, FHWA grant 693JJ31950016, Microsoft Research Faculty Fellowship 8300751, and Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation, and workforce development. For more information about CCI, please visit cyberinitiative.org.

References

[1] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *Proc. of NSDI*, 2022.

[2] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proc. of SIGCOMM*, 2023.

[3] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *Proc. of NSDI*, 2023.

[4] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. of OSDI*, 2023.

[5] Amazon AWS. Inference dominates ML infrastructure cost. https://aws.amazon.com/machine-learning/inferentia/, 2023.

[6] Forbes. Generative AI breaks the data center. https://www.forbes.com/sites/tiriasresearch/2023/05/12/generative-ai-breaks-the-data-center-data-center-infrastructure-and-operating-costs-projected-to-increase-to-over-76-billion-by-2028/, 2023.

[7] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *Proc. of NSDI*, 2022.

[8] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. of EuroSys*, 2023.

[9] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proc. of SC*, 2021.

[10] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *Proc. of OSDI*, 2020.

[11] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *Proc. of OSDI*, 2020.

[12] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *Proc. of ATC*, 2021.

[13] Nvidia. Nvidia Multi Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html, 2021.

[14] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *Proc. of ATC*, 2022.

[15] Nvidia. Nvidia MIG. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2023.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*, 2014.

[17] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, pages 1–14, 2011.

[18] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[19] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[20] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proc. of SIGCOMM*, 2022.

[21] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *Proc. of NSDI*, 2023.

[22] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning {Exploratory-Training} framework. In *Proc. of OSDI*, 2020.

[23] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic {Stem-Sharing} for {Multi-Tenant} video processing. In *Proc. of ATC*, 2018.

[24] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199, 2022.

[25] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proc. of ECCV*, 2014.

[26] Ross Girshick. Fast r-cnn. In *Proc. of ICCV*, 2015.

[27] Yu-Chen Chiu, Chi-Yi Tsai, Mind-Da Ruan, Guan-Yu Shen, and Tsu-Tian Lee. Mobilenet-ssdv2: An improved object detection model for embedded systems. In *Proc. of ICSSE*, 2020.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of CVPR*, 2016.

[29] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. of CVPR*, 2009.

[30] Saifuddin Hitawala. Evaluating resnext model architecture for image classification. *arXiv preprint arXiv:1805.08700*, 2018.

[31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[32] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proc. of CVPR*, 2018.

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. of CVPR*, 2018.

[34] Ke Zhang, Yurong Guo, Xinsheng Wang, Jinsha Yuan, and Qiaolin Ding. Multiple feature reweight densenet for image classification. *IEEE Access*, 7, 2019.

[35] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proc. of AAAI*, 2017.

[36] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. of CVPR*, 2016.

[37] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proc. of ICML*, 2019.

[38] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[39] WMT. WMT dataset. https://huggingface.co/datasets/wmt19, 2019.

[40] Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *arXiv preprint arXiv:2302.09419*, 2023.

[41] IMDB. IMDB dataset. https://huggingface.co/datasets/imdb, 2011.

[42] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[43] WikiPedia. WikiText dataset. https://huggingface.co/datasets/wikitext/blob/main/README.md, 2016.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[45] HuggingFace. HuggingFace Models. https://huggingface.co/models, 2023.

[46] DeepSpeed. DeepSpeed Library. https://github.com/microsoft/DeepSpeed, 2023.

[47] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of ATC*, 2020.

[48] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proc. of ECCV*, 2014.

[49] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proc. of NeurIPS*, 2014.

[50] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proc. of SoCC*, 2021.

[51] ONNX. ONNX Operators. https://github.com/onnx/onnx/blob/main/docs/Operators.md, 2023.

[52] Nvidia. Nvidia Nsight. https://developer.nvidia.com/nsight-systems, 2023.

[53] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proc. of PLDI*, 2021.

[54] Fan Lai, Yinwei Dai, Harsha V Madhyastha, and Mosharaf Chowdhury. {ModelKeeper}: Accelerating {DNN} training via automated training warmup. In *Proc. of NSDI*, 2023.

[55] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54, 2004.

[56] Abiodun M Ikotun, Absalom E Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Heming. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622, 2023.

[57] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3), 2017.

[58] David B Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. Comparing heuristics for graph edit distance computation. *The VLDB journal*, 29(1), 2020.

[59] MB Wright. Speeding up the hungarian algorithm. *Computers & Operations Research*, 17(1), 1990.

[60] Simon Korman, Daniel Reichman, Gilad Tsur, and Shai Avidan. Fast-match: Fast affine template matching. In *Proc. of CVPR*, 2013.

[61] USHER. Author code. https://github.com/ss7krd/Usher, 2023.

[62] Tensorflow. Tensorflow XLA Optimization. https://www.tensorflow.org/xla, 2023.

[63] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *Proc. of OSDI*, 2018.

[64] Nvidia. Nvidia TensorRT. https://docs.nvidia.com/tensorrt/index.html, 2023.

[65] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proc. of SoCC*, 2020.

[66] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proc. of SOSP*, 2021.

[67] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *Proc. of ATC*, 2023.

[68] Amazon AWS. AWS on demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2023.

[69] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *Proc. of NSDI*, 2022.

[70] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proc. of SOSP*, 2019.

[71] Zhou Fang, Dezhi Hong, and Rajesh K Gupta. Serving deep neural networks at the cloud edge for vision applications on mobile platforms. In *Proc. of MMSys*, 2019.

[72] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proc. of EuroSys*, 2019.

[73] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *Proc. of NSDI*, 2017.

[74] Vinod Nigade, Ramon Winder, Henri Bal, and Lin Wang. Better never than late: Timely edge video analytics over the air. In *Proc. of SenSys*, 2021.

[75] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proc. of SIGCOMM*, 2018.

[76] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proc. of DAC*, 2022.

[77] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *Proc. of OSDI*, 2022.

[78] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. of SOSP*, 2023.

[79] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[80] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[81] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[82] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. {RECL}: Responsive {Resource-Efficient} continuous learning for video analytics. In *Proc. of NSDI*, 2023.

[83] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proc. of SOSP*, 2023.

[84] Yoonsung Kim, Changhun Oh, Jinwoo Hwang, Wonung Kim, Seongryong Oh, Yubin Lee, Hardik Sharma, Amir Yazdanbakhsh, and Jongse Park. Dacapo: Accelerating continuous learning in autonomous systems for video analytics. *arXiv preprint arXiv:2403.14353*, 2024.

[85] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, Vijaykrishnan Narayanan, and Chita R Das. Usas: A sustainable continuous-learning´ framework for edge servers. In *Proc. of HPCA*, 2024.

[86] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proc. of EuroSys*, 2024.

Fairness in Serving Large Language Models

Ying Sheng[1,2] Shiyi Cao[1] Dacheng Li[1] Banghua Zhu[1] Zhuohan Li[1] Danyang Zhuo[3]
Joseph E. Gonzalez[1] Ion Stoica[1]

[1]UC Berkeley [2]Stanford University [3]Duke University

Abstract

High-demand LLM inference services (e.g., ChatGPT and BARD) support a wide range of requests from short chat conversations to long document reading. To ensure that all client requests are processed fairly, most major LLM inference services have request rate limits, to ensure that no client can dominate the request queue. However, this rudimentary notion of fairness also results in under-utilization of the resources and poor client experience when there is spare capacity. While there is a rich literature on fair scheduling, serving LLMs presents new challenges due to their unpredictable request lengths and their unique batching characteristics on parallel accelerators. This paper introduces the definition of LLM serving fairness based on a cost function that accounts for the number of input and output tokens processed. To achieve fairness in serving, we propose a novel scheduling algorithm, the Virtual Token Counter (VTC), a fair scheduler based on the continuous batching mechanism. We prove a 2× tight upper bound on the service difference between two backlogged clients, adhering to the requirement of work-conserving. Through extensive experiments, we demonstrate the superior performance of VTC in ensuring fairness, especially in contrast to other baseline methods, which exhibit shortcomings under various conditions. The reproducible code is available at `https://github.com/Ying1123/VTC-artifact`.

1 Introduction

In a very short time, Large Language Models (LLMs), such as ChatGPT-4 Turbo [36], have been integrated into various application domains, e.g., programming assistants, customer support, document search, and chatbots. The core functionality rendered by LLM providers to these applications is serving their requests. In addition to the response accuracy, the request response time is a key metric that determines the quality of service being provided. Furthermore, LLM providers seek to utilize their resources efficiently so they can reduce costs and increase their competitiveness in the market.

Today's LLM serving systems [20, 24] typically use First-Come-First-Serve (FCFS) to schedule incoming requests. While simple, this scheduling discipline has several drawbacks. One such drawback is the lack of *isolation*: a client sending a disproportionate number of requests can negatively impact the service of all the other clients sharing the same server (i.e., slow down their requests or even cause timeouts) even when they send very little traffic. In multi-tenant personalized serving (S-LoRA [43], Punica [8]) that uses a dedicated adapter for each user, it is important to ensure fairness among the adapters as well. One solution to address this problem is to limit the incoming load of each client. Many of the existing LLM services do this today by imposing a request-per-minute (RPM) limit [37] for each client.

Unfortunately, RPM can lead to low resource utilization. A client sending requests at a high rate will be restricted even if the system is underutilized. This leads to wasted resources, an undesirable situation given the cost and the scarcity of GPUs. Thus, we want a solution that provides not only isolation (like RPM limit) but also high resource utilization.

This is a common problem in many other domains like networking and operating systems. The solution of choice to achieve both isolation and high resource utilization in those domains has been *fair queueing* [30]. Fair queueing ensures that each client will get their "fair share". In the simplest case, if there are n clients sharing the same resource, the fair share is at least $1/n$ of the resource, which means that each client gets at least $1/n$ of the resource. Furthermore, if some clients do not use their share, other clients with more demands can use it, hence leading to higher resource utilization.

In this paper, we apply fair sharing to the domain of LLM serving at the token granularity. We do it at the token rather than request granularity to avoid unfairness due to request heterogeneity. Consider two clients, client A sends requests of 2K tokens each (both input and output), and client B sends requests of 200 tokens each. Serving an equal number of

[*] Part of the work was done when Ying was visiting UC Berkeley.

requests for each client would be unfair to client B as her requests consume much fewer resources than client A's requests. This is similar to networking where fair queuing is typically applied to the bit granularity, rather than packet granularity.

Despite these similarities, we cannot directly use the algorithms developed for networking and operating systems, as LLM serving has several unique characteristics. First, the request output lengths are unknown in advance. In contrast, in networking, the packet lengths are known before the packet is scheduled. Second, the cost of each token can vary. For instance, the cost of processing an input (prompt) token is typically lower than that of an output token, because input token processing is parallelizable. In contrast, the cost of sending a bit or the cost of a CPU time slice are the same irrespective of the workload. Third, the *effective* capacity of an LLM server (i.e., processing rate expressed in token/sec when the request queue is non-empty) can vary over time. For example, longer input sequences take more memory. This limits the number of batched parallel requests during generation, leading to GPU under-utilization and a lower processing rate. In contrast, the network or CPU capacity is assumed to be fixed.

In this paper, we discuss the factors that need to be considered when defining fairness in the context of LLM serving. We show how different definitions can be incorporated into a configurable service cost function in Section 3. While the cost function can be customized, a simple metric of counting input and output tokens at different prices is extensively used in analysis for the sake of simplicity. We then present a fair scheduling algorithm called *Virtual Token Counter (VTC)* that can be easily adapted for different service cost functions. At a high level, VTC tracks the services received for each client and will prioritize the ones with the least services received, with a counter lift each time a client is new to the queue. It updates the counters at a token-level granularity on the fly, which addresses the unknown length issue. VTC integrates seamlessly with current LLM serving batching techniques (Section 2.1), and its scheduling mechanism does not depend on the server's capacity, overcoming the problem of the dynamically fluctuating server capacity. We also provide theoretical bounds of fairness for VTC in Section 4.1. The serving architecture of VTC is illustrated in Figure 1.

In summary, this paper makes the following contributions:

- This is the first work to discuss the fair serving of Large Language Models to the best of our knowledge. We identify its unique challenges and give the definition of LLM serving fairness (Section 3).
- We propose a simple yet effective fair-serving algorithm called VTC. We provide rigorous proofs for VTC on fairness guarantee, which gives fairness bound within 2× of the optimal bound (Section 4).
- We conduct in-depth evaluations on our proposed algorithm VTC. Results confirm that our proposed algorithms are fair and work-conserving (Section 5).

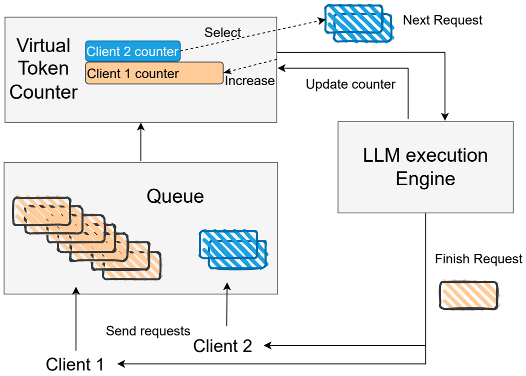


Figure 1: Serving architecture with Virtual Token Counter (VTC), illustrated with two clients. VTC maintains a queue of requests and keeps track of tokens served for each client. In each iteration of the LLM execution engine, some tokens from some clients are generated. The counters of these clients are correspondingly updated. When the condition of adding new requests is satisfied (e.g. memory is released when some other requests finish), VTC will be invoked to choose the requests to be added. VTC achieves fairness by prioritizing clients with the lowest counter and carefully handling clients' leave and rejoin (Section 4.1).

2 Background

In this section, we first introduce how an LLM serving system operates. Then we describe existing methods for ensuring fairness in LLM serving.

2.1 Large Language Models Serving

LLM serving with a single request First, a request contains information about its arrival time (a), input tokens (x), and its associated client (u). Formally, we represent a request using a three-tuple (a, x, u). The system generates output tokens based on the input tokens. For instance, the input tokens can be an incomplete sentence, and the system generates the rest of the sentence [35].

The generation procedure consists of two stages: the initial **prefilling** stage, and the **decoding** stage [39]. Mathematically, x is a sequence of tokens $(x_1, x_2, ..., x_n)$. In the prefilling stage, the LLM computes the probability of the first new tokens: $P(x_{n+1}|x_1, ..., x_n)$. In the decoding stage, the system *autoregressively* generates a new token. At time t ($t \geq 1$), the process is written as: $P(x_{n+t+1}|x_1, ..., x_{n+t})$.

The decoding stage ends when the LLM generates a special end-of-sentence (EOS) token or the number of generated tokens reaches a pre-defined maximal length.

LLM serving with multiple requests In the online serving scenarios, multiple clients submit requests to the serving system. To process these requests, the system maintains two concurrent streams: A monitoring stream adds requests to a waiting queue; an execution stream selects and executes request(s) from the waiting queue.

Naively, the execution stream can choose to execute requests one by one. However, this is highly GPU inefficient due to various natures of the LLM generation procedure. For instance, the decoding steps must be carried out sequentially where the arithmetic intensity is relatively low in a single step. Contemporary serving systems usually perform batching that executes multiple requests concurrently to maximize the system throughput. The most widely used approach in LLM serving is continuous batching [50]. Algorithm 1 shows the pseudocode for continuous batching.[2] The monitoring stream enqueues requests to a waiting queue. The execution stream performs a check on whether there are finished requests at the end of each decoding step. If there are, the system removes these requests and adds new requests from the queue.

Fairness with continuous batching We can naturally integrate fairness policies into the continuous batching algorithm, by designing a fair `select_new_requests()` function in Algorithm 1. Intuitively, the execution stream should keep track of how much service a particular client has received, and prioritize clients that haven't received much service in the next selection. We formally define fairness in the LLM serving context in Section 3 and design a method with theoretical guarantee in Section 4.

We adopt a continuous batching scheme in which a request only leaves the batch when it generates an EOS token or reaches the pre-defined maximum number of generated tokens (i.e., no preemption). This paper focuses on integrating fair scheduling with continuous batching, and we leave an investigation on preemption as an orthogonal future work (discussed in Appendix C.3).

Algorithm 1 LLM serving with Continuous batching

```
Initialize current batch $B \leftarrow \emptyset$, waiting queue $Q \leftarrow \emptyset$
▷ with monitoring stream:
while True do
    if new request $r$ arrived then
        $Q \leftarrow Q + r$
▷ with execution stream:
while True do
    if can_add_new_request() then
        $B_{new} \leftarrow$ select_new_requests(Q)
        prefill($B_{new}$)
        $B \leftarrow B + B_{new}$
    decode($B$)
    $B \leftarrow$ filter_finished_requests($B$)
```

2.2 Existing Fairness Approaches

Fairness is a key metric of interest in computer systems that provide service to multiple concurrent clients [5]. A *fair* LLM serving system should protect clients from a misbehaving client who may try to overload the serving system by submitting too many requests.

RPM Limit Per Client As a common practice of API management (e.x. [37]), specific rate limits are established for each client's API usage to prevent potential abuse or misuse of the API and ensure equitable access for all clients. This limitation is on the metric request-per-minute (RPM). Once a client reaches the RPM limit, the client is only allowed to submit more requests in the next time window. However, it's important to note that while these limits are effective in managing resource allocation during periods of high demand, they may not be *work-conserving* when the number of active clients is low. In such scenarios, the system's capacity might be underutilized, as the imposed limits prevent the full exploitation of available resources.

Fair Queueing [30] The fairness problem has been extensively studied in the past for traditional compute resources, such as CPU cycles and network bandwidth. Fair queuing and its variants (e.g., Weighted Fair Queuing (WFQ) [11], Self-clocked Fair Queueing [15], and Start-time Fair Queueing (SFQ) [17]) have been proposed to achieve the fair allocation of link bandwidth in packet-switching networks.

In the traditional packet-switching network, a *flow* f is referred to as a sequence of packets $p_f^0, p_f^1, \dots p_f^n$ transmitted by a source. Each packet p_f^j is of length l_f^j. A flow is *backlogged* during the time interval $[t_1, t_2)$ if it has one or more outstanding packets waiting in the queue at any time $t \in [t_1, t_2)$.

All fair queueing algorithms maintain a system *virtual time*, $v(t)$, which intuitively measures the service received by a continuously backlogged flow in terms of bits forwarded. Each packet, p is associated two tags: *Start* tag $S(p)$ and a *Finish* tag $F(p) = S(p) + l_p$. The Start tag (a.k.a. packet's virtual starting time) is computed based on both the system virtual time and the Finish tag (a.k.a. packet's virtual finishing time). These algorithms schedule packets in the ascending order of either the Finish or Start tags.

In networking, fairness is simply defined as follows: for any two flows, f and g, that are backlogged during time interval $[t_1, t_2)$, we have

$$\left|W_f(t_1, t_2) - W_g(t_1, t_2)\right| \leq U(f, g), \qquad (1)$$

where $W_f(t_1, t_2)$ and $W_g(t_1, t_2)$ denote the service received in bits by flow f and g, respectively, during interval $[t_1, t_2)$, and $U(f, g)$ is a function of the properties of flows f and g (e.g., maximum packet length) and the system (e.g., link capacity).

[2] For a simple presentation, we consider an implementation that only uses continuous batching for decode steps but keeps the prefill step separated. as how TGI [21] adopted the original proposed iteration-level scheduling in Orca [50]. For more discussions, see Appendix C.1.

Intuitively, for packets-switching networks, the allocation of a link bandwidth is fair if, for any time interval during which two flows are backlogged, each of these flows receives approximately the same service in terms of the number of bits being forwarded during that interval. A scheduling algorithm is said to be *work-conserving* if a link always forwards packets when the queue is not empty [23].

There exists a distinct strand of research [3, 6, 47] focusing on the fair scheduling of preemptible tasks (e.g., CPU scheduling). The Completely Fair Scheduler (CFS) [1], implemented in Linux 2.6.23 and applying fair queuing to CPU scheduling, is closely related to our algorithm. In CFS, a "vruntime" is maintained for each task, and the task with the smallest "vruntime" is scheduled next. The tasks can be presented with a small time slice, aiming to maximize overall CPU utilization while also maximizing interactive performance.

2.3 Challenges

There are several unique challenges in LLM serving that prevent a direct application of fair-queuing-like algorithms. The first challenge is that the definition of fairness in the context of LLM serving is unexplored, and likely very different than that discussed in fair-queuing literature.

Traditional fairness is defined by measuring the cost of requests, which is usually a fixed value that is easy to estimate in either network or operating systems. For example, in networking, requests correspond to packets, and the cost is usually the number of bits of a packet. However, in LLM generations, how to define the cost of a request is not obvious. The cost per token can vary. Especially, processing an input (prompt) token is typically less expensive than processing an output token, as input tokens are processed in parallel while output tokens must be generated sequentially. Batching the output tokens from different requests can parallelize the fully connected layers but is still slower than processing input tokens for the attention layers.

Additionally, in LLM serving, the server has variable token-rate capacity, although the memory allocated for a batch is constant. Firstly, even if the request queue is not empty, we are not guaranteed that each batch is full. This is because we need to preserve spaces for future generated tokens, and also because the tokens added to the batch are not at the token but the request granularity. Secondly, the number of tokens processed highly depends on the requests' arrival patterns because of the continuous batching mechanism (Section 2.1). Furthermore, the capacity depends on the mix between input and output tokens of existing requests. If all requests have long past tokens, then the capacity is likely to be low (See Figure 2). Then there is no way to define a fixed amount of equal share.

The second challenge is the characteristic of unknown output length before finishing a request. This prevents a direct adaptation of classical algorithms like SFQ and Deficit Round Robin (DRR) [45] into the LLM serving. SFQ-style algorithms can provide good bounds in fairness by setting the Start and Finish tags through virtual time, as introduced in Section 2.2. However, computing Start and Finish tags requires knowing the request length in advance. DRR performs round-robin scheduling with a "deficit counter" mechanism to achieve fair scheduling of packets of variable length. In DRR, each client is assigned a specific quantum of service. It tracks the "deficit" of service for each client to ensure fairness over time. During each round, the scheduler allows each client to dispatch as many requests as possible, provided that the total length of these requests does not exceed the sum of the client's assigned quantum for that round and any accumulated deficit from previous rounds. Without knowing the length in advance, DRR cannot determine how many jobs can be scheduled within the quantum. Compared to CPU scheduling, although exploring adequate preemption is worthwhile in LLM serving, it cannot occur frequently. We need to define service fairness in LLM serving and operate at the granularity of individual tokens when frequent preemption is not possible. Additionally, the Completely Fair Scheduler (CFS) in CPU scheduling does not account for the concurrency of each task. It seeks fairness among individual tasks rather than among streams of tasks that can be executed concurrently. [3]

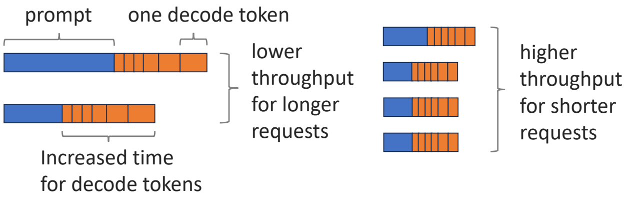

prompt
one decode token
lower throughput for longer requests
Increased time for decode tokens
higher throughput for shorter requests

Figure 2: An illustration of how request length can affect the cost and server capacity in terms of throughput. The visualized length is not precise but for illustration purposes only.

We will give the definition for LLM serving fairness in Section 3 and give a scheduling algorithm to achieve the LLM serving fairness in Section 4. We will outline our algorithm in a basic format for clarity, while details on its general form and integration with existing serving frameworks can be found in Appendix C.1. Although our algorithm is closely related to CFS, we also discuss the adaptation of DRR in Appendix C.2. Further discussions on future work are included in Appendix C.3.

3 Definition of Fairness in LLM Serving

In this section, we discuss the cost of a request, and the measurement of the service a client has received (Section 3.1). After defining the measurement of service, we can define fairness among clients in Section 3.2.

[3] Our algorithm does not consider preemption. Discussion about preemption is in Appendix C.3.

Notation	Explanation
$W_f(t_1,t_2)$	service received by f during interval $[t_1,t_2)$ (write as $W(t_1,t_2)$ when f is clear in the context)
n_p	number of processed input tokens
n_q	number of processed output tokens
w_p	weight of input tokens in the cost function
w_q	weight of output tokens in the cost function
$h(n_p,n_q)$	customized cost function
c_i	virtual token counter for client i
Q	waiting queue of requests to be processed
$i \in Q$	$\exists r \in Q$, r is a request from client i
L_{input}	maximum number of input tokens in a request
L_{output}	maximum number of output tokens in a request
M	maximum number of tokens that can be fitted in a running batch
U	invariant bound: $\max(w_p \cdot L_{input}, w_q \cdot M)$

Table 1: The upper half includes notations for service measurement. The lower half includes notations for the VTC algorithm and its analysis. The terms n_p, n_q can refer to either a single request or a single client, depending on the context.

3.1 Measurement of Service

In this subsection, we discuss the measurement of the service a client has received. Specifically, we define $W_f(t_1,t_2)$ and $W_g(t_1,t_2)$ from Equation (1) in the context of LLM serving. We omit the subscript and write $W(t_1,t_2)$ when the client is clear from the context or is irrelevant. The number of processed input and output tokens are denoted as n_p, n_q. Notations that will be introduced and used multiple times in this paper are summarized in Table 1.

Number of tokens A straightforward way to measure the service provided to a client is by summing the number of input tokens that have been processed and the number of output tokens that have been generated so far, i.e., $W(t_1,t_2) = n_p(t_1,t_2) + n_q(t_1,t_2)$ during the time window $[t_1,t_2)$.

Number of FLOPs Alternatively, one can measure the total FLOPs used in each stage, i.e., $W(t_1,t_2) = FLOP_{input}(t_1,t_2) + FLOP_{output}(t_1,t_2)$. This can be more precise because it captures the difference among tokens in attention computation, where tokens with longer prefixes require more computation.

However, both of these formulations cannot accurately reflect the actual LLM serving cost: The computation of the tokens at the prefill stage can be parallelized and achieve high GPU utilization. However, at the generation stage, we can only generate tokens one by one, as each token depends on all previous tokens as described in Section 2.1.

Weighted number of tokens To better reflect the actual LLM serving cost, a more accurate measure should capture the difference in costs of the prefilling and generation phases. One simple way to implement this idea is by using a weighted combination of the prefilling (input) tokens and decoding (output) tokens, inspired by the pricing mechanism used in OpenAI's API[4]. Formally, let w_p be the weight of input tokens and w_q be the weight of output tokens. Then, we have $W(t_1,t_2) = w_p \cdot n_p(t_1,t_2) + w_q \cdot n_q(t_1,t_2)$. Due to its simplicity, we will use this measure extensively in our analysis and evaluation.

Customized, unified representation. The definition of fairness in LLM serving can also be extended to other aspects, such as the weighted number of FLOPs or a more sophisticated method introduced in [31] that uses piecewise linear functions for the number of input and output tokens. Generally, the service can be represented as a function of the number of input and output tokens (n_p, n_q, respectively). Let $h(n_p,n_q)$ be the cost function that is monotonically increasing according to n_p and n_q. Our method can easily accommodate different h (Section 4.2).

3.2 Fairness in LLM Serving

In this paper, we apply fair sharing to the domain of LLM serving to provide performance isolation across multiple clients sharing the same LLM server. In particular, we employ the classic formulation of max-min fairness [5], which computes a *fair share* for the clients sharing a given server. In a nutshell, given the metric of service fairness, if a client sends requests at no more than its fair share, all its requests are served. In contrast, if a client sends requests at more than its fair share, its excess requests will be delayed or even dropped. As a result, a misbehaving client cannot deny the service to other clients, no matter how many requests it sends. To achieve max-min fairness, an idealized serving system follows desirable properties as below:

1. **Backlogged clients** Any two clients f,g that are continuously backlogged during a given time interval $[t_1,t_2)$ should receive the same service during this interval, i.e. $W_f(t_1,t_2) = W_g(t_1,t_2)$.

2. **Non-backlogged clients** Client f that is continuously backlogged during time interval $[t_1,t_2)$ should not receive less service than another client, g, that is not continuously backlogged during the same time interval, i.e., $W_f(t_1,t_2) \geq W_g(t_1,t_2)$.

3. **Work-conservation** As long as there are requests in the queue, the server should not be idle.

The first property means that two clients sending requests at more than their fair share will get the same service, regardless of the discrepancy between their sending rates. The second property says that a client sending requests at a higher rate will not get less service than a client sending at a lower rate. Basically, the first two properties say that a misbehaving client is contained (i.e., doesn't receive more service than other backlogged clients), and not punished (i.e., doesn't receive

[4] https://openai.com/pricing

less service than other non-backlogged clients). Finally, the work conserving property aims to maximize the utilization, addressing a key weakness of the RPM-based solutions.

The three properties above assume an idealized fair serving system. A practical system will approximate these properties. In general, the best we can achieve is deriving bounds that are independent of the length of the time interval, e.g., in the first property, the difference between $W_f(t_1,t_2)$ and $W_g(t_1,t_2)$ is bounded by a value that is independent of $t_2 - t_1$. We give the formal guarantees provided by our method in Section 4.1.

4 Achieving Fairness

In this section, we present our algorithm VTC with proved fairness properties in Section 4.1, and show its generalization for customized service measurement in Section 4.2. Variants of VTC, including weighted VTC and VTC with length prediction, are introduced in Section 4.3 and Section 4.4.

4.1 Virtual Token Counter (VTC)

Based on insights from prior discussions, we've identified key challenges inherent in large language model (LLM) serving that hinder direct adaptation of existing algorithms to deliver approximately fair LLM service. We then propose the Virtual Token Counter (VTC), a mechanism for achieving fair sharing in LLM Serving (Algorithm 2). To quantify the service received by a client we use the weighted number of tokens metric, as described in Section 3.1. We discuss the generalization to other metrics in Section 4.2.

Intuitively, VTC tracks the services received for each client and will prioritize the ones with the least services received, with a counter lift each time a client is new to the queue. The *counter lift* is needed to fill the gap created by a low load period of the client, so that it will not be unfairly served more in the future. In other words, the credits for a client are utilized immediately and cannot be carried over or accumulated. The virtual counters are updated each time a new token is generated, which can reflect the services received instantly. This operates at the token-level granularity, and thus addresses the unknown length issue. VTC can be easily integrated into the continuous batching mechanism, and its scheduling mechanism does not depend on the server's capacity, overcoming the problem of variable token-rate capacity.

Algorithm 2 shows how VTC can be implemented in the continuous batching framework described in Section 2.1. A more general integration for VTC is described in Appendix C.1. It maintains a virtual counter for each client, denoted as $\{c_i\}$. The counters are initialized as 0 (line 2). The program runs with two parallel streams.

The monitoring stream listens to the incoming requests, described in lines 5-14. The new request will be added to the waiting queue Q immediately. If the new request is the only request in Q for its sender client, a counter lift (lines 8-13) will happen. Because this client could have been underloaded before, its counter could be smaller than the other active counters. However, since the credits cannot be carried over, we need to lift it to the same level as other active counters, thus maintaining fairness among this client and others. Lines 9-10 address the scenario where the entire system was in an idle state. We do not reset all the counters to avoid nullifying a previously accumulated deficit upon a system restart.

Algorithm 2 Virtual Token Counter (VTC)

```
Input: request trace, input token weight $w_p$, output token
    weight $w_q$, upper bound from Equation (2) denoted as $U$.
 1: let current batch $B \leftarrow \emptyset$
 2: let $c_i \leftarrow 0$ for all client $i$
 3: let $Q$ denote the waiting queue, which is dynamically
    changing.
 4: ▷ with monitoring stream:
 5: while True do
 6:     if new request $r$ from client $u$ arrived then
 7:         if not $\exists r' \in Q, client(r') = u$ then
 8:             if $Q = \emptyset$ then
 9:                 let $l \leftarrow$ the last client left $Q$
10:                 $c_u \leftarrow \max\{c_u, c_l\}$
11:             else
12:                 $P \leftarrow \{i \mid \exists r' \in Q, client(r') = i\}$
13:                 $c_u \leftarrow \max\{c_u, \min\{c_i \mid i \in P\}\}$
14:         $Q \leftarrow Q + r$
15: ▷ with execution stream:
16: while True do
17:     if can_add_new_request() then
18:         $B_{new} \leftarrow \emptyset$
19:         while True do
20:             let $k \leftarrow \arg\min_{i \in \{client(r) \mid r \in Q\}} c_i$
21:             let $r$ be the earliest request in $Q$ from $k$.
22:             if $r$ cannot fit in the memory then
23:                 Break
24:             $c_k \leftarrow c_k + w_p \cdot input\_length(r)$
25:             $B_{new} \leftarrow B_{new} + r$
26:             $Q \leftarrow Q - r$
27:         forward_prefill($B_{new}$)
28:         $B \leftarrow B + B_{new}$
29:     forward_decode($B$)
30:     $c_i \leftarrow c_i + w_q \cdot |\{r \mid client(r) = i, r \in B\}|$
31:     $B \leftarrow$ filter_finished_requests($B$)
```

The execution stream is the control loop of an execution engine that implements continuous batching. Line 17 controls the frequency of adding a minibatch B_{new} of new requests into the running batch B. Commonly, the server will add a new minibatch after several decoding steps. The minibatch B_{new} is constructed by iteratively selecting the request from the client with the smallest virtual counter (lines 20-26). The counters will be updated when adding new requests according to the service invoked by the input tokens (line 24). After each decoding step (line 29), $\{c_i\}$ will be updated immediately according to the service invoked by the newly generated output tokens (line 30).

The VTC algorithm is (mostly) work-conserving because it only manipulates the dispatch order and does not reject a request if it can fit in the batch.

4.1.1 Fairness for backlogged clients in VTC

In this subsection, we provide the theoretical guarantee for fairness among overloaded clients in VTC. More precisely, the overload of a client is reflected by its backlog, which can be formally defined as follows. Intuitively, a client being backlogged means its requests are queued up.

Definition 4.1 (Backlog)**.** A client f is backlogged during time interval $[t_1,t_2)$, if at any time $t \in [t_1,t_2)$, f has a request that is waiting in the queue.

We adapt the traditional definition of fairness for backlogged clients in the network to our scenario. The following definition formally defined the item 1 introduced in Section 3.2, that for any interval, and any two continuously backlogged clients during the time interval, the difference of their received service should be bounded by a value that is independent of the interval length.

Definition 4.2 (Fairness adapted from [16])**.** Let $W_f(t_1,t_2)$ be the aggregated service received by client f in the interval $[t_1,t_2)$. A schedule is fair w.r.t. δ, if for any clients f and g, for all intervals $[t_1,t_2)$ in which clients f and g are backlogged, we have $|W_f(t_1,t_2) - W_g(t_1,t_2)| \leq \delta$.

In the rest of the paper, as in Algorithm 2, we let Q denote the set of requests in the waiting queue. We abuse the notation of $i \in Q$ for a client i to indicate there exists $r \in Q$, such that r is a request from client i. Let L_{input} and L_{output} be the maximum number of input and output tokens in a request. Let M be the maximum number of tokens that can be fitted in a running batch. Lemma 4.3 reflects the core design of Algorithm 2, that the virtual counters for active clients are chasing each other to ensure their maximum difference is bounded. The missing proof for Lemma 4.3 and all following theorems are in Appendix A.

Lemma 4.3. *The following invariant holds at any time in Algorithm 2 when $Q \neq \emptyset$:*

$$\max_{i \in Q} c_i - \min_{i \in Q} c_i \leq \max(w_p \cdot L_{input}, w_q \cdot M) \tag{2}$$

We then introduce our main theorem which provides a bound for Definition 4.2.

Theorem 4.4 (Fairness for overloaded clients)**.** *For any clients f and g, for any time interval $[t_1,t_2)$ in which f and g are backlogged, Algorithm 2 guarantees* [5]

$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \leq 2\max(w_p \cdot L_{input}, w_q \cdot M).$$

Proof. For any f, if f is backlogged during time t_1 to t_2, we have $W_f(t_1,t_2) = c_f^{(t_2)} - c_f^{(t_1)}$. This is because the line 7 will not be reached for client f during t_1 to t_2, and the c_f keeps increasing during t_1 to t_2 by adding w_p product the number of served input tokens and w_q product the number of served output tokens. By Lemma 4.3, from Equation (2), we have

$$\begin{aligned} |W_f(t_1,t_2) - W_g(t_1,t_2)| &\leq |c_f^{(t_1)} - c_g^{(t_1)}| + |c_f^{(t_2)} - c_g^{(t_2)}| \\ &\leq 2\max(w_p \cdot L_{input}, w_q \cdot M) \end{aligned}$$

□

Remark 4.5. An empirical illustration of this theorem can be found in Figure 3a, where the difference between services received by backlogged clients is bounded, regardless of how long they have been backlogged.

Remark 4.6. Line 13 can be modified to take any value between $\min\{c_i | \exists r' \in Q, client(r') = i\}$ and $\max\{c_i | \exists r' \in Q, client(r') = i\}$. The proof of Theorem 4.4 should still hold.

Remark 4.7. To tighten the bound in Theorem 4.4, we can restrict the memory usage for each client in the running batch. This might compromise the work-conserving property, as we will demonstrate in Theorem 4.8. Therefore, there is a *trade-off* between achieving a better fairness bound and maintaining work conservation. Heuristically, predicting the request length in advance could result in a smaller discrepancy, as detailed in Section 4.4. Additionally, preemption is another method to achieve smaller differences, discussed in Appendix C.3.

We also prove in the next theorem that the bound in Theorem 4.4 is tight within a factor of 2 for a family of work-conserving schedulers. We say a scheduler is work-conserving if it stops adding requests to a partially-filled minibatch (line 22 in Algorithm 2) only when it runs out of memory[6] but not for fairness reasons.

Theorem 4.8. *For any work-conserving schedule without preemption, there exists some query arrival sequence such that for client f,g and a time period t_1,t_2, such that*

$$|W_f(t_1,t_2) - W_g(t_1,t_2)| \geq w_q \cdot M,$$

where clients f,g are backlogged during the time $[t_1,t_2)$.

As we mentioned before, output tokens are more expensive than input tokens, so normally we have $w_q > w_p$. Therefore the right-hand side of the inequality in Theorem 4.4 is $2w_q \cdot M$, which is 2× of the lower bound in Theorem 4.8.

[5] The service of a served request incurred by pre-filling (service for input tokens) is counted at the time when the request is added to the running batch (line 24 in Algorithm 2), rather than the time when prefill is finished. This is because we want to count the input tokens immediately to avoid selecting all the same k at line 20 in Algorithm 2 for B_{new}.

[6] Different implementation may have different criteria of "not enough memory". This can only be achieved heuristically because the number of output tokens is unknown before it finishes.

4.1.2 Fairness for non-backlogged clients in VTC

In this subsection, we discuss item 2 in Section 3.2. A backlogged client will not receive less service than another client. This can be reflected in the following theorem.

Theorem 4.9. *If a client f is backlogged during time interval $[t_1,t_2)$, for any client g, there is*

$$W_f(t_1,t_2) \geq W_g(t_1,t_2) - 4U.$$

Here U is the upper bound from Equation (2).

In addition to that, clients who send requests constantly less than their share should have their requests serviced nearly instantly. This property intuitively can be implied by the first item in Section 3.2, as if a low-rate client cannot be served on time, it becomes backlogged, which requires the same level of service with backlogged clients. We formally prove this property to offer a fairness assurance for clients who are not overloaded. This intuitively acts as a safeguard against misbehaving clients [10].

We start with Definition 4.10 and Theorem 4.11 discussing the aspect of latency bounds. Intuitively, if a client is not backlogged and has no requests running, the next request from it will be processed within a latency bound that is independent of the request rate of other clients.

Definition 4.10. Assume there are n active clients during $[t_1, t_2)$, and the server capacity at time $t \in [t_1,t_2)$ is defined as $S(t)$, where

$$\int_{t_1}^{t_2} S(t)\,dt = \sum_{i=1}^{n} W_i(t_1,t_2)$$

Because the server capacity is always positive and bounded, there exists $a,b \in \mathbf{R}^+$ such that $\forall$ t, $a < S(t) \leq b$.

Theorem 4.11. *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1,t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded:*

$$D(r_f) - A(r_f) \leq 2\cdot(n-1)\cdot\frac{\max(w_p\cdot L_{input}, w_q\cdot M)}{a} \tag{3}$$

Here a is the lower bound of the capacity in Definition 4.10.

Remark 4.12. The bound in Theorem 4.11 is irrelevant to the request rate of others, giving an upper bound for latency against ill-behavior clients.

The above is about one request not getting delayed. The following theorem shows that during time period $[t_1,t_2)$, if there are n active clients sending requests, and client f is sending requests with a rate constantly less than $1/n$ of the server's capacity (with some constant gap), client f should have all its requests been served.

Theorem 4.13. *(Fairness for non-overloaded clients) For any time interval $[t_1,t_2)$, we claim the following.*

Assume a client f is not backlogged at time t_1 and for any time interval $[t,t_2), t_1 \leq t < t_2$, f has requested services less than $\frac{T(t,t_2)}{n(t,t_2)} - 5U$, where $T(t,t_2)$ is the total services received for all clients during the interval $[t,t_2)$, $n(t,t_2)$ is the number of clients that have requested services during the interval, and U is the upper bound from Equation (2).

Then, all of the services requested from f during the interval $[t_1,t_2)$ will be dispatched.

4.2 Adapt to Different Fairness Criteria

Algorithm 2 is designed for fairness with the service function $W(t_1,t_2)$ as a linear combination of the number of processed input tokens and the number of generated tokens. For a different definition of $W(t_1,t_2)$, Algorithm 2 can be easily modified to update the counter according to the other definitions described in Section 3.1.

Assume we aim for fairness using $\sum_r h(n_p^r,n_q^r)$ as the metric of service, where h is a specific function. In this context, r indexes the served requests, and n_p^r, n_q^r represent the number of input and output tokens served for request r, respectively. Line 24 will be changed to

$$c_k \leftarrow c_k + h(n_p^r, 0).$$

Line 30 will be changed to

$$c_i \leftarrow c_i + \sum_{r|client(r)=i, r\in B} \left(h(n_p^r,n_q^r) - h(n_p^r,n_q^r-1)\right).$$

The fairness bound will also be changed according to $h(\cdot,\cdot)$. Under the assumption that output tokens are more expensive than input tokens, the bound will become the maximum value of aggregated $h(\cdot,\cdot)$ for a set of requests that can be fitted in one running batch. Algorithm 4 in Appendix C.1 is the pseudocode of a general VTC framework.

4.3 Weighted VTC

VTC can be applied when clients have tiers. Similar to weighted fair queuing, clients can have different weights to represent their priority in service. If a client f has a weight w_1, that is twice the weight w_2 of client g, client f is expected to receive twice the service than client g. When they are continuously backlogged during the interval $[t_1,t_2)$, we want $\left|\frac{W_f(t_1,t_2)}{w_1} - \frac{W_g(t_1,t_2)}{w_2}\right|$ to be bounded instead of $\left|W_f(t_1,t_2) - W_g(t_1,t_2)\right|$.

Weighted VTC can be easily implemented by modifying the lines that update the virtual tokens. For example, the line 22 in Algorithm 4 will be changed to

$$c_i \leftarrow c_i + \frac{\sum_{r|client(r)=i}\left(h(n_p^r,n_q^r) - h(n_p^{r(old)},n_q^{r(old)})\right)}{w_i}.$$

Here c_i is the virtual counter of client i, and w_i is its corresponding weight.

4.4 VTC with Length Prediction

As mentioned in Remark 4.7, using VTC with length prediction can heuristically reduce the service discrepancy. In standard VTC, the counters only reflect served tokens. Tokens generated in the future can only be passively added to the counter. This results in a large service discrepancy because requests are overly added due to underestimation of their costs at the time of prompting, leading to the forced serving of over-compensated output tokens. Incorporating a prediction mechanism can help reduce this variance.

The theoretical worst-case scenario won't change, according to the lower bound proved in Theorem 4.8. But practically, the average-case service discrepancy could be smaller.

The modified pseudocode of VTC with length prediction is described in Algorithm 3 in Appendix B.3. Intuitively, when a request r is selected, the cost associated with the predicted number of output tokens is immediately added to the virtual counter of the client sending the request. During the actual decoding process, adjustments are made to the virtual counter based on the actual number of output tokens produced. If the actual number of tokens exceeds the prediction, the virtual counter is increased accordingly. Conversely, if fewer tokens are generated than predicted when finished, the virtual counter is reduced. The effectiveness of the length predictor is contingent upon both the workload and the accuracy of predictions, as demonstrated in our evaluations.

5 Evaluations

In this section, we evaluate VTC against other alternatives under different workloads. The results confirm the fairness properties introduced in Section 3 of VTC, and show that all other alternatives will fail in at least one workload.

5.1 Setup

Implementation We implement our VTC and other baseline schedulers in S-LoRA [43], a system that serves a large amount of LoRA adapters concurrently. Its backbone is a general serving system adapted from LightLLM [29]. It includes the implementation of continuous batching [50] and PagedAttention [24][7]. Our VTC scheduler is built on top of those two techniques. Our implementation is elegant and can be implemented as a thin layer on top of the existing scheduler, it contains only about 100 lines of code on top of S-LoRA. The simplicity demonstrates its wide applicability. Fairness can be considered among general clients, and our experiments are done in this way. But we would like to note that fairness also could be taken into consideration among adapters, especially under the scenario of personalization that uses one adapter per customer, which originally motivated this paper.

Baselines In this section, we benchmark VTC and the baselines as below:

- First Come First Serve (FCFS): In the First-Come-First-Serve method, requests are handled strictly in the order they are received, irrespective of the requesting client. This is the default scheduling strategy in many prevalent LLM serving systems, including vLLM [24] and Huggingface TGI [20].
- Request per minute (RPM): This method limits the maximum number of requests that a client can make to the server within a one-minute timeframe. The definition of service corresponds to Section 3. When a client exceeds this limit, subsequent requests are blocked until the limit resets at the start of the next minute.
- Least Counter First (LCF): This is a variant of VTC without the counter lift component. Each client will maintain a counter for the service it received so far. The request from the client with the smallest counter will be scheduled each time.

We also benchmark the VTC with length predictions as described below:

- VTC (predict): This variant of VTC, detailed in Algorithm 3, utilizes the average output length of the last five requests from each client to predict the output length.
- VTC (oracle): This variant employs a hypothetical output length predictor that achieves 100% accuracy.

Synthetic Workload We run Llama-2-7b on A10G (24GB), using the memory pool for the KV cache with size 10000[8]. We use various workloads to demonstrate different aspects of fairness, and compare VTC with other baselines. The detailed results are in Section 5.2. We start with synthetic workloads to give a clear message for fairness properties.

Real Workload To validate the effectiveness of VTC in more complex real-world scenarios, we also experiment with VTC and other baselines under workloads constructed from the trace log of LMSYS Chatbot Arena [52, 53], which is an LLM serving platform for real-world clients.

Ablation Study In the ablation study, to evaluate the impact of different memory pool sizes and request lengths on the scheduling fairness, we run Llama-2-13b on A100 (80GB) with a memory pool of size 35000 and 65000 respectively. For each memory pool size, we evaluate the absolute difference in the accumulated service of two clients.

[7] with block size equals 1.

[8] There are in total 10000 tokens for KV cache that can be stored on GPU.

Metrics We apply the weighted number of tokens described in Section 3.1 as the measurement of services in our evaluation. Following OpenAI pricing, we set $w_p = 1$ and $w_q = 2$.

- The service received by client i at time t is measured as $W_i(t-T, t+T)$ for a certain T.
- The absolute difference in service between clients is quantified based on accumulated services, represented as $\max_{i,j} |W_i(0,t) - W_j(0,t)|$.
- The response time of client i at time t is measured as the average first token latency of the requests sent by client i during the time window $[t-T, t+T]$.

In all settings, we set $T = 30$ seconds.

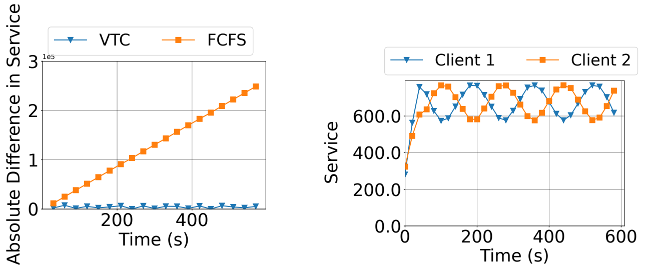
We employ *service difference* as a quantitative metric to assess the deviation from ideal fairness. A smaller difference in service indicates more equitable scheduling. Formally, the service difference between two clients is defined as the minimum of two values: the difference between their received services, and the difference between the lower service and its corresponding request rate. For example, consider two clients that received services s_1 and s_2, such that $s_1 \leq s_2$, and let r_1 denote the request rate sent from the first client. Then the service difference is defined as $\min(s_2 - s_1, |r_1 - s_1|)$.

VTC Variants The experiments for weighted VTC are presented in Appendix B.1, demonstrating its capability to serve clients with varying priorities. To illustrate the versatility of the service function beyond the linear model used in our primary analysis, we evaluate a profiled service cost function in Appendix B.2, which is a quadratic function. Additional experiments on VTC with length prediction are detailed in Appendix B.3.

5.2 Results on Synthetic Workloads

We design a set of experiments to visualize the fairness properties of VTC. We start with synthetic traces to show plots reflecting the ideal case's fairness. We experiment from the simplest setting, where clients send requests following a uniform distribution with the same input and output length, to complex settings, where requests arrive stochastically, with various input and output lengths.

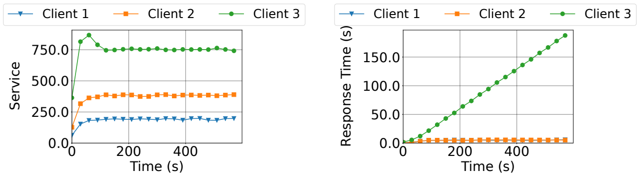
Constant request rates We start with scenarios where requests arrive deterministically with the same input and output length. In Figure 3, two clients send requests at different rates, but are both constantly overloaded. In this case, Figure 3a shows VTC can keep the difference between services received by both clients to be small. FCFS cannot maintain fairness, which always serves more for the client who is sending requests at a higher rate. Figure 3b shows the real-time received service rate for two clients in VTC, which confirms that the two received the same level of services at any time interval. This experiment empirically validates Theorem 4.4.



(a) Absolute difference for accumulated service

(b) Received service rate, calculated as an average of 60s time windows. (VTC)

Figure 3: Two clients with different request rates and both overloaded. Client 1 sends 90 requests per minute. Client 2 sends 180 requests per minute, both evenly spaced out so that each request is sent at a consistent time interval throughout the minute. Every request has input lengths of 256 and output lengths of 256. Both clients are backlogged because they exceed the server capacity.



(a) Received service rate (VTC).

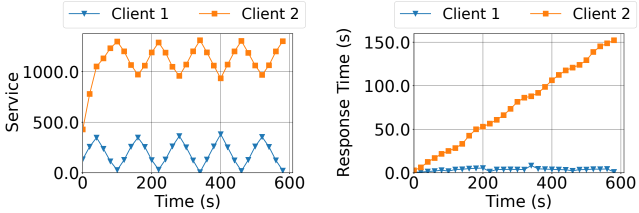
(b) Response time (VTC).

Figure 4: Client 3 who is overloaded can consume more than its share as Clients 1 and 2 are sending requests lower than their share. Clients 1, 2, and 3 send 15, 30, and 90 requests per minute, respectively, under uniform distribution. Requests have input lengths of 256 and output lengths of 256. Client 3 is backlogged, while Clients 1 and 2 are not.

In Figure 4, three clients send requests at around $2/13$, $4/13$, and $> 7/13$ of the server's capacity, respectively. In this case, Clients 1 and 2 can be served immediately when their requests arrive (Figure 4b), and Client 3 will consume the remaining capacity (more than 1/3), which is an empirical illustration of the work-conserving property of VTC. The service received for Client 1 and Client 2 have a ratio $1:2$, which is consistent with their request rates (15 versus 30).

ON/OFF request pattern In real-world applications, clients usually do not always send requests to the server. They may occasionally be idle ("OFF" phase). We call this the "ON/OFF" pattern. In Figure 5, Client 2 is always in the "ON" phase, sending requests at a rate of 120 per minute. Client 1 sends 30 requests per minute (less than half of the capacity) during the ON phase and switches to OFF phase periodically. Since Client 1 uses less than half the system capacity when it is in the ON phase, its requests are mostly processed before it switches to the OFF phase (Figure 5b). When it is in the OFF phase, Client 1 thus takes all the system capacity. The total service rate remains the same, which confirms VTC's flexibility in achieving work-conserving.

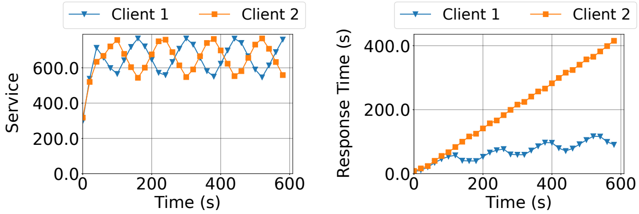
On the contrary, in Figure 6, client 1 sends much more than half the capacity during the ON phase, and makes itself always backlogged. Thus, even when it is in the OFF phase, it is still in the backlog status. In this case, Client 1 and Client 2 should still receive the same level of service rate.



(a) Received service rate (VTC).

(b) Response time (VTC).

Figure 5: ON/OFF request pattern. Client 1 sends 30 requests per minute (less than half of the capacity) during the ON phase and switches to OFF phase periodically. Client 2 is always in the ON phase, sending requests at a rate of 120 requests per minute (larger than half of the capacity). Requests have input lengths of 256 and output lengths of 256.

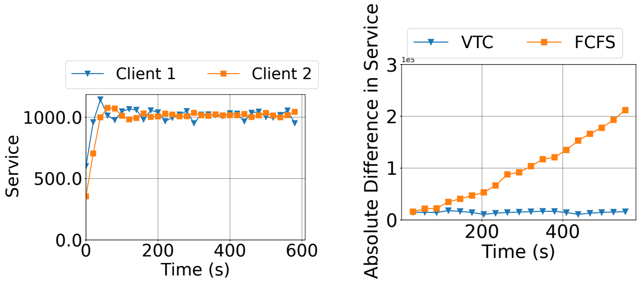


(a) Received service rate (VTC).

(b) Response time.

Figure 6: ON/OFF request pattern. Client 1 sends 120 requests per minute constantly during the ON phase (over its share), and stops sending during the OFF phase. Client 2 sends 180 requests per minute all the time (over its share). Requests have input lengths of 256 and output lengths of 256.

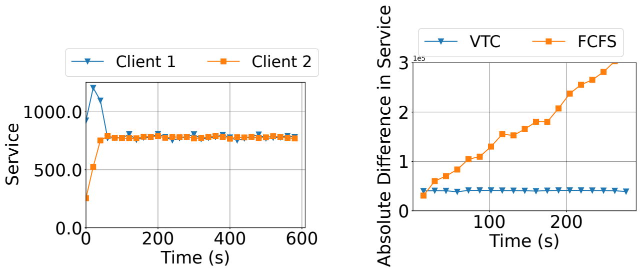
Variable input/output length and poisson process In this experiment, we simulate scenarios where requests arrive stochastically. Furthermore, they send requests with different input and output lengths. In both Figure 7 and Figure 8, Client 1 sends requests with a high rate and Client 2 sends requests with a rate lower but still over its share. Requests arrive according to a Poisson process with the coefficient of variance 1. In Figure 7, client 1 sends short requests, and client 2 sends long requests. In Figure 8, Client 1 sends requests with short input and long output, while Client 2 sends requests with long input and short output. Similarly, with the observation before, VTC maintains a bounded difference between the services received by two clients. FCFS cannot preserve fairness according to Figure 7b and Figure 8b. This confirms that VTC can work under stochastic workloads with variable lengths.



(a) Received service rate (VTC).

(b) Absolute difference for accumulated service.

Figure 7: Client 1 sends 480 requests per minute. Client 2 sends 90 requests per minute. Requests arrive according to a Poisson process with the coefficient of variance 1. Requests sent from Client 1 have input lengths of 64 and output lengths of 64. Requests sent from Client 2 have input lengths of 256 and output lengths of 256.



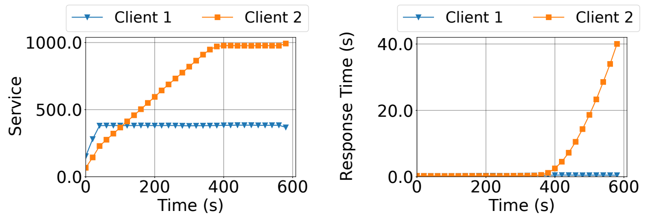
(a) Received service rate (VTC).

(b) Absolute difference for accumulated service.

Figure 8: Client 1 sends 480 requests per minute. Client 2 sends 90 requests per minute. Requests arrive according to a Poisson process with the coefficient of variance 1. Requests sent from Client 1 have input lengths of 64 and output lengths of 512. Requests sent from Client 2 have input lengths of 512 and output lengths of 64.

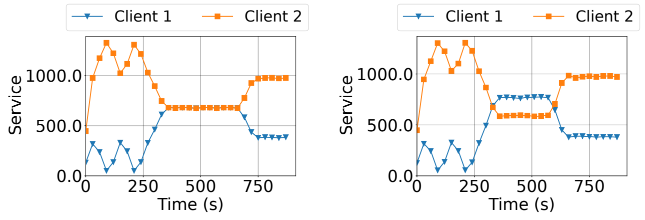
Isolation To illustrate the isolation property, we use the setup with a deterministic arrival pattern and the same input length and output length of 256. In Figure 9, Client 1 sends 30 requests per minute, which is under half of the server's capacity. Client 2 acts as an "ill-behaved" client. It sends requests at a linearly increasing rate, and gradually over half of the system capacity. We observe that the response time of requests from client 1 is roughly unchanged, empirically validating the property stated in Theorem 4.13.

Distribution shift In reality, clients' behavior may change over time. To this end, we evaluate the robustness of VTC



(a) Received service rate (VTC). (b) Response time (VTC).

Figure 9: Client 1 sends 30 requests per minute, Client 2 sends 120 requests per minute, in a uniform arrival pattern. Requests have input lengths of 256 and output lengths of 256. Client 1 sends 30 requests per minute, which is under half of the server's capacity. Client 2 sends requests at a linearly increasing rate, and gradually over half of the system capacity.



(a) Received service rate (VTC). (b) Received service rate (LCF).

Figure 10: Clients send requests in three phases, all with uniform arrival patterns. The first 5 minutes is ON/OFF phase. Client 1 sends 30 requests per minute during the ON phase (less than its share) and stops sending during the OFF phase. Each ON or OFF phase has 60 seconds. The second 5 minutes is the overload phase. Both Client 1 and Client 2 send 60 requests per minute, which causes the server to be overloaded. In the last 5 minutes, Client 1 sends 30 requests per minute (less than its share), and Client 2 sends 90 requests per minute, which causes the server to be still overloaded. Requests all have input lengths of 256 and output lengths of 256.

when the distribution of client requests shifts. In Figure 10, we construct a 15-minute workload comprising three phases. The first phase is an ON/OFF phase, in which Client 1 sends requests less than its share only during the ON phase and stops during the OFF phase. Client 2 sends requests at a constant rate, which makes the server overloaded. We can observe the pattern for the first phase to be similar to Figure 5a, which maintains a constant total service. During the second phase, because the two clients both send requests over their share, a fair server should let them receive the same level of service. Figure 10a demonstrates that VTC yields a desired pattern, similar to that shown in Figure 3b. Figure 10b reveals that LCF disproportionately serves Client 1, as it inherits Client 1's deficit from the first phase. In the last phase, the serving pattern for VTC and LCF are similar, because they simply serve all requests from Client 1 immediately as Client 1 sends requests under its share.

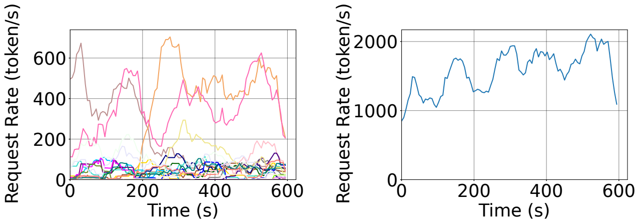


Figure 11: Request rate distribution during the sampled 10 minutes duration with re-scale. The figure on the left denotes the real-time request rate for the 27 clients. A few clients have sent many more requests than others, reflecting the original trace of a few most popular models. The figure on the right depicts the total request rate from all 27 clients.

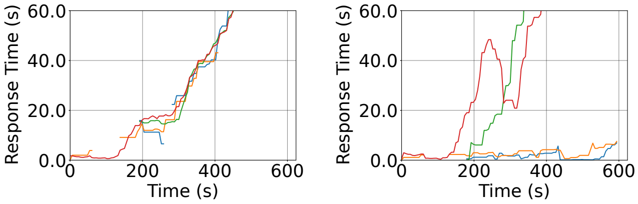


Figure 12: Response time of 4 selected clients when using FCFS (Left) and VTC (Right) in real traces. Each curve corresponds to one client. There are some curves that show disconnected because, during some periods, a client may have no requests served. Requests distribution see Figure 11.

5.3 Results on Real Workloads

We construct real workload traces from the traces of LMSYS Chatbot Arena [52, 53], following a similar process in [43]. The trace is from a server that serves multiple LLMs. To adapt it to our setting, we treat each LLM as a client. In total, there are 27 clients. To sample from this log, we define D, the duration, and R, the request rate. We then sample $R*D$ requests from the trace, and re-scale the real-time stamps to $[0,D]$. We use a duration of 10 minutes to be consistent with previous experiments, and a request rate of 210 requests per minute for the whole system. With the adapted workload, we run Llama-2-7b on A10G (24GB). In summary, the prompts from the 27 clients are collected from the real world interactions, which will be sent to the server for inference on Llama-2-7b. The timestamps are re-scaled from the real-world trace.

For better visualization of the evaluation results, we select two clients that send the most requests and two clients that send a medium number of requests. We sort 27 clients according to the number of requests they send, and depict the statistics of the $13^{th}, 14^{th}$ and $26^{th}, 27^{th}$ clients. We do not choose clients that send the least requests because they typically only send requests in a small interval.

Request distribution The request rate distribution is visualized in Figure 11. The request rate of individual clients

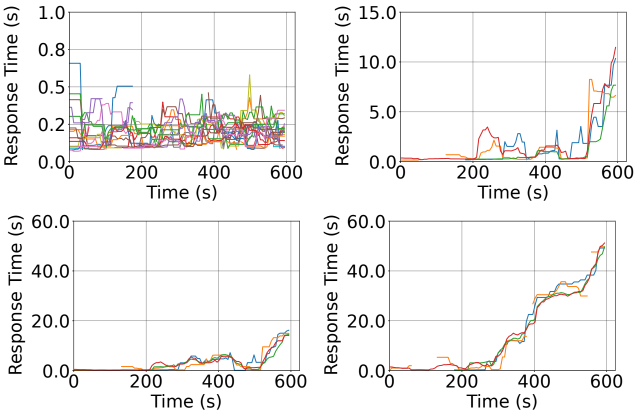


Figure 13: Response time of 4 selected clients (all 27 clients when rpm=5) when using RPM in real traces. Left-upper to right-bottom corresponds to a different rate limit (5, 15, 20, 30 requests per minutes, respectively). There are some curves that show disconnected because, during some periods, a client may have no requests served.

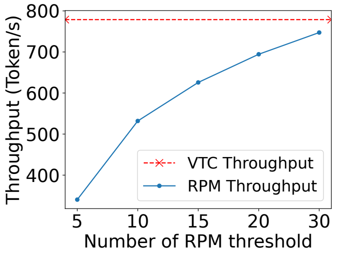


Figure 14: Throughput of RPM versus different number of requests per minute threshold. Compared with VTC, RPM consistently exhibits a lower throughput.

and the total request rate are all highly dynamic. The input and output length distribution is depicted in Figure 20 in the Appendix. The average input length is 136, and the average output length is 256. The input and output lengths have the range of $[2,1021]$ and $[2,977]$, respectively.

Effect on response time Figure 12 shows the response time of 4 selected clients on the real trace. With FCFS scheduling, the response time of all clients increases drastically because some clients send over their share, monopolizing the service and impacting other clients. With VTC, only clients that send requests over its share will have a drastic increase in the response time.

Analysis of request rate per limit approach In Figure 13, we show the response of RPM approach with different rate limits. In Figure 14, we show the corresponding throughput comparison with VTC. These plots reveal a core dilemma of the RPM approach - the system has to choose between fairness or throughput, but not both. If the rate limit is low, then the system rejects many requests from clients that send over their share. This opens the capacity for clients with fewer requests. As demonstrated in the uppermost plot in Figure 13, all requests have a similar response time. However, this low rate limit rejects more requests than needed, causing a lower throughput (cluster-wise throughput is ≈ 340 output tokens per second when RPM=5, as opposed to ≈ 779 tokens per second in VTC or FCFS). When the rate limit is set higher, the system throughput is gradually increasing, i.e., increasing from 340 tokens to 747 tokens per second. However, the response time for all requests grows up. When the request rate is set higher and higher, the response time curve converges to the one in FCFS, and there is no fairness guarantee anymore. In other words, the RPM approach can be summarized as follows: it functions as an FCFS (First-Come, First-Served) approach with admission control (rate limiting), rather than as a truly fair scheduler. Its fairness is achieved by rejecting numerous requests from other clients, which compromises the overall system throughput.

Quantitative Measurement We measured the maximum and average service difference described in Section 5.1 during the time window (10 minutes) in which we ran the experiments. Table 2 is a summary for all baselines using this quantitative measurement for real workload trace.

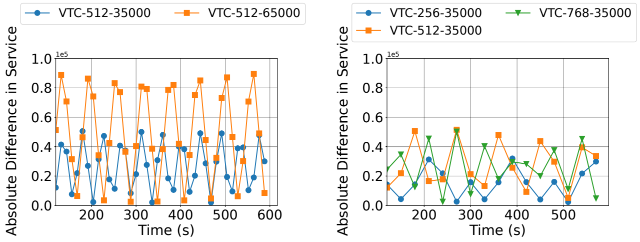
Scheduler	Max Diff	Avg Diff	Diff Var	Throu	Isolation
FCFS	759.97	433.53	32112.00	777	No
LCF	750.49	323.82	29088.90	778	Some[9]
VTC	368.40	251.66	6549.16	779	Yes
VTC(predict)	365.47	240.33	5321.62	773	Yes
VTC(oracle)	**329.46**	**227.51**	**4475.76**	**781**	**Yes**
RPM(5)	143.86	83.58	1020.46	340	Some
RPM(20)	446.76	195.71	7449.79	694	Some
RPM(30)	693.66	309.45	24221.31	747	Some

Table 2: The service difference is counted by summing the service difference between each client and the client who received the maximum services. Throughput is the total number of tokens (including input and output tokens) processed divided by the total execution time.

5.4 Ablation Study

In Figure 15, we evaluate how different memory pool sizes and request lengths will affect scheduling fairness. As shown in Figure 15a, with a larger memory pool size, the attainable batch size becomes larger. Therefore, there is greater variation in the absolute difference of accumulated services received by the clients when the memory pool is 65000 than that is 35000, which empirically validates Theorem 4.4. Figure 15b demonstrates that larger request lengths will also lead to greater variations in the service difference. This is

[9] LCF achieves isolation if the workload does not change. However, the isolation can be broken by newly joined clients whose virtual counter is lagging behind.



(a) Different memory pool size. (b) Different request length.

Figure 15: In all settings, both clients are sending requests of the same lengths with uniform arrival patterns. They send requests with different request rates but are both backlogged. Three different request lengths (256*2, 512*2 and 768*2) are evaluated for the 35000 KV cache setting.

caused by the unknown output length of request generation. At line 24 in Algorithm 2, the most conservative way of only counting the input tokens leads to over-compensation for the smallest counter, as all the potential output tokens are not counted. A shorter request length has a milder effect of over-compensation. The curves of $(512*2)$ and $(768*2)$ show the same variance. This is because at length $(512*2)$, the upper bound given by VTC has been reached.

6 Related Works

Fairness in scheduling Achieving fairness in scheduling resources in a multi-client environment has been a long-standing topic in computer science [14, 41, 42, 51]. Among these, Fair Queuing [30] has been adapted into many variants for different contexts such as CPU scheduling [3], link bandwidth allocation [11, 15, 17, 23, 38], and memory allocation [33]. Deficit round robin [45] and stochastic fair queuing [27] are non-real-time fair queuing algorithms for variable-size packets, providing guarantees for long-term fairness. There are also real-time fair queuing algorithms (e.g., WFQ [11] and SFQ [17]) that can make more strict short-term delay guarantees [12]. Our scheduling algorithm is different from these algorithms because we need to consider the batching effects across multiple clients' requests and deal with unknown request length. Further, we need to accommodate a flexible notion of fairness on both performance and GPU resource consumption.

Fairness in ML training Within the realm of deep learning, research has delved into scheduling jobs in shared clusters [7, 26, 32, 40], with a primary focus on long-duration training jobs. Machine Learning training jobs have unique characteristics and traditional fair schedulers [18, 22] designed for big-data workflow usually fail [26]. In particular, Themsis [26] points out that ML jobs are device placement sensitive, where jobs will be envious of other's placement even if they are assigned the same number of resources. It then defines a finish-time fairness metric to measure fairness in ML training scenarios. Pollux [40] further points out that ML jobs should jointly consider the throughput and the statistical efficiency, and develop a goodput-based scheduler that further improves the finish-time fairness of ML jobs. In this paper, we consider fairness in LLM serving. The fairness problem in LLM serving is quite different from the fairness problem in model training. In model training, different clients' GPUs are isolated and the problem is which GPUs are assigned to each client. Achieving fairness in LLM serving requires design for a different set of issues, including how to batch requests from multiple clients to achieve high GPU utilization.

LLM Serving Systems How to improve the performance of LLM serving systems has recently gained significant attention. Notable techniques cover advanced batching mechanisms [13, 50], memory optimizations [24, 44], GPU kernel optimizations [2, 9, 34, 48], model parallelism [2, 25, 39], parameter sharing [55], and speculative execution [28, 46] were proposed. FastServe [49] explored preemptive scheduling to minimize job completion time (JCT). However, none of these works consider fairness among clients. Our work bridges this gap, and our proposed scheduling methods can be easily integrated with many of these techniques. Our implementation used for this paper is built atop continuous batching (iteration-level scheduling) [50][10] and PagedAttention [24].

7 Conclusion

We studied the problem of fair serving in Large Language Models (LLMs) with regard to the service received by each client. We identified unique characteristics and challenges associated with fairness in LLM serving, as compared to traditional fairness problems in networking and operating systems. We then defined what constitutes fairness and proposed a fair scheduler, applying the concept of fair sharing to the domain of LLM serving at the token granularity.

Acknowledgment

We thank Aurick Qiao, Shu Liu, Stephanie Wang, Hao Zhang for helpful discussions and feedback. This research was supported by gifts from Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware. Ying is partly supported by the Stanford Center for Automated Reasoning. We thank Clark Barrett for academic advising and funding support.

[10] We presented VTC on continuous batching with separated prefill and decode steps in the main context. A general integration is discussed in Appendix C.1.

References

[1] Linux 2.6.23. Completely fair scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[3] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[4] Jon CR Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on networking*, 5(5):675–689, 1997.

[5] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[6] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual Technical Conference (ATC)*, pages 337–352. USENIX, 2005.

[7] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[8] Lequn Chen. Potentials of multitenancy fine-tuned llm serving. https://le.qun.ch/en/blog/2023/09/11/multi-lora-potentials/, 2023.

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[10] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[11] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In Lawrence H. Landweber, editor, *ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, pages 1–12. ACM, 1989.

[12] Peter L Dorlan. *An introduction to computer networks*. Autoedición, 2016.

[13] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[14] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of Networks and Systems Design and Implementation (NSDI)*, 2011.

[15] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings IEEE INFOCOM '94, The Conference on Computer Communications, Thirteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking for Global Communications*, pages 636–646. IEEE Computer Society, 1994.

[16] P. Goyal, H.M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[17] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 157–168. ACM, 1996.

[18] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[19] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. {Multi-Queue} fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019.

[20] Hugging Face. Text generation inference. https://github.com/huggingface/text-generation-inference. Accessed: 2023-11.

[21] HuggingFace. Text-generation-inference(tgi). https://github.com/huggingface/text-generation-inference, 2023.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 261–276. Association for Computing Machinery, 2009.

[23] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 37–48. ACM, 2004.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[25] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[27] P.E. McKenney. Stochastic fairness queueing. In *Proceedings. IEEE INFOCOM '90: Ninth Annual Joint Conference of the IEEE Computer and Communications Societies@m_The Multiple Facets of Integration*, pages 733–740 vol.2, 1990.

[28] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[29] ModelTC. Lightllm: Python-based llm inference and serving framework. https://github.com/ModelTC/lightllm, 2023. GitHub repository.

[30] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, 1987.

[31] Deepak Narayanan, Keshav Santhanam, Peter Henderson, Rishi Bommasani, Tony Lee, and Percy Liang. Cheaply estimating inference efficiency metrics for autoregressive transformer models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[32] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[33] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006.

[34] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer.

[35] OpenAI. Openai api reference. https://platform.openai.com/docs/api-reference. Accessed: 2023-11.

[36] OpenAI. Gpt-4 turbo, 2023.

[37] OpenAI. Rate limit. https://platform.openai.com/docs/guides/rate-limits?context=tier-free, 2023.

[38] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[39] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[40] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[41] Bozidar Radunovic and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Transactions on networking*, 15(5):1073–1083, 2007.

[42] Uwe Schwiegelshohn and Ramin Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[43] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[44] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[45] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[46] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

[47] Ion Stoica, Hussein M. Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 288–299. IEEE Computer Society, 1996.

[48] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, 2021.

[49] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[51] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

[53] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Conference on Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023.

[54] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

[55] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. Pets: A unified framework for parameter-efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022.

A Missing Proofs in Proving Fairness of VTC

Lemma A.1. *In Algorithm 2,* $\min_{i\in Q}(c_i)$ *is non-decreasing during the time when* $Q \neq \emptyset$.

Proof. We prove the lemma by case study on each line of changing the c_i's.

- In the initialization, all $c_i = 0$, lemma holds.
- If the condition of line 7 is satisfied, at lines 8-14, a new client will be added to Q. If lines 9-10 are reached, the $\min_{i\in Q}(c_i)$ is equals to its value at the last time when $Q \neq \emptyset$. If lines 12-13 are reached, since $c_u = \max\{c_u, \min_{i\in Q} c_i\}$, the $\min_{i\in Q}(c_i)$ will not change.
- At line 24 and line 30, the c_i's can only increase, so that $\min_{i\in Q}(c_i)$ is non-decreasing.
- At line 26, if a client has cleared all its requests from Q, that the client is removed from Q, $\min_{i\in Q}(c_i)$ cannot decrease.

□

Lemma A.2. *The following invariant holds at any time in Algorithm 2 when* $Q \neq \emptyset$*:*

$$\max_{i\in Q} c_i - \min_{i\in Q} c_i \leq \max(w_p \cdot L_{input}, w_q \cdot M) \tag{2}$$

Proof. We prove the lemma by induction. During the induction, for each line of change of c_i in Algorithm 2, we use c_i' to denote the new value and c_i to denote the original value. Similarly, we use Q' to donate the new value and Q to denote the original value. We also use $c_i^{(t)}$ to denote the value of c_i at time t, and $Q^{(t)}$ to denote the value of Q at time t.

1. In the initialization, all $c_i = 0$, Equation (2) holds.

2. If a client $u \notin Q$ receive a new request and thus $Q' = Q \cup \{u\}$, line 12-13 will be reached, and thus $c_u' = \max\{c_u, \min_{i\in Q} c_i\} \geq \min_{i\in Q} c_i$. Then we have,

$$\min_{i\in Q'} c_i' = \min\{c_u', \min_{i\in Q} c_i\} = \min_{i\in Q} c_i. \tag{4}$$

Let t be the last time that u was in Q before the change, and thus $c_u^{(t)} = c_u$. From Equation (2), there is

$$\max_{i\in Q^{(t)}} c_i^{(t)} - \min_{i\in Q^{(t)}} c_i^{(t)} \leq \max(w_p \cdot L_{input}, w_q \cdot M).$$

Then we have

$$c_u^{(t)} \leq \max_{i\in Q^{(t)}} c_i^{(t)} \leq \min_{i\in Q^{(t)}} c_i^{(t)} + \max(w_p \cdot L_{input}, w_q \cdot M).$$

From Theorem A.1, there is $\min_{i\in Q^{(t)}} c_i^{(t)} \leq \min_{i\in Q} c_i$, so we have

$$c_u = c_u^{(t)} \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M),$$

which can derive

$$c_u' = \max\{c_u, \min_{i\in Q} c_i\} \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).$$

Combine with Equation (2) and Equation (4), there is

$$\begin{aligned}\max_{i\in Q'} c_i' &= \max\{c_u', \max_{i\in Q} c_i\} \\ &\leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M) \\ &\leq \min_{i\in Q'} c_i' + \max(w_p \cdot L_{input}, w_q \cdot M)\end{aligned}$$

Therefore, Equation (2) holds after the change.

3. If a client u is left from Q at line 26, the difference $\max_{i\in Q} c_i - \min_{i\in Q} c_i$ will not increase. Because $\max(C') - \min(C') \leq \max(C) - \min(C), \forall C \supseteq C', C' \neq \emptyset$. Therefore, Equation (2) still holds.

4. At line 24, since $c_k = \min_{i\in Q} c_i$, there is

$$\min_{i\in Q} c_i \leq \min_{i\in Q} c_i' \leq c_k' \leq \min_{i\in Q} c_i + w_p \cdot L_{input}. \tag{5}$$

From Equation (2), we have

$$\max_{i\in Q} c_i \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).$$

Because:

$$\max_{i\in Q} c_i' = \max(\max_{i\in Q} c_i, c_k')$$

We have:

$$\max_{i\in Q} c_i' \leq \max(\min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M), c_k') \tag{6}$$

In Equation (5) we have derived that:

$$c_k' \leq \min_{i\in Q} c_i + w_p \cdot L_{input}$$

Thus:

$$\begin{aligned}c_k' &\leq \min_{i\in Q} c_i + w_p \cdot L_{input} \\ &\leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M)\end{aligned}$$

Thus:

$$\max(\min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M), c_k') = \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M).$$

Thus Equation (6) gives:

$$\max_{i\in Q} c_i' \leq \min_{i\in Q} c_i + \max(w_p \cdot L_{input}, w_q \cdot M) \tag{7}$$

Finally, combining the inequality from Equation (5) that

$$\min_{i\in Q} c_i \leq \min_{i\in Q} c_i',$$

we arrive at:

$$\max_{i\in Q} c_i' \leq \min_{i\in Q} c_i' + \max(w_p \cdot L_{input}, w_q \cdot M).$$

Therefore, Equation (2) holds.

5. At line 30, let $k = \arg\max_{i \in Q} c'_i$, so that $c'_k = \max_{i \in Q} c'_i$. Let r be the last one among requests from k that have been scheduled. Let t be the time when r was selected at line 21. Since r is the last one been scheduled from k, there is
$$\max_{i \in Q} c'_i = c'_k \leq c_k^{(t)} + w_q \cdot M \tag{8}$$
Because request r from client k has been scheduled at time t, from line 20, there is $c_k^{(t)} = \min_{i \in Q^{(t)}} c_i^{(t)}$. From Lemma A.1, we have $\min_{i \in Q^{(t)}} c_i^{(t)} \leq \min_{i \in Q} c'_i$. Combine with Equation (8), we have
$$\max_{i \in Q} c'_i - \min_{i \in Q} c'_i \leq w_q \cdot M.$$
Therefore, Equation (2) holds.

□

Theorem 4.8. *For any work-conserving schedule without preemption, there exists some query arrival sequence such that for client f, g and a time period t_1, t_2, such that*
$$|W_f(t_1, t_2) - W_g(t_1, t_2)| \geq w_q \cdot M,$$
where clients f, g are backlogged during the time $[t_1, t_2)$.

Proof. Consider at time 0 the client f sends a list of requests which cannot fit in the memory at once. Because of work-conserving, client f will fill the whole running batch. In this case, client f is backlogged, and any new query is not processed until the existing queries finish processing. Assume that all existing queries finish at time T, and that at time ε with ε close to 0, a second client g sends another batch of requests. Now during the time interval $[\varepsilon, T]$, both clients f, g are backlogged since there exist queries from both clients in the queue. At time T, client f received service from the first batch of processing, which can be up to $w_q \cdot M$ if the memory is luckily fully utilized. Thus we have
$$W_f(\varepsilon, T) = w_q \cdot M.$$
On the other hand, client g did not receive any service during the time period $[\varepsilon, T]$. Thus $W_g(\varepsilon, T) = 0$. In this case, we have constructed an instance with
$$|W_f(t_1, t_2) - W_g(t_1, t_2)| \geq w_q \cdot M.$$

□

Theorem 4.11. *Let $A(r)$ and $D(r)$ denote the arrival time and dispatch time of a request r. Assume there are in total n clients, $\forall t_1, t_2$, if at t_1, a client f is not backlogged and has no requests in the running batch, then the next request r_f with $t_1 < A(r_f) < t_2$ will have its response time bounded:*
$$D(r_f) - A(r_f) \leq 2 \cdot (n-1) \cdot \frac{\max(w_p \cdot L_{input}, w_q \cdot M)}{a} \tag{3}$$
Here a is the lower bound of the capacity in Definition 4.10.

Proof. Let the counter for f be c_f after line 13 for r_f. Before D_{r_f}, since r_f is always in the queue, the counter for f will not be lifted. Since there is no running batch of f in the server, line 21 will select r_f to be the next one for f. Lemma 4.3 shows that for any other client g,
$$c_g - c_f < \max(w_p \cdot L_{input}, w_q \cdot M).$$
In the worst case where these counters are incremented sequentially, it will take at most $2 * (n-1) * \frac{\max(w_p \cdot L_{input}, w_q \cdot M)}{a}$. Thus, giving a bound for the dispatch time of r_f. □

Theorem 4.9. *If a client f is backlogged during time interval $[t_1, t_2)$, for any client g, there is*
$$W_f(t_1, t_2) \geq W_g(t_1, t_2) - 4U.$$
Here U is the upper bound from Equation (2).

Proof. If g is not backlogged during the entire $[t_1, t_2)$, then $W_g(t_1, t_2) \leq U$, the theorem trivially holds. Next, assume g is backlogged at some point during $[t_1, t_2)$. Let t'_1, t'_2 be the first time and the last time g is backlogged between $[t_1, t_2)$. Since there is no request submitted in $[t_1, t'_1)$ and $[t'_2, t_2)$, we have
$$W_g(t_1, t'_1) \leq U, \quad W_g(t'_2, t_2) \leq U. \tag{9}$$
Since c_i's in Algorithm 2 are non-decreasing,
$$c_f^{(t_1)} \leq c_f^{(t'_1)} \leq c_f^{(t'_2)} \leq c_f^{(t_2)}, \tag{10}$$
$$c_g^{(t_1)} \leq c_g^{(t'_1)} \leq c_g^{(t'_2)} \leq c_g^{(t_2)}. \tag{11}$$
According to Lemma 4.3 there is,
$$c_g^{(t'_2)} \leq c_f^{(t'_2)} + U, \quad c_g^{(t'_1)} \geq c_f^{(t'_1)} - U.$$
By Equation (10) and Equation (11):
$$c_g^{(t'_2)} \leq c_f^{(t_2)} + U, \quad c_g^{(t'_1)} \geq c_f^{(t_1)} - U.$$
Since $W_g(t'_1, t'_2) \leq c_g^{(t'_2)} - c_g^{(t'_1)}$, there is
$$W_g(t'_1, t'_2) \leq c_f^{(t_2)} - c_f^{(t_1)} + 2U.$$
Combine with Equation (9), there is:
$$\begin{aligned} W_g(t_1, t_2) &= W_g(t_1, t'_1) + W_g(t'_1, t'_2) + W_g(t'_2, t_2) \\ &\leq c_f^{(t_2)} - c_f^{(t_1)} + 4U. \end{aligned}$$
Since f is backlogged during (t_1, t_2),
$$W_f(t_1, t_2) = c_f^{(t_2)} - c_f^{(t_1)}$$
Thus:
$$W_f(t_1, t_2) \geq W_g(t_1, t_2) - 4U$$

□

Theorem 4.13. *(Fairness for non-overloaded clients) For any time interval $[t_1,t_2)$, we claim the following.*

Assume a client f is not backlogged at time t_1 and for any time interval $[t,t_2), t_1 \le t < t_2$, f has requested services less than $\frac{T(t,t_2)}{n(t,t_2)} - 5U$, where $T(t,t_2)$ is the total services received for all clients during the interval $[t,t_2)$, $n(t,t_2)$ is the number of clients that have requested services during the interval, and U is the upper bound from Equation (2).

Then, all of the services requested from f during the interval $[t_1,t_2)$ will be dispatched.

Proof. We prove by contradiction. Assume there is a request from f that has not been dispatched in t_2, i.e., f is backlogged at t_2. Since f is not backlogged at t_1, there exists a (non-empty) set of time steps such that f becomes backlogged. We let t be the largest element in the set, i.e. f is backlogged at any time in $[t,t_2)$. We claim that $W_f(t,t_2) \ge \frac{T(t,t_2)}{n(t,t_2)} - 4U$.

From the pigeonhole principle, there is at least one client g who has received services $W_g(t,t_2) \ge \frac{T(t,t_2)}{n(t,t_2)}$. If $f = g$, the claim holds. If not, from Theorem 4.9, we have

$$W_f(t,t_2) \ge W_g(t,t_2) - 4U \ge \frac{T(t,t_2)}{n(t,t_2)} - 4U.$$

Since f swicthes from non-backlogged to backlogged at t, requests sent before t at most contributes a U increase in $W_f(t,t_2)$. Thus, requests sent in (t,t_2) at least contribute to $\frac{T(t,t_2)}{n} - 5U$, which contradicts to the assumption in the theorem. □

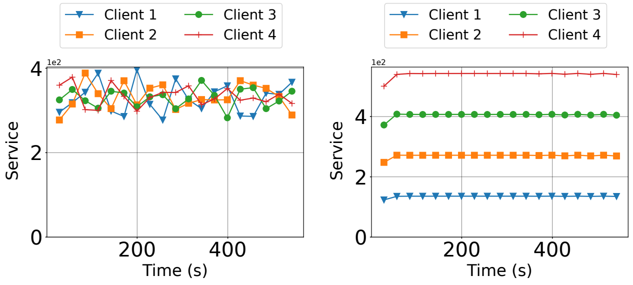
B Advanced VTC Variants

This section presents additional experiments on various variants of VTC. Appendix B.1 evaluate weighted VTC, which is introduced in Section 4.3. In Appendix B.2, we show concretely how VTC can be tailored to specific cost functions, using a profiled service cost function as an example. In Appendix B.3, we include more analysis of VTC with length prediction, which is introduced in Section 4.4. We empirically show its effectiveness in obtaining a better service discrepancy.

For all experiments shown in this section, we run Llama-2-7b on A10G (24GB), using the memory pool of 10000 tokens for KV cache.

B.1 VTC for Weighted Fairness

Figure 16 demonstrates the effectiveness of the weighted VTC in managing clients with varied priority levels. We conducted a test using a synthetic workload involving four overloaded clients. The results depicted in Figure 16a were achieved using standard VTC, which illustrates the comparable levels of service received by all four clients. In contrast, Figure 16b, which was obtained through the application of weighted VTC, shows differentiated service levels. The clients were assigned weights of 1, 2, 3, and 4, respectively, and the resulting service distribution closely adhered to these ratios.



(a) VTC

(b) Weighted VTC

Figure 16: Received service during the 10 minutes of the synthetic overloaded workload with input and output length both at 256. The figure on the left is obtained through standard VTC. The figure on the right is obtained through weighted VTC with weights 1:2:3:4 for the 4 clients.

B.2 VTC with Profiled Cost Function

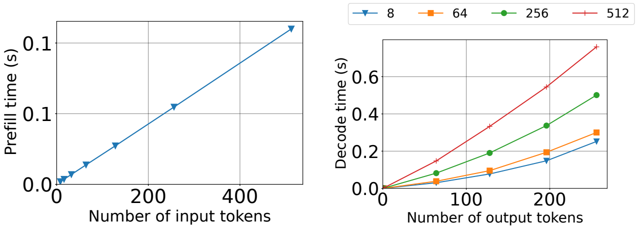
In this section, we demonstrate the generalizability of the token cost function used in VTC (see Section 4.2) by using a profiled service cost function.

To match our experimental setup, we profiled the inference time for Llama-2-7b on an A10G (24GB) across various conditions, as shown in Figure 17. We employed a batch size that utilizes the entire memory pool for each data point corresponding to specific input and output lengths. Consequently, shorter lengths allow for larger batch sizes, while longer lengths necessitate smaller ones. The prefill time is determined by dividing the total prefill time of the batch by the batch size. Similarly, the decode time is calculated by dividing the time taken to decode all tokens in the batch by the batch size. The function $h(n_p,n_q)$ is defined as the sum of prefill and decode times for the data point with input length n_p and output length n_q.

When considering the same total number of input and output tokens, the decode time for scenarios involving all output tokens is about 2 to 5 times the prefill time for scenarios involving all input tokens. The profiled cost function does not follow a linear model. We proceeded to fit the profiled data points and adjusted the coefficients to derive the following cost function:

$$h(n_p,n_q) = 2.1 \cdot n_p + n_q + 0.04 \cdot n_p n_q + 0.032 \cdot n_q^2 + 11.46$$

We conducted real trace experiments using this profiled cost function as the metric, the results of which are presented in Table 3. The disparity between VTC and other baseline methods is insignificant because clients with low request rates,



(a) Prefill Time

(b) Decode Time

Figure 17: Profiled prefill and decode time in different settings. For each data point, the batch size is set to the maximum to fulfill the memory pool (full utilization). The prefill time and decode time are all divided by the batch size. For the figure on the right, the legend for each curve denotes its number of input tokens.

when starved, do not substantially impact the overall service difference. However, as observed in Figure 18, VTC successfully maintains low response times for clients with low request rates, a feat not matched by other baselines except for LCF. In the case of LCF, clients with consistently high request rates face undue penalties, resulting in excessively high response times. We reinforce our findings by assessing the profiled cost function on a synthetically overloaded workload to highlight the differences between VTC and FCFS, as shown in Table 4.

The results empirically show that VTC can achieve fairness using a customized cost function. However, our goal is not to determine the optimal cost function or pricing model, as these can vary based on numerous factors in a production environment and may change over time. The investigation into the cost function and pricing model is designated for future research.

Scheduler	Max Diff	Avg Diff	Diff Var	Throu	Isolation
FCFS	743.23	457.29	26645.42	777	No
LCF	709.35	384.78	23299.20	778	Some
VTC	707.35	368.74	21918.67	780	Yes
VTC(predict)	617.22	337.05	11803.41	778	Yes
VTC(oracle)	**387.43**	**277.18**	**4541.57**	**783**	**Yes**
RPM(5)	230.78	151.00	823.15	340	Some
RPM(20)	445.34	270.51	5938.52	694	Some
RPM(30)	801.16	377.22	25980.39	747	Some

Table 3: Results run on real workload under the profiled cost function introduced in Appendix B.2. The service difference is counted by summing the service difference between each client and the client who received the maximum services. Throughput is the total number of tokens (including input and output tokens) processed divided by the total execution time.

Scheduler	Max Diff	Avg Diff	Diff Var	Throughput
FCFS	323.18	317.13	15.98	876
VTC	137.27	74.87	2819.40	900
VTC(oracle)	4.28	0.34	0.91	893

Table 4: Results run on the synthetic overloaded workload with 2 clients under the profiled cost function introduced in Appendix B.2. The work difference is counted by summing the work difference between each client and the client who received the maximum services.

B.3 VTC with Length Prediction

The adapted pseudocode for VTC with length prediction is detailed in Algorithm 3. In line 25, the cost associated with the predicted number of output tokens is preemptively calculated. Lines 32-37 describe the adjustments made to the cost to correspond with the actual number of output tokens produced.

Figure 19 demonstrates how length prediction reduces service discrepancies among clients in a synthetic workload scenario where all clients are overloaded. "VTC (oracle)" refers to a simulation using a predictor with 100% accuracy. "VTC (±50%)" simulates a predictor that randomly selects a value within 50% of the actual output length, either above or below. While standard VTC ensures that the absolute differences in services received by clients remain bounded and do not grow over time, VTC with length prediction significantly lowers these differences throughout the test period, even with a prediction error margin of 50%. Table 5 and Table 6 provide quantitative assessments of the service discrepancies among overloaded clients under the same conditions.

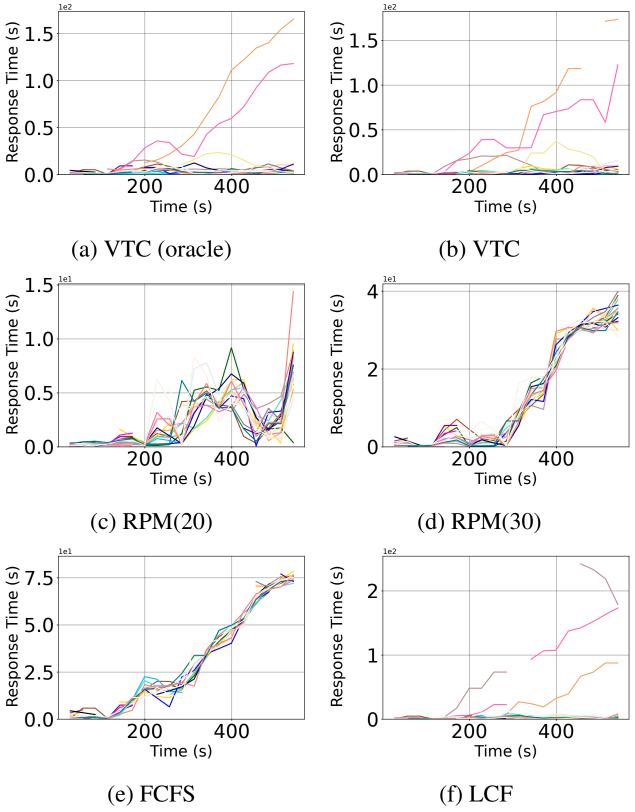
Scheduler	Max Diff	Avg Diff	Diff Var	Throughput
VTC	192.88	103.77	6981.24	893
VTC (±50%)	33.98	12.54	111.94	904
VTC (oracle)	5.87	0.51	1.71	895

Table 5: Results run on 10-minute synthetic workload same with Figure 19 for 2 clients. The service difference is counted by summing the work difference between each client and the client who received the maximum services. Throughput is the total number of tokens (including input and output tokens) processed divided by the total execution time.

C Discussions

C.1 VTC Integration in Real Systems

In Algorithm 2, we have shown an example of VTC integration with continuous batching. In implementation, VTC integration should be a simple change in the request scheduler. Generally, for an existing serving system, there are three mod-



(a) VTC (oracle)

(b) VTC

(c) RPM(20)

(d) RPM(30)

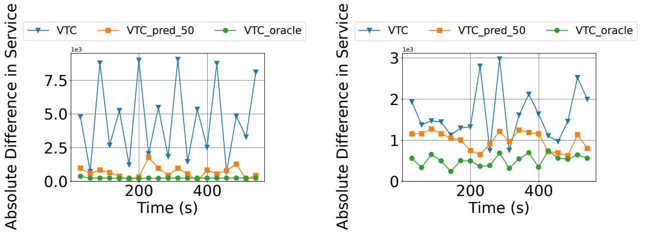
(e) FCFS

(f) LCF

Figure 18: Response time of the 27 clients during the 10 minutes of real trace simulation using different schedulers. The VTC style schedulers are using the profiled cost function introduced in Appendix B.2. There are some curves that show disconnected because, during some periods, a client may have no requests served. Requests distribution see Figure 11.

ules that need to be modified. First, the monitoring stream handles counter-lifting when a new request comes, as shown in Algorithm 4, which is the same as in Algorithm 2. Second, when new tokens have been processed, the counters should be updated according to a pre-defined cost function as discussed in Section 4.2. Third, when new requests need to be selected for processing, we schedule the request from a user with the lowest counter first. The added modules are demonstrated in Algorithm 4. We are assuming a customized cost function $h(n_p, n_q)$ as introduced in Section 3.1. At line 22, n_p^r, n_q^r denote the number of processed input and output tokens, and $n_p^{r(old)}, n_q^{r(old)}$ denote the number of processed input and output tokens before processing the new tokens.

Those modules for maintaining the virtual token counters and selecting requests according to the counters could be additive features of an existing serving system. However, in some cases, VTC is possibly in conflict with a scheduling algorithm that optimizes performance while being against fairness. Cache-aware scheduling introduced in [54] is an example in which requests with shared prefixes will always be prioritized. A natural solution to combine the two is adding a policy of switching between the two schedulers by setting tolerable fairness bounds. We leave such exploration as future research.



(a) 2 Clients

(b) 8 Clients

Figure 19: The figures illustrate the maximum difference in accumulated service received by clients during a 10-minute period of synthetic workload, where both the input and output lengths are set at 256. The left figure is derived from a simulation involving two clients, while the right figure comes from a simulation involving eight clients. In both scenarios, the request rate for each client surpasses the available capacity, resulting in continuous backlogging of each client.

Scheduler	Max Diff	Avg Diff	Diff Var	Throughput
VTC	322.16	162.20	5151.49	875
VTC (± 50%)	99.43	66.32	487.10	875
VTC (oracle)	43.23	36.34	56.52	875

Table 6: Results run on 10-minute synthetic workload same with Figure 19 for 8 clients. The service difference is counted by summing the work difference between each client and the client who received the maximum services. Throughput is the total number of tokens (including input and output tokens) processed divided by the total execution time.

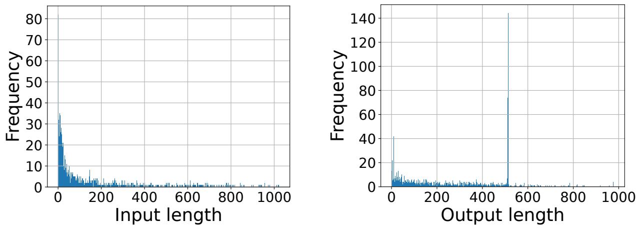


Figure 20: Request input and output length distribution in the real workload trace during the sampled 10 minutes duration with re-scale. The average input length is 136, and the average output length is 256. The input and output lengths have the range of $[2, 1021]$ and $[2, 977]$, respectively.

C.2 Adapted Deficit Round Robin

We have briefly discussed in Section 2.3 why Deficit Round Robin (DRR) cannot be directly applied. In this section, we

Algorithm 3 VTC with Length Prediction

```
Input: request trace, input token weight $w_p$, output token
    weight $w_q$, upper bound from Equation (2) denoted as $U$.
let current batch $B \leftarrow \emptyset$
let $c_i \leftarrow 0$ for all client $i$
let $Q$ denote the waiting queue, which is dynamically
changing.
▷ with monitoring stream:
while True do
    if new request $r$ from client $u$ arrived then
        if not $\exists r' \in Q, client(r') = u$ then
            if $Q = \emptyset$ then
                let $l \leftarrow$ the last client left $Q$
                $c_u \leftarrow \max\{c_u, c_l\}$
            else
                $P \leftarrow \{i \mid \exists r' \in Q, client(r') = i\}$
                $c_u \leftarrow \max\{c_u, \min\{c_i \mid i \in P\}\}$
        $Q \leftarrow Q + r$
▷ with execution stream:
while True do
    if can_add_new_request() then
        $B_{new} \leftarrow \emptyset$
        while True do
            let $k \leftarrow \arg\min_{i \in \{client(r) | r \in Q\}} c_i$
            let $r$ be the earliest request in $Q$ from $k$.
            if $r$ cannot fit in the memory then
                Break
            $c_k \leftarrow c_k + w_p \cdot input\_length(r)$
            $c_k \leftarrow c_k + w_q \cdot predicted\_output\_length(r)$
            $B_{new} \leftarrow B_{new} + r$
            $Q \leftarrow Q - r$
        forward_prefill($B_{new}$)
        $B \leftarrow B + B_{new}$
    forward_decode($B$)
    ▷ Adjust the cost of output tokens
    for each $r \in B$ do
        $\delta \leftarrow output\_len(r) - predicted\_output\_len(r)$
        if $\delta > 0$ then
            $c_{client(r)} \leftarrow c_{client(r)} + w_q$
        if $r$ is finished and $\delta < 0$ then
            $c_{client(r)} \leftarrow c_{client(r)} + w_q \cdot \delta$
    $B \leftarrow$ filter_finished_requests($B$)
```

Algorithm 4 General VTC

```
Input: request trace, input token weight $w_p$, output token
    weight $w_q$, upper bound from Equation (2) denoted as $U$.
let current batch $B \leftarrow \emptyset$
let $c_i \leftarrow 0$ for all client $i$
let $Q$ denote the waiting queue, which is dynamically
changing.
▷ with monitoring stream:
while True do
    if new request $r$ from client $u$ arrived then
        if not $\exists r' \in Q, client(r') = u$ then
            if $Q = \emptyset$ then
                let $l \leftarrow$ the last client left $Q$
                $c_u \leftarrow \max\{c_u, c_l\}$
            else
                $P \leftarrow \{i \mid \exists r' \in Q, client(r') = i\}$
                $c_u \leftarrow \max\{c_u, \min\{c_i \mid i \in P\}\}$
        $Q \leftarrow Q + r$
▷ when process new request:
if add_new_request() then
    let $k \leftarrow \arg\min_{i \in \{client(r) | r \in Q\}} c_i$
    let $r$ be the earliest request in $Q$ from $k$.
    $Q \leftarrow Q - r$
    original process when selecting $r$.
▷ when new tokens been processed:
$c_i \leftarrow c_i + \sum_{r|client(r)=i} \left( h(n_p^r, n_q^r) - h(n_p^{r(old)}, n_q^{r(old)}) \right)$
```

discuss an adaptation of Deficit Round Robin [45] and show it is equivalent to our proposed VTC scheduler.

The original DRR can be described as follows:

1. The algorithm maintains a constant Q, which is the quantum that each client has.
2. Every client maintains a variable C_i that represents its deficit, which is initialized as 0.
3. On each round, the algorithm visits each client with a non-empty queue and schedules its requests as many as possible if the incurred cost P is less than or equal to $Q + C_i$. The C_i is then updated to $Q + C_i - P$ if $P > C_i$ or $C_i - P$ if else.

The obstacle to applying DRR in LLM serving is that we do not know how many requests we should schedule to meet the requirement of $P \leq Q + C_i$ since the number of output tokens is unknown in advance.

We then give an adapted version for LLM serving:

1. The algorithm maintains a constant Q, which is still the quantum that each client has.
2. Every client maintains a variable C_i that represents its debt, which is initialized as 0.
3. In each round, the algorithm processes each client. If $C_i \leq 0$, it refills C_i by adding Q to it. Should the updated C_i become positive, the algorithm schedules as many requests as possible, such that the cost associated with the prompt tokens P slightly exceeds C_i with the addition of the last scheduled request. After scheduling, P is subtracted from C_i.
4. Each time a new token is decoded, the associated cost is deducted from the respective C_i. Consequently, C_i may become negative, exceeding the value of Q multiple times, and it might require waiting through several

rounds before it can be scheduled again.

Fairness is no longer strictly bounded by Q, yet a smaller Q promotes a tighter constraint. When $Q = \varepsilon$ is extremely small, smaller than the cost of a single prompt token, the algorithms revert to functioning like the VTC algorithm. This is because each round results in one of two outcomes: either all C_i values remain non-positive, prompting another round, or the highest C_i turns positive and the corresponding client is scheduled. The client with the highest C_i is the one who has received the least service, which corresponds to having the smallest virtual counter in VTC.

If a client has no requests in the queue at a given time, it will cease to be refilled once $C_i \geq 0$. When a new request arrives, its C_i will be within $(0, \varepsilon]$, approximating $\max_i C_i$. The $\max_i C_i$ remains within the range of $(0, \varepsilon]$ because the algorithm persistently adds ε to C_i to maintain it positive, but then rapidly pulls it back into the negative by scheduling new requests. This process mirrors the counter lift mechanism in VTC.

In addition to its similarity to VTC, practically, simulating repeated round-robin with a small quantum Q is inefficient. Therefore, we focus solely on analyzing VTC in this paper, leaving the discussion of the round-robin simulation here for reference.

C.3 Future Work

Preemption As we mentioned in Section 2.1, this paper focuses on how to integrate fair scheduling with continuous batching, and leaving an investigation on preemption as an orthogonal future work. But we still would like to discuss how preemption will affect the VTC algorithm, and point out a possible future research on it.

The nature of unpredictable length in a no-preemption framework directly affects the fairness bound in the main theorem Theorem 4.4, which is $U = 2\max(w_p \cdot L_{input}, w_q \cdot M)$. Intuitively, the worst case occurs when many requests from one client are added, generating a large number of tokens that cannot be preempted. During the process, other clients cannot catch up arbitrarily because the memory is occupied. Essentially, this is caused by an underestimation of a future number of tokens, similarly explained in the ablation study (Section 5.4) and VTC with length prediction (Section 4.4).

In Theorem 4.7, we mentioned that we could restrict the memory usage for each client in the running batch to obtain a better bound. However, this can potentially lower the overall throughput because the memory may not always be fully utilized. Having a preemption mechanism could be a good solution to address the problem of underestimating and tightening the bound. Basically, if the difference in service is larger than a threshold, we can preempt the requests in processing and swap in requests from clients with lower counters.

VTC for distributed systems Integrating VTC in a distributed LLM serving system is an interesting direction for future work. For a distributed setup where there are many replicas of serving engines, we will have a central request dispatcher where we can keep the token counter and enforce the algorithm (this is similar to hierarchical fair sharing [4] in the network domain, and multi-queue fair queuing [19]). The bound now is dependent on the total memory capacity of all the serving engines. However, in the distributed setting, the counter will be updated by different serving engines concurrently, raising the problem of counter synchronization, which will be interesting to explore as a future work.

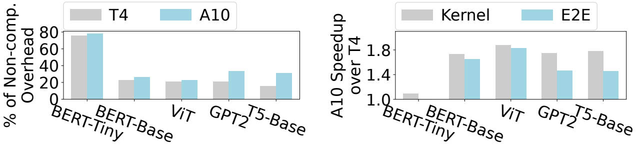
VTC and Auto-scaling The VTC algorithm does not rely on a constant capacity. Adding and removing GPUs will not affect the algorithm but may need a hierarchical virtual counter as discussed in the paragraph about distributed systems. However, auto-scaling is a possible approach to mitigate the issue of throughput degradation in RPM. The resources can be auto-scaled to fit the fluctuating traffic, but this requires flexible and responsive resource management. Auto-scaling has its own challenges, including operational cost overhead, inaccurate workload prediction, and delays. A combination of VTC and auto-scaling is a future direction worth exploring.

MonoNN: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern GPU-Centric Architectures

Donglin Zhuang [†*◇], Zhen Zheng [‡*], Haojun Xia [†◇], Xiafei Qiu [‡], Junjie Bai [‡], Wei Lin [‡]
Shuaiwen Leon Song [†]
[†]*The University of Sydney* [‡]*Alibaba Group*

Abstract

In this work, we reveal that the kernel-by-kernel execution scheme in the existing machine learning optimizing compilers is no longer effective in fully utilizing hardware resources provided by the advances of modern GPU architectures. Specifically, such scheme suffers from severe non-computation overhead and off-chip memory traffic, making the optimization efforts from the state-of-the-art compiler techniques greatly attenuated on the newer generations of GPUs. To address this emerging challenge, we propose *MonoNN*, the first machine learning optimizing compiler that enables a new monolithic design and optimization space for common static neural network (NN) inference tasks on a single GPU. MonoNN can accommodate an entire neural network into a single GPU kernel, drastically reducing non-computation overhead and providing further fine-grained optimization opportunities from the newly formed monolithic optimization space. Most importantly, MonoNN identifies the resource incompatibility issue between various NN operators as the key design bottleneck for creating such a monolithic optimization space. Then MonoNN effectively tackles it by systematically exploring and exploiting the parallelism compensation strategy and resource trade-offs across different types of NN computations, and by proposing a novel schedule-independent group tuning technique to significantly shrink the extremely large tuning space. Finally, MonoNN provides a compiler implementation that incorporates our proposed optimizations and automatically generates highly efficient kernel code. Extensive evaluation on a set of popular production inference tasks demonstrates that MonoNN achieves an average speedup of 2.01× over the state-of-the-art frameworks and compilers. Specifically, MonoNN outperforms TVM, TensorRT, XLA, and AStitch by up to 7.3×, 5.9×, 1.7× *and* 2.9× in terms of end-to-end inference performance, respectively. MonoNN source code is publicly available at `https://github.com/AlibabaResearch/mononn`.



(a) Percentage of non-computation overhead on two generations of inference GPUs.

(b) GPU kernels-only speedup vs end-to-end (E2E) speedup by shifting hardware from T4 to A10.

Figure 1: Low hardware utilization for inference caused by growing non-computation overhead.

1 Introduction

In recent years, machine learning (ML) inference tasks have become one of real-world systems' most fundamental computation types. Existing optimization approaches [2, 7, 18, 24, 39, 41, 42] transform an ML computational graph into hundreds or thousands of computation kernels, and offload them onto high-performance AI accelerators, e.g., GPUs, for drastic latency reduction. However, with the increasing hardware advances of these complex GPUs on computation capability, the traditional kernel-by-kernel execution scheme is no longer effective in fully utilizing hardware resources.

Take XLA [2] as an example, which is one of the most popular and effective optimizers for ML workloads, Fig.1a shows the non-computation overheads (i.e., the end-to-end inference latency minus the pure kernel execution time on GPU) of five popular models on two generations of NVIDIA GPUs. Typically, the non-computation overhead mainly originates from frequent context switches between the host and GPU, e.g., framework scheduling and kernel launching. With the significant increase in computing power from T4 to A10, although the NN operators are executed faster, the non-computation

[◇] Work was done when interned at Alibaba.

[*] Equal contribution.

overheads tend to dominate the end-to-end performance, As illustrated in Fig.1b, the achieved end-to-end performance speedup can be far less than the kernel execution speedup when shifting across generations of hardware.

Moreover, the recent increase in computing power remains faster than that of memory bandwidth in recent generations of GPUs [42], making off-chip memory traffic among different GPU kernels within a model a significant performance bottleneck. Furthermore, there is a common scenario that often occurs in real-world systems and exacerbates the situation: CPUs are usually busy with data pre-/post-processing for real-time ML tasks, causing further delays in scheduling and launching their GPU kernels and subsequently increasing the non-computation overhead. To the best of our knowledge, although the kernel fusion scope and the corresponding techniques might be different, all the existing ML compilers [7, 18, 24, 39, 41, 42] suffer from performance issues discussed above due to the fundamental kernel-by-kernel execution scheme. Therefore, there is an urgent demand for a general solution with minimal non-computation overhead that can be widely applied to common ML inference tasks.

In this paper, we make a key observation that there exists *a monolithic design and optimization space* accommodating a wide spectrum of prevalent static DNN models in single GPU inference (Sec.3). MonoNN keeps the computation flow of the entire neural network on the GPU side without going back to the host to seek scheduling control. Such a scheme effectively avoids the non-computation overhead caused by the CPU-GPU context switch. With the structure of modern static DNNs consisting of repetitive layers, it would be more justified to aggressively enlarge the fusion scope, even result in a single kernel[1].

However, it is non-trivial to provide a general optimization scheme to consolidate all types of computations of an entire neural network into a *monolithic kernel*, while guaranteeing high performance and providing further fine-grained optimization opportunities from the newly formed monolithic optimization space. We observe that the main difficulty in forming such an optimization space is resource incompatibility between different types of neural network computations. On the one hand, a resource configuration that favors some operators can lead to a dramatic drop in performance on some other operators (e.g., low thread-level parallelism for GEMM computation is inefficient for element-wise operators). On the other hand, the resource configuration (e.g., parallelism configuration, register, and shared memory allocation) is fixed during the lifetime of a GPU kernel. Failure in reconciling such resource incompatibility in a monolithic kernel will result in poor performance. Furthermore, accommodating all operators of a complete NN into one GPU kernel results in an extremely large optimization space, making it very difficult for performance tuning on the whole computation graph.

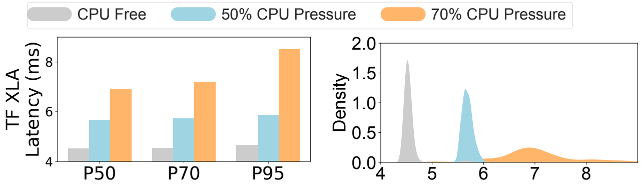
To address these emerging problems, we propose MonoNN, an ML optimizing compiler that enables a new monolithic optimization space for common NN inference tasks on modern GPU-centric architectures. Specifically, to address the significant resource incompatibility issue and accommodate the different resource requirements from various operators, we propose a *context-aware instruction rescheduling* technique (Sec.4.2.2). The key insight is to exploit the hidden instruction-level parallelism (ILP) for memory-intensive computations (e.g., element-wise, reduction) to compensate for the loss of the thread-level parallelism (TLP) under the monolithic kernel context. To further accelerate memory access, MonoNN classifies the memory access patterns into *streaming* and *temporal*, and comprehensively exploits all types of on-chip memory resources for the access patterns accordingly (Sec.4.3). It further exploits whole-graph level transformation inside the kernel to rearrange independent subgraphs together to reduce global thread barrier overhead (Sec.4.4). Finally, we systematically abstract the optimization space of the monolithic kernel and propose a *schedule-independent group tuning* approach to drastically compress the tuning space (Sec.5). Extensive evaluation on a set of neural network inference tasks demonstrates that MonoNN outperforms the state-of-the-art optimizers with an average of 2.01× speedup. Specifically, MonoNN outperforms TVM, TensorRT, XLA, and AStitch by up to 7.3×*,* 5.9×*,* 1.7× *and* 2.9× in end-to-end inference performance. To summarize, this work makes the following contributions:

- To the best of our knowledge, MonoNN is the first ML optimizing compiler that discovers a new monolithic optimization space for common static DNNs' inference scenarios that are served on a single GPU, and provides automatic high-performance kernel generation. This is also the first study that explores and evaluates this monolithic optimization design space and its limitations so that the community has a better understanding of the tradeoffs;
- It is the first optimizing compiler that explicitly exploits instruction-level parallelism optimization for memory-intensive operators to compensate for thread-level parallelism loss, enabling a new optimization dimension for neural network inference optimization;
- MonoNN enables a sophisticated compression mechanism to significantly shrink the tuning space for our proposed monolithic NN kernel;
- Extensive evaluation results have demonstrated the effectiveness of MonoNN on both single inference tasks as well as multi-inference processing scenarios.

2 Background and Motivation

Emerging Challenges in Optimizing NN Inference. From an optimization perspective, operators in neural network (NN) models can be classified into two categories, compute-intensive operators and memory-intensive opera-

[1] We also present a study on the fusion granularity under the monolithic optimization space in Sec.7.3



(a) Tail latency. (b) Latency distribution (ms).

Figure 2: T5 model latency statistics under each CPU pressure. The input sequence length is 128. (a): P50/P70/P90 tail latency. (b): latency distribution.

tors. Compute-intensive operators are typically composed of heavy arithmetic computations (e.g., GEMM and Conv), while memory-intensive operators are typically bounded by memory bandwidth (e.g., element-wise and reduction operations). Note that previous studies [18, 41, 43] have concluded both types of operators can dominate the execution time.

With the rapid growth of computing power for recent GPU generations[2], the execution time of compute-intensive operators decreases drastically. For example, Tensor Core brings an order of magnitude improvement in arithmetic unit throughput for compute-intensive operators since NVIDIA Volta architecture [3] (similarly, Matrix Core was also introduced in AMD GPUs since CDNA architecture [8]). However, there exists a disproportionate performance gain between hardware throughput improvement and end-to-end inference speedup. For instance, for the two common inference GPUs, NVIDIA A10 GPU has 1.9× more half-precision floating point throughput than its predecessor NVIDIA T4, while we only observe a 1.6× end-to-end inference speedup for the BERT model[3] [19] with XLA compiler optimization enabled [2]. Furthermore, we identified two emerging fundamental difficulties in optimizing inference scenarios on increasingly advanced modern GPUs:

(i) Continuous advances in computation throughput leads to an increasing portion of non-computation overhead.

Faster GPUs can offer shorter per-kernel execution in NN inference. However, the major portion of performance gains from hardware speed improvements for regular-size models begins to diminish as non-computation overhead becomes a notable portion of end-to-end latency. This new bottleneck mainly originates from frequent non-computation overhead which includes (1) context switch between host and GPU accelerator due to framework scheduling and kernel launch, and (2) off-chip memory traffic between operators.

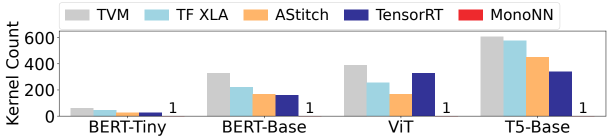


Figure 3: Number of inference GPU kernels for existing frameworks and MonoNN (1).

As the breakdown of the overall context switch overhead in Fig.1a, our measurements indicate that the framework scheduling accounts for 38.3% while the kernel launching overhead accounts for around 61.7% (see Sec.7.2.5 for more details). As for off-chip memory traffic, the memory bandwidth growth across hardware generations is generally slower than that for arithmetic throughput. Fig.1a demonstrates the non-computation inference overhead via XLA optimizations for five common models. It is worth noting that A10 suffers from more severe non-computation inference overhead than its predecessor T4 as newer generations of GPUs have much shorter per-kernel duration. Fig.1b illustrates that there is an average of 1.64× kernel execution speedup benefiting from shifting the underlying accelerator from T4 to A10. Unfortunately, such speedup decreases to 1.48× for the end-to-end latency as the non-computation overhead is not the highest optimization priority for the existing inference engines. Thus, the non-computation inference overhead for neural network models is becoming increasingly essential for the next generations of faster GPU hardware [6].

(ii) Ever-present, non-negligible CPU workloads exacerbate non-computation overhead.

Moreover, a commonly neglected factor is that CPU is usually busy with pre- and post-processing of input and output for NN tasks in real-world execution. Thus, CPU contention often further delays the scheduling and kernel launching of a large number of GPU kernels within a model execution. This further exacerbates the CPU-GPU context switch overhead and makes it a much more severe problem, causing an additional slowdown of model inference tail latency. In Fig.2a, when measuring the tail latency under XLA optimizations on a server with a 64-core CPU (128 threads) and an NVIDIA A10 GPU under 50% (70%) CPU utilization, the tail latency increases by 25% (52%), 26% (58%), and 26% (82%) at P50, P70, and P95, respectively, over the latency of an idle CPU. Fig.2b shows a detailed inference latency distribution of 1000 times of inference when CPU is under various utilization. With the increasing CPU contention, the end-to-end inference latency belongs to a wider range of much slower outliers. Note that it is impractical to designate a specific CPU core exclusively just for kernel launching in the datacenter because the CPUs are typically very busy performing pre-/post-processing. Besides, designating such a core requires a hardcoded list of CPU cores to be isolated from the default CPU scheduler in the system boot phase, resulting in rebooting for every new

[2] In this work, we focus our discussion on the most widely-adopted general-purpose AI accelerators: GPUs. Although the technical terminologies used in this paper are adopted from NVIDIA GPUs [4, 5], our proposed techniques aim to serve as general principles that are valuable for modern general-purpose machine learning system designs, and are applicable to other SIMT accelerators [8].

[3] Data is collected under TensorFlow XLA v2.7 with Tensor Core enabled, using 1 as the batch size and 128 as the sequence length.

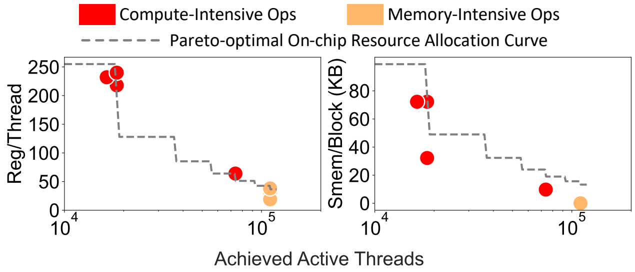


Figure 4: Different resource requirements between compute-intensive operators and memory-intensive operators for TensorRT BERT inference. Each data point may represent multiple GPU kernels with similar resource usage within a model.

inference service deployment.

Challenges in the State-of-The-Art Designs. TVM [18] applies a basic fusion strategy but still unnecessarily launches a large number of kernels. Some recent works propose more advanced fusion techniques to alleviate the problems above. AStitch [41, 42] leverages hierarchical GPU memory to fuse multiple memory-intensive operators with complex data dependencies into a single GPU kernel, named stitch optimization. TensorRT [11] also exploits a similar strategy since v8.

Although it helps reduce the kernel number to some extent, it still results in a large number of kernels since it is not capable to fuse globally along with all the compute-intensive operators, for which the bottlenecks that we discussed above still exist. As illustrated in Fig.3, TVM, XLA, TensorRT, and AStitch are all launching a large number of kernels during model inference. Furthermore, Rammer [24] partially addresses this problem with a persistent-thread technique [14, 17] to generate the schedule of multiple operators within one kernel. However, Rammer is incapable of handling the resource incompatibility between different operators in an entire neural network (see Sec.3.1). As a result, Rammer has to partition the neural network into separate GPU kernels for NN inference. For example, Rammer still launches 734 kernels on GPU for BERT-Large [19] model inference.

3 Monolithic Optimization Space

To address the emerging challenges discussed in Sec.2, we explore the *monolithic optimization space* where the entire neural network can be compiled into a single GPU kernel. This approach is appealing because it only incurs minimal non-computation overhead and enables the opportunities for whole graph optimization within the same kernel space. However, a general approach enabling this optimization space is non-trivial, especially when handling various NN models with very different execution patterns. Here we summarize two major challenges to auto-generate a highly-efficient GPU kernel containing all the operators of a given neural network.

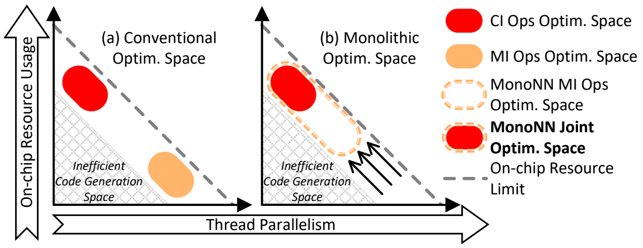

On-chip Resource Usage
(a) Conventional Optim. Space
Inefficient Code Generation Space
(b) Monolithic Optim. Space
Inefficient Code Generation Space
Thread Parallelism
CI Ops Optim. Space
MI Ops Optim. Space
MonoNN MI Ops Optim. Space
MonoNN Joint Optim. Space
On-chip Resource Limit

Figure 5: Optimization space comparison.

3.1 Main Challenges of Enabling A Monolithic Kernel Optimization Space

Challenge 1: Resource incompatibility between compute-intensive and memory-intensive operators. The resource incompatibility between compute-intensive and memory-intensive operators hinders the state-of-the-art techniques to consolidate all operators into a monolithic kernel. Compute-intensive operators usually require a large amount of on-chip resources (e.g., registers and shared memory) whereas memory-intensive ops rely on massive concurrent threads to hide off-chip memory access. Thus, *it is extremely difficult to accommodate all types of operators by creating a GPU kernel with both high on-chip usage and massive concurrent threads due to the resource constraints on modern GPUs*. For example, the active TLP on an SM core will inevitably drop when a kernel uses a large number of registers and shared memory due to the limited on-chip resources. We illustrate this phenomenon quantitatively using GPU kernels from a TensorRT optimized BERT [19] and the Pareto-optimal on-chip resource allocation curve on an NVIDIA A10 GPU in Fig.4. Compute-intensive kernels in NN models tend to be closer to the upper-left corner, representing high on-chip resource allocation and relatively low achieved concurrently active threads. In contrast, memory-intensive kernels tend to be closer to the bottom-right corner, representing low on-chip resource allocation and massive concurrently active threads (or high TLP). All data points in Fig.4 are subject to resource constraints and thus will not be above the Pareto-optimal curve.

Challenge 2: Extremely high implementation cost and huge tuning space. Modern ML models usually consist of thousands of operators with diverse computation patterns, resulting in intricate data dependencies. Manual implementation and optimization are no longer viable for developing a monolithic kernel. In terms of compiler optimization, the monolithic kernel approach significantly expands the optimization search space as all the operators coexist within the same kernel. Consequently, it becomes exceedingly challenging to identify suitable configurations and implementations for each operator to achieve optimal end-to-end inference efficiency within a monolithic kernel collectively.

3.2 A High-level Glance of MonoNN

The ultimate objective of MonoNN is to create an efficient joint optimization space for both compute-intensive operators (*CI Ops*) and memory-intensive operators (*MI Ops*). However, as elaborated in Fig.4 and Sec.3.1, these two types of operators naturally reside in disjoint optimization space in conventional solutions due to resource incompatibility. We conceptually illustrate this observation in Fig.5(a).

MonoNN enables a new monolithic optimization space that can effectively accommodate both *CI Ops* and *MI Ops*. The key idea is to align the optimization space of *MI Ops* as closely as possible with that of *CI Ops* (Fig.5(b)). Specifically, MonoNN leverages the hidden instruction-level parallelism (ILP) to offset the reduction in TLP for memory-intensive subgraphs, thereby achieving a similar resource allocation to *CI Ops* (Sec.4.2.2). Furthermore, MonoNN strategically utilizes abundant on-chip resources, including registers, shared memory, and cache, to buffer and prefetch off-chip data based on an analysis of memory access patterns (Sec.4.3). This approach enables both *CI Ops* and *MI Ops* to coexist within the same monolithic kernel efficiently. Additionally, MonoNN explores global optimization opportunities to minimize global synchronizations between computations, further enhancing the efficiency of neural network models (Sec.4.4).

4 System Design

4.1 Overview of MonoNN

Fig.6 illustrates the overview of MonoNN. MonoNN first formulates the input neural network into different subgraphs for subsequent optimizations (Fig.6(1), Sec.4.2.1). Then, MonoNN enables the hidden parallelism of memory-intensive subgraphs through context-aware instruction rescheduling (Fig.6(2), Sec.4.2.2), and comprehensively optimizes the usage of various on-chip resources according to memory access patterns (Fig.6(3), Sec.4.3). Next, it reorders and clusters subgraphs to reduce the required Global Thread Barriers (*GTBs*) to minimize the synchronization overhead (Fig.6(4), Sec.4.4). Finally, MonoNN abstracts, compresses, and tunes for the large monolithic optimization space with *schedule-independent group tuning*, and compiles the monolithic kernel into an executable binary (Fig.6(5)-(6), Sec.5).

4.2 Exploiting Hidden Parallelism for Memory-intensive Subgraphs

We address the resource incompatibility issue discussed in Sec.3 for memory-intensive computations with *context-aware instruction rescheduling* (the compute-intensive computations will be discussed in Sec.4.5.) MonoNN performs instruction rescheduling under the context of *monolithic optimization space* to recover potential TLP loss for memory-intensive computations with high-level instruction-level parallelism (ILP) enhancement. Thereby, MonoNN fully leverages the abundant registers under the new monolithic optimization space to unleash the hidden potential of reaching high performance.

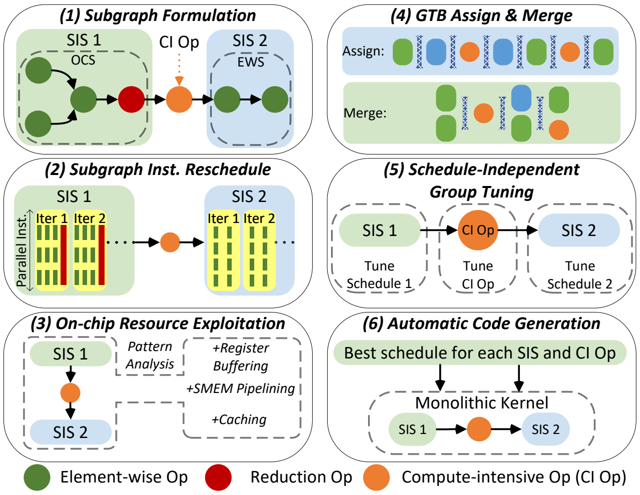


Figure 6: MonoNN overview. *SIS*: Schedule-independent subgraph. *EWS*: Element-wise subgraph. *OCS*: Output-only contracted subgraph.

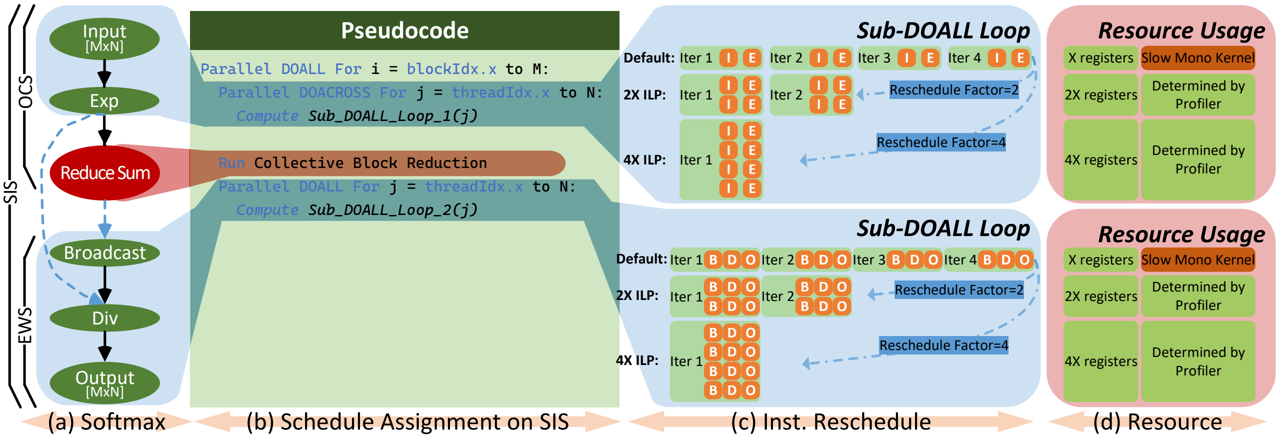
4.2.1 Memory-intensive Subgraph Formulation

Before unveiling the details of *context-aware instruction rescheduling*, we first present how subgraphs are formulated as the basic units of optimization exploration. MonoNN converts the whole graph of a model into one kernel. Instead of optimizing and generating the code of the whole graph all in one shot, MonoNN generates the schedules[4] of different partitions (i.e., subgraphs) of the graph separately under the same monolithic context, and then stitches them together with shared memory or global memory data buffering.

Subgraph Formulation. The compute-intensive operators divide the whole computation graph into a set of memory-intensive subgraphs. We use the following criterion to further categorize memory-intensive subgraphs based on data dependencies between data elements in input and output tensors.

Formally, for a subgraph with m input tensors $[\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}]$ and n output tensors $[\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}]$, the computation of the subgraph is: $[\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}] = f([\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}])$. If each pair of $\mathcal{X}_i \in [\mathcal{X}_0, \mathcal{X}_1, \cdots \mathcal{X}_{m-1}]$ and $\mathcal{Y}_i \in [\mathcal{Y}_0, \mathcal{Y}_1, \cdots \mathcal{Y}_{n-1}]$ that $\frac{\partial \mathcal{Y}_i}{\partial \mathcal{X}_i} \neq 0$ satisfies $\forall ey \in \mathcal{Y}_i, \left|\{ex | \frac{\partial ey}{\partial ex} \neq 0, ex \in \mathcal{X}_i\}\right| \leq 1$, in which ex represents a scalar data element in tensor $\mathcal{X}_i$ and ey represents a scalar data element in tensor $\mathcal{Y}_i$. It indicates that all data elements in any of the output tensors rely on at most one data element in one input tensor. We call a subgraph with such property as an *element-wise subgraph* (**or EWS**).

[4] In code generation, *schedule* means how the threads are mapped to hardware to process the data (e.g., tiling size, on-chip resource configuration, parallelism configuration for GEMM code generation).



```
Parallel DOALL For i = blockIdx.x to M:
  Parallel DOACROSS For j = threadIdx.x to N:
    Compute Sub_DOALL_Loop_1(j)
  Run Collective Block Reduction
  Parallel DOALL For j = threadIdx.x to N:
    Compute Sub_DOALL_Loop_2(j)
```

Figure 7: Context-aware instruction rescheduling for `softmax` computation.

Otherwise, if there exists an output tensor element that relies on multiple input tensor elements, the subgraph contains contraction operations that combine several data elements into one (one-on-many element-level data dependency). We refer to such a subgraph as a *contracted subgraph* (**CS**). In machine learning graphs, contractions are often represented by `reduce` operations (e.g., reduce-sum) in the intermediate representation (IR). If all the `reduce` ops of a subgraph are the output operations, we call the subgraph an *output-only contracted subgraph* (**OCS**). Note that a *CS* is either an *OCS* or could be decomposed into *OCS* and *EWS*.

Basic Codegen Scheme. MonoNN will first identify all the largest *OCS* through reverse traversal on the memory-intensive subgraphs. The remaining subgraphs are then *EWS*, which can be converted into DOALL loop [13] (i.e., loop with no inter-iteration dependency) for full parallelization.

An *OCS* contains the contraction computation in `reduce` op. The `reduce` ops in typical inference graphs are doing contraction over elements residing in a continuous address in memory (e.g., $Reduce([x,y]) => [x,1]$). For `reduce` ops on GPU, the non-contracted dimension (e.g., x in the above example) forms a DOALL loop *without* inter-iteration dependency. Whereas the inner contracted dimension (e.g., y in the above example) forms a DOACROSS loop [13] *with* inter-iteration dependency due to contraction computation. Note that in some cases, it might be beneficial to use a uniform schedule for adjacent subgraphs with loop fusion if certain locality constraints are met. For example, in Fig.7(a), an *OCS* followed by an *EWS* can use a uniformed schedule by fusing the outer loop, as the output of *OCS* can buffer on on-chip cache for subsequent read from *EWS*. This technique, also known as stitch fusion [42], is shown as dotted lines in Fig.7(a). We call subgraphs that have independent schedule *schedule-independent subgraphs*, *SIS* in short. Several subgraphs that use a uniform schedule after loop fusion are regarded as one *SIS* (e.g., Fig.7(b) shows an *SIS* after fusing an *OCS* and an *EWS*). The schedule within an *SIS* is constrained by loop structure and block locality, while the schedules among different *SIS* are independent. Different *SIS* with its own schedule is finally stitched together under MonoNN with global memory buffering for intermediate transferring.

4.2.2 Context-Aware Instruction Rescheduling

We illustrate how to enable the hidden parallelism given an *SIS* subgraph with the example in Fig.7(a). Note that the contracted dimension of the `reduce` op is *N*, which maps to the inner loop (i.e., parallel threads within a thread block), The non-contracted dimension is *M*, which maps to the outer loop (i.e., different thread blocks).

According to the property of *OCS*, it can be divided into a sub-*EWS* followed by a `reduce` op. Thus, the inner loop of *OCS*, which is a DOACROSS loop, can be converted to a sub-DOALL loop (Fig.7(c)) followed by the corresponding reduction. With the conversion above, the inner-loop of the *SIS* is converted to the computation sequence of "*sub-DOALL loop* ⇒ *reduction* ⇒ *sub-DOALL loop*" (Fig.7(b)-(c)).

The key insight of context-aware instruction rescheduling is to rearrange the instructions according to the property of the DOALL loop.

In Fig.7(c), the DOALL loop has no inter-iteration dependencies, allowing MonoNN to explore the default schedule as well as the ILP enhanced schedules (e.g., 2*X* ILP) by merging instructions from different iterations into parallel instructions within the same iteration to ensure stall-free instruction issue. Theoretically, all schedules shown in Fig.7(c) achieve near-maximum overall parallelization ($TLP \times ILP$), with the default schedule yielding varied TLP for different operators, thus necessitating numerous GPU kernels. In contrast, MonoNN can identify a schedule that maximizes overall parallelization and optimally fits TLP into a single monolithic kernel. The optimization space for utilization abundant registers (Fig.7(d)) and corresponding performance in the monolithic kernel will be explored, as determining the best scheduling factor involves balancing ILP and resource usage. We will discuss

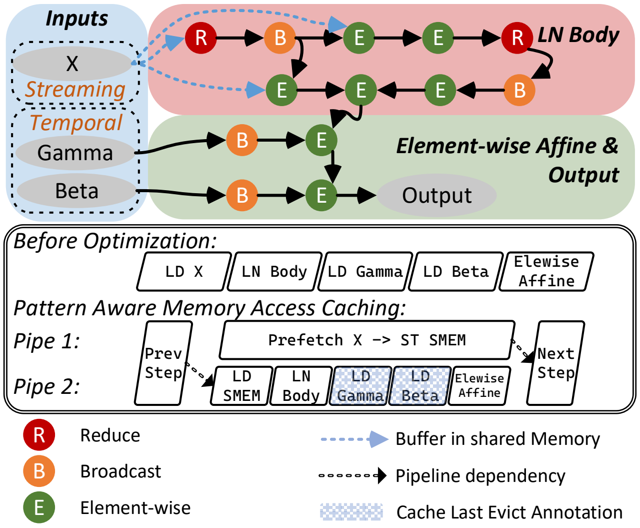


Figure 8: On-chip resource exploitation for layer norm.

how to find the optimal rescheduling factor for each *SIS* in Sec.5

4.3 On-Chip Resource Exploitation

For an SIS memory-intensive subgraph, global memory access often takes up a significant amount of execution time, particularly when thread-level parallelism (TLP) is limited in a monolithic context. Along with boosting parallelism through instruction rescheduling as discussed in Sec.4.2.2, MonoNN performs a comprehensive on-chip memory resource exploitation to maximize the use of memory resources based on access patterns.

We observe that there are two major memory access patterns for an *SIS* subgraph: (1) *Streaming*: Each element of the input tensor is accessed once in the computation graph. (2) *Temporal*, Each data item in the input tensor is read multiple times by its consumers, commonly due to `broadcast` operators in modern ML models. MonoNN implements a series of memory access optimizations based on these patterns.

(I) Streaming Access Optimization. MonoNN leverages the abundant shared memory resource allocated by the compute-intensive computations in the monolithic kernel to pipeline the streaming global memory access with other computations. As mentioned, the outer loop of an *SIS* subgraph is a DOALL loop, where different iterations are independent. MonoNN organizes the computations between different iterations of the outer loops to form a computation pipeline and a memory copy pipeline. Particularly, during the computation of each outer loop iteration, it will prefetch the streaming accessed input data for the next iteration into the shared memory buffer. Fig.8 presents the input data access pipelining for layer norm [15]. *LN Body* represents the main layer norm computation and *Element-wise affine* represents the following element-wise affine transformation parameterized by *Gamma* and *Beta*. The memory access to input *X* is prefetched onto shared memory in the computation pipeline, fully overlapping the data fetching and computation. Note that the input *X* in Fig.8 is consumed by multiple operators. If the shared memory is not enough for the data buffering, MonoNN will not make a pipelined buffer *X*. Instead, MonoNN will buffer *X* on the register file (or local memory if facing register spills), for which the multiple consumers will reuse the data through the faster register file rather than the global memory.

(II) Temporal Access Optimization. If the input data access is temporal rather than streaming, MonoNN will annotate cache hints to these memory operations to guide the cache behavior to preserve the data on the cache as long as possible (e.g., *evict_last* in NVIDIA GPU semantics). MonoNN will annotate memory read as temporal access from the node with a smaller tensor shape until an empirical value is reached to accommodate as many tensors as possible and prevent cache thrashing. As shown in Fig.8, *Gamma* and *Beta* have temporal locality because they are connected to the subsequent `broadcast` op. A load of *Gamma* and *Beta* will be annotated with *evict_last* for longer cache occupation, improving the temporal locality in *SIS*.

4.4 Global Thread Barrier Merging

As mentioned in Sec.4.2, there are cross thread block data dependencies between different *SIS* subgraphs. MonoNN inserts global thread barrier (**GTB**) between *SIS* subgraphs to ensure correctness. Note that *GTBs* are also required between compute-intensive operators and *SIS* subgraphs. Similar with [42], *GTB* in MonoNN is implemented in two stages: one-block-wait-all and one-block-notify-all. Each block has a flag in global memory (typically cached in GPU L2) to represent whether the corresponding thread block has arrived. The first thread block waits for all the remaining blocks to report waiting, and then notifies them to proceed. Furthermore, the inner-kernel *GTB* is much shorter than the non-computation overhead as the latter is composed of both kernel launching and framework scheduling overheads. The overhead measurement results of inner-kernel *GTB* [42] and kernel launching [37] from the previous studies are aligned with our observation that a typical kernel launching overhead is often multiples of a *GTB* length, e.g., a single kernel launching with framework scheduling is around $8 \sim 10$ microseconds which is $4 \sim 5\times$ of a *GTB* length.

Longest-path based GTB merging. One *GTB* introduces minimal synchronization overhead, but this can accumulate when the number of *GTBs* is large. We have observed that some *SIS* subgraphs do not exhibit producer-consumer or topology dependencies. By clustering these independent *SIS* subgraphs in topological order, MonoNN can eliminate the need for *GTBs* between them. To address graph complexity, we propose the *longest-path based GTB merging* approach to find the optimal *SIS* clustering strategy. For example, in Fig.9, the nodes (A-E) represent the *SIS* subgraphs, with *GTBs*

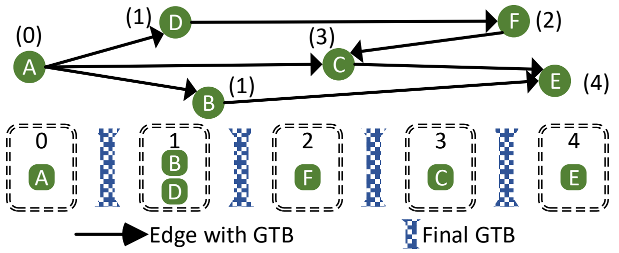


Figure 9: Global thread barrier merging. The numbers above nodes represent the longest distance from the source node to the current node.

required between them (indicated by edges). MonoNN calculates the longest path to each node from the first node. Nodes with the same longest path length (e.g., B and D in Fig.9) can be clustered together for *GTB* merging. Traditional topology ordering methods only order nodes and do not cluster them, making them inadequate for guiding *GTB* merging.

4.5 Optimizing Compute-Intensive Operators

While the memory-intensive subgraphs are effectively optimized to maximally leverage on-chip resources, the compute-intensive operators in MonoNN directly adopt the existing implementations from CUTLASS [1] as tunable basic building blocks: the tuning space of CUTLASS is included in the tuning space for the monolithic kernel.

5 The MonoNN Compiler

This section details the design and implementation of an optimizing compiler that automatically generates efficient monolithic kernels using the techniques outlined in Sec.4. Unlike previous works that focus on tuning single operators or subgraphs [18,42], MonoNN optimizes the entire graph, resulting in a vast optimization space. This complexity makes finding the optimal global configuration challenging. We explain how we systematically abstract this extensive optimization space in Sec.5.1 and how we reduce it to efficiently identify a suitable global configuration in Sec.5.2.

5.1 Optimization Space Abstraction

First, we categorize the proposed optimizations into two types: (1) Deterministic optimizations are always beneficial. Including *comprehensive on-chip resource exploitation* (Sec.4.3) and *global thread barrier merging* (Sec.4.4). (2) Tunable optimizations: All other optimizations not included in the deterministic category are considered tunable. We classify the tunable factors of the monolithic kernel into three main classes:

(I) Code generation schedule of each operator in a neural network. In the code generation process, element-wise operators follow the code generation schedule of `reduce` operators through input-inline. Thus, we only need to tune the schedule of `reduce` operators and compute-intensive operators (*CI Ops*). Note that a grid-stride loop will be used to iterate over its input elements if an element-wise operator cannot find a `reduce` or *CI Ops* that it associates with. There are two common schedules for row-major `reduce` operators. One is to reduce a row of elements with all threads in a thread block. The other one is to reduce a row with one warp. For *CI Ops*, MonoNN will jointly consider all the tunable factors, including tiling size, on-chip resource configuration, parallelism configuration, hardware intrinsic (e.g., Tensor Core instruction and CUDA async-copy), input prefetching, etc.

(II) Context-aware instruction rescheduling factor. ILP is important for the *SIS* subgraphs to compensate for parallelism loss under the constraint TLP in a monolithic kernel. A too-small rescheduling factor may be insufficient to improve the overall parallelism. A rescheduling factor that is too large will use massive registers and may cause register spilling. MonoNN explores a spectrum of the rescheduling factors for each memory-intensive operator (*MI Op*). Specifically, for each *MI Op*, MonoNN explores up to 32*X* rescheduling factors [5] via *context-aware instruction rescheduling*. In Sec.7.2.1, we quantitatively evaluate how different ILP rescheduling factors impact *MI Ops* on performance.

(III) TLP and on-chip resource of the overall monolithic kernel. TLP on GPUs is defined as the thread block size and number of blocks for a GPU kernel, and on-chip resource constraints are critical performance factors for efficient program execution. For the monolithic kernel, these factors not only affect the optimal execution configuration (e.g., tiling size) for *CI ops*, but also impact the optimal rescheduling for *context-aware instruction rescheduling* and optimal code generation schedule for memory-intensive subgraphs (e.g., warp reduction vs block reduction). The candidate block sizes for tuning are 128 and 256 for MonoNN, which are the main block sizes used in the existing machine learning compilers [2, 7] and CUTLASS for achieving good performance for both *CI Ops* and *MI Ops*. Other block sizes may also be trivially included to the optimization space. Our monolithic kernel requires that all thread blocks be able to be scheduled onto GPU concurrently in one wave to avoid deadlock in synchronization. Thus, the total thread block number should be no more than the max number of thread blocks that GPU can tolerate. Specifically, the number of candidate TLP choices is $N_{TLP} = \left|\{128, 256\}\right| \times N_{blocks-per-sm} = 2 \times N_{blocks-per-sm}$, where $N_{blocks-per-sm} = \left|\{1, 2, ..., N_{max-blocks-per-sm}\}\right|$ We empirically choose $N_{max-blocks-per-sm}$ as 5 because too many co-existing thread blocks will result in insufficient available on-chip cache per block and further slow down *CI Ops*.

[5] The range of ILP is constrained by on-chip resources and thus is up to 32 for hardware we evaluated.

5.2 Schedule-Independent Group Tuning

5.2.1 Extremely Large Tuning Space

The tuning complexity on the optimization space is up to: $O_{naive} = (S_C)^{N_C} \times (S_M \times N_{ILP})^{N_M} \times N_{TLP}$.[6] Indicates the code generation schedules for *CI ops* ($(S_C)^{N_C}$), schedules and ILP sizes for *SIS* ($(S_M \times N_{ILP})^{N_M}$), and candidate TLP sizes for the monolithic kernel (N_{TLP}). All operators in an *SIS* share the uniform schedule. If there exists a `reduce` operator in the *SIS*, we only need to enumerate the schedule of one reduce operator; otherwise, it will adopt a grid-stride loop as the schedule. A uniformed ILP will apply to all operators in the same *SIS* since all operators in the same *SIS* share similar resource and parallelism requirements. Unfortunately, this is an excessive tuning space and will have a size of approximately 10^{500} for a BERT-base model.

5.2.2 Tuning Space Compression

We make two important observations for the monolithic kernel. (1) A monolithic kernel is separated into a set of *SISs* and *CI Ops* by *GTBs*. The code generation schedules for different subgraphs are not interleaved. We call an *SIS* or a *CI Op* as a *schedule-independent group* (SIG). (2) The connection between two *schedule-independent groups* is the TLP and on-chip resource allocation. Meanwhile, the overall kernel's TLP and on-chip resource allocation are fixed throughout the monolithic kernel. According to the observations above, the code generation schedule of different *SIGs* can be safely tuned individually without missing the optimal solution.

Based on the above observations, we propose *schedule-independent group tuning* to compress the tuning space significantly. Different *SIGs* are tuned independently for each candidate TLP setting. Particularly, MonoNN concatenate the best-tuned configurations of all the *SIGs* to get the overall best configuration. Finally, we chose the TLP setting that performs the best and all its associated configurations.

For a *schedule-independent group* that is a *CI Op*, we enumerate S_C code generation schedules. There are N_C such groups, and the overall complexity is $N_C \times S_C$ under each overall TLP configuration. For a *schedule-independent group* that is an *SIS* subgraph, we enumerate the possible code generation schedules and overall ILP sizes. There are N_M such groups, and the overall complexity is $N_M \times S_M \times N_{ILP}$ under each overall TLP configuration. As a result, the shrunken tuning complexity of our monolithic kernel is up to:

$$O_{opt} = (N_C \times S_C + N_M \times S_M \times N_{ILP}) \times N_{TLP}.$$

It is worth noting that MonoNN will check the *SIG* hash and reuse the tuning result if an identical *SIG* has been tuned previously. This will prevent duplicated tuning effort under repetitive neural network layers.

Algorithm 1 Monolithic Kernel Tuning

```
 1: procedure GETTUNINGSPACETLP
 2:     $C_{block-size} \leftarrow \{128, 256\}$
 3:     $C_{blocks-per-sm} \leftarrow \{1, ..., N_{max-blocks-per-sm}\}$
 4:     return $C_{block-size}\ \mathcal{X}\ C_{blocks-per-sm}$
 5: procedure OPTIMIZEMISIS(SIS, TLP)
 6:     $CandidateILPFactors \leftarrow \{1, 2, 3, ..., 32\}$
 7:     $BestSolution, BestTime \leftarrow NULL, \infty$
 8:     for $ILP \in CandidateILPFactors$ do
 9:         $S, Time \leftarrow ProfileAndOptimize(SIS, TLP, ILP)$
10:         if $Time < BestTime$ then
11:             $BestSolution, BestTime \leftarrow S, Time$
12:     return $BestSolution, BestTime$
13: procedure OPTIMIZEFORTLP(TLP)
14:     $Solution, TotalTime \leftarrow \{\}, 0$
15:     for $SIG \in GetAllSIG()$ do
16:         if $IsCIOp(SIG)$ then
17:             $S, Time \leftarrow ProfileAndOptimize(SIG, TLP)$
18:         else                                              ▷ Is MI SIS
19:             $S, Time \leftarrow OptimizeMiSIS(SIG, TLP)$
20:         $Solution \leftarrow Solution \cup \{S\}$
21:         $TotalTime \leftarrow TotalTime + Time$
22:     return $Solution, TotalTime$
23: procedure MONONNTUNE
24:     $BestSolution, BestTime \leftarrow NULL, \infty$
25:     for $TLP \in GetTuningSpaceTLP()$ do
26:         $S, Time \leftarrow OptimizeForTLP(TLP)$
27:         if $Time < BestTime$ then
28:             $BestSolution, BestTime \leftarrow S, Time$
29:     return $BestSolution, BestTime$
```

Algo.1 details the tuning procedure in MonoNN begins with sampling *MonoNNTune* in line 25. MonoNN takes Cartesian product between candidate block size $C_{block-size}$ and co-existing blocks per SM $C_{blocks-per-sm}$ (line 4). The optimal solution under each TLP will be tuned independently (line 26). MonoNN will optimize every *SIG* in the neural network (line 15). For memory-intensive subgraphs, the best solution across all rescheduling factors will be selected as the final solution of the current subgraph (line 10-12). MonoNN iterates over the solution under each distinct TLP and chooses the one with the shortest duration as the final solution (line 26-29).

5.3 Implementation

We implement MonoNN with 64k lines of C++ code on top of XLA compiler [2] and integrate it into TensorFlow [12] framework as a drop-in replacement to the backend execution engine. This allows MonoNN to accelerate existing TensorFlow models without requiring any code changes. Additionally, MonoNN can compile a neural network into a standalone assembly file that can be directly executed, potentially offering better performance by eliminating the runtime overhead from the deep learning framework. In Sec.7, we only report the performance number from the first mode for a fair comparison across frameworks. Unlike AStitch [42], MonoNN does not support cross-block reduction. We are not aware of any performance degradation on evaluated models as the

[6] N_C (or N_M): number of *CI ops* (or *SIS*). S_C (or S_M): possible schedules of *CI ops* (or SIS). N_{ILP} (or N_{TLP}): possible ILP (or TLP) sizes.

reduction dimension for these popular models are all small.

6 Scope, Impact, and Limitations

The current optimization scope of MonoNN mainly focuses on general static DNNs where different layers have similar computational sizes. For example, MonoNN effectively supports various popular Transformer models without dynamic control flows, including Transformer encoder models such as BERT, encoder-decoder models such as T5, and every step of the decoder models such as GPT-like models. We compare the performance of different fusion granularity in Sec.7.3, ranging from basic element-wise fusion, stitch fusion, layer-wise monolithic kernel (i.e., one monolithic kernel per layer) to a single monolithic kernel of the entire model.

MonoNN established a new monolithic optimization space for common static DNN inference scenarios by resolving the long-existing global optimization challenge within a single kernel, addressing the resource incompatibility problem of various operators. MonoNN introduce key contributions such as *Context-Aware Instruction Rescheduling* (Sec.4.2.2), *On-Chip Resource Exploitation* (Sec.4.3), and *Global Thread Barrier Merging* (Sec.4.4). Moreover, MonoNN is designed to be forward-looking, performing even more effectively for the upcoming GPU architectures. The increased computing power of future GPUs will likely exacerbate issues related to off-chip memory access and CPU-GPU context switch overhead. Moreover, as supported by [6], distributed shared memory access can enable more flexible and efficient intermediate data buffering for large-scale operator fusion.

Despite the contribution of MonoNN, several potential limitations should be noted. (1) MonoNN mainly addresses common static DNN inference scenarios rather than the models with dynamic control flows [22, 36]. However, users can still optimize the subgraphs separated by control flow operators using MonoNN techniques. (2) MonoNN focuses on DNN inference scenarios that fit within a single GPU, covering a wide range of real-world inference service cases. Extending MonoNN to incorporate collective communication primitives is beyond the scope of this work, but users can still optimize the subgraphs separated by the communication operators using MonoNN. (3) MonoNN may be less effective for DNNs with varied tensor sizes in different layers due to the imbalanced workloads in the single monolithic kernel. While our experiments did not show significant performance regression, this potential limitation in the monolithic kernel should be highlighted.

7 Evaluation

Model specifications: We use a set of representative machine learning applications as our evaluation workloads, including BERT-Base, BERT-Large [19], Transformer T5-Small, T5-Base [30] for natural language processing, ViT [20] for image recognition (with both Convolution and Transformer components), CLIP [29] for computer vision and text, OPT [38] for text generation (OPT-125M version). All the models are publicly available from Huggingface [34]. For all BERT-like and Transformer-like models, we used sequence length equal to 128 unless specified elsewhere.

Software specifications: We compare MonoNN against TensorFlow [12] (v2.7), XLA [2] (v2.7), TensorRT v8.2[7] (via TF-TRT integration [7]), TVM (commit f6f9056) [18], AStitch [42], Rammer [24], PyTorch [28] (v1.12.1), and CUDAGraph [9] (via PyTorch integration). We use CUDA v11.6 and cuDNN 8 for all the experiments[8]. We enable Tensor Cores for all the frameworks we evaluated except in Sec.7.5 cause Rammer [24] only supports SIMT cores.

Hardware Platforms: *A10 server*: NVIDIA A10 GPU (Ampere), and two Intel(R) Xeon(R) Platinum 8369B CPUs. *T4 server*: NVIDIA T4 GPU (Turing), and two Intel(R) Xeon(R) Platinum 8163 CPUs. *A100 server*: NVIDIA A100 80GB SXM (Ampere), and two Intel(R) Xeon(R) Platinum 8369B CPUs.

7.1 End-to-End Performance Comparison

7.1.1 Overall Results

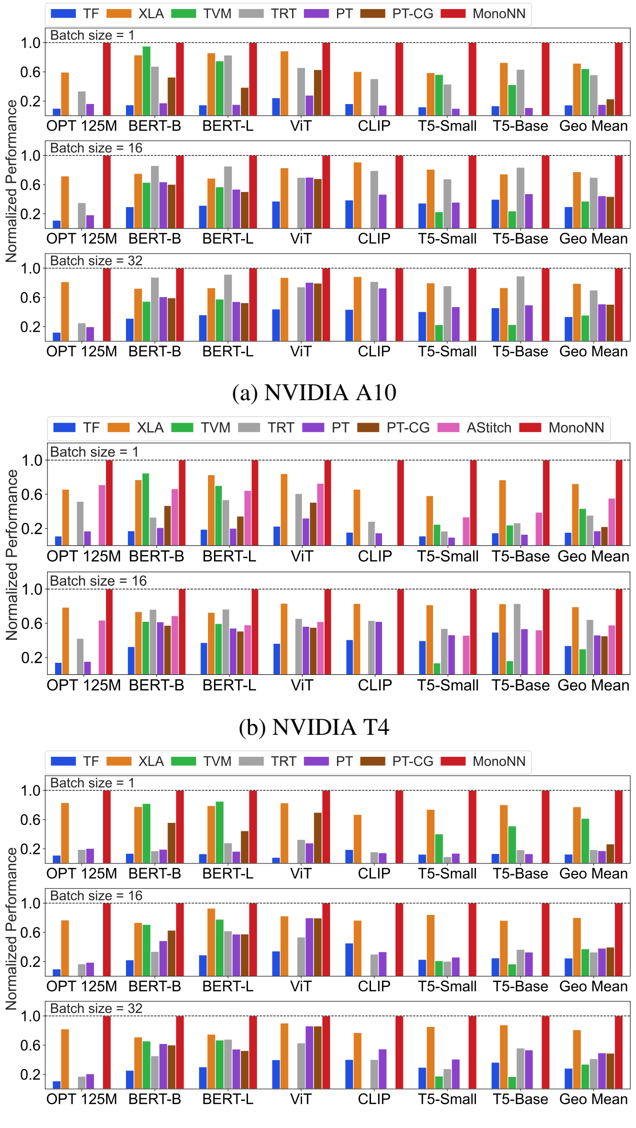
Fig.10 shows the end-to-end performance speedup on NVIDIA A10, T4, and A100 GPU for all experiments with three batch size variations. *Geo Mean* refers to the geometric mean across all models. All the execution time is normalized against the best optimizer in the group. TVM failed to optimize OPT, ViT, and CLIP due to incomplete operator support. PyTorch-CUDAGraph failed to optimize OPT, CLIP, and T5 for unsupported operations in the graph-capturing phase.

As demonstrated in Fig.10, on A10 GPU, MonoNN achieve 6.9×, 1.4×, 1.6×, 1.8×, 6.6×, and 2× average speedup over Tensorflow, XLA, TVM, TensorRT, PyTorch, and PyTorch-CUDA Graph on batch size 1, respectively. In addition, for batch size 16, and 32, MonoNN achieve on average 1.88×, and 1.80× speedup over baselines. On NVIDIA T4, MonoNN achieve 6.5×, 1.4×, 2.3×, 2.8×, 5.8×, 2.3×, and 1.8× average speedup over Tensorflow, XLA, TVM, TensorRT, PyTorch, PyTorch-CUDA Graph, and AStitch on batch size 1, respectively. In addition, for batch size is 16, MonoNN achieves on average 1.91× speedup over all baselines. We also have comprehensively tested MonoNN on A100 as detailed in Fig.10c.

Given the diverse set of baselines we compare against, the extent of performance improvements can vary. MonoNN consistently achieves the best performance across all baselines, with significant improvements observed for all testing batch sizes. It is important to note that reduced performance gains with larger batch sizes are anticipated, as

[7] TensorRT 8 is the latest version at the time of submitting the paper (Dec. 2022), with much performance improvement compared to TensorRT 7.

[8] AStitch is using its released artifact (CUDA 10.2 and cuDNN 7).



(a) NVIDIA A10

(b) NVIDIA T4

(c) NVIDIA A100

Figure 10: MonoNN End-to-End speedup (higher is better).

larger batches generally lead to better device utilization, leaving less room for performance enhancements. For example, XLA/TensorRT/MonoNN show on average 4.9×->2.6×->2.3× / 3.8×->2.3×->2.0× / 6.9×->3.4×->3.0× performance gain over the TF baseline when expanding the batch size from 1 -> 16 -> 32 on A10 respectively.

We test CUDA Graph via Pytorch integration. Despite the failure in some of the models in our benchmark, CUDA Graph achieves on average 2.6×, 0.95×, 0.98× speed up over PyTorch when batch size is $1, 16, 32$ on A10. Obviously, the performance gain of the CUDA Graph diminishes drastically (even with no performance gain) when the batch size is larger than one. The average speedup is far less than the achieved performance speedup of MonoNN. Specifically, MonoNN outperforms PyTorch-CUDA Graph by an average of 2× (batchsize=1) and continues to outperform it when batchsize is larger than 1. We attribute the reason as follows. On the one hand, MonoNN can perform various optimizations in the monolithic optimization space that CUDA Graph cannot, e.g., whole graph-level optimizations, instruction rescheduling, on-chip resource exploitation, and *GTB* merging. On the other hand, as pointed out by previous literature [37], a new GPU kernel has several types of overhead (e.g., kernel launching, kernel initialization) but CUDA Graph can only optimize kernel launching.

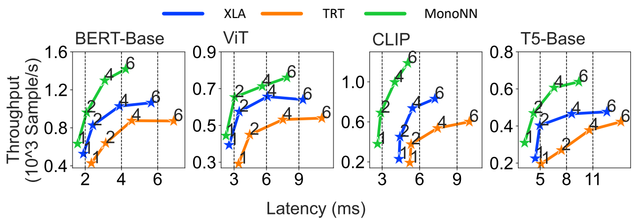


Figure 11: MPS Performance.

XLA achieves the best average speedup among our baselines. But XLA can only explore register-level data buffering rather than multi-dimensional optimization techniques in MonoNN. We only run AStitch [42] experiment on T4 GPU (with CUDA 10.2) because the artifact released does not support newer NVIDIA A10 architecture.

In addition, we evaluate MonoNN and the baselines on A10 using longer input for the BERT model. We use an input sequence length equal to 512, which is the maximum sequence length supported by the model's pre-trained positional embedding. As illustrated in Tab.1, MonoNN achieves on average 1.94×, 1.52×, and, 1.48× speedup over baselines when batch size is 1/16/32 respectively.

	TF	XLA	TRT	PT	PT-CG	MonoNN
BS=1	0.27	0.90	0.44	0.39	0.57	1
BS=16	0.40	0.87	0.75	0.63	0.62	1
BS=32	0.42	0.90	0.79	0.63	0.63	1

Table 1: Normalized performance.

Among the evaluated models, the OPT-125M model has the smallest computation shape; only a single output token is generated at each step. This results in severe non-computation overhead, making MonoNN especially advantageous.

7.1.2 Impact on Throughput with MPS

This section is to demonstrate that MonoNN's optimizations can perform well for GPU-shared scenarios for higher throughput. In the real-world inference scenario, a common approach is to share a single GPU with multiple inference tasks to improve inference throughput and hardware utilization. NVIDIA Multi-Process Service (MPS) [10] is one of the most widely adopted solutions for GPU sharing. We test our solution with *MPS* for BERT-Base, ViT, CLIP, and T5-Base on A10 and plot the latency-throughput curve in Fig.11. The

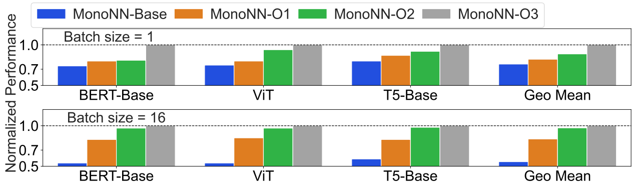


Figure 12: Ablation study on NVIDIA A10.

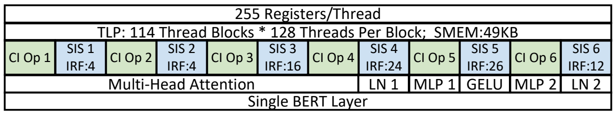
numbers on the line indicate how many instances are used in *MPS*. The batch size is one in this experiment. MonoNN consistently outperforms baselines, achieving 1.5 × −2× QPS throughput under the same latency constraint. It demonstrates that a monolithic kernel is capable of delivering meaningful speedup in GPU-shared inference. In practice, we divide TLP for each instance by the number of instances co-existing in *MPS* to ensure all instances can run concurrently on a single GPU.

7.1.3 Ablation Study

Fig.12 dissects the main optimizations in MonoNN. We build *MonoNN-Base*, a lightweight monolithic kernel generator that has all the optimization techniques of MonoNN except for the three: *context-aware instruction rescheduling*, *comprehensive on-chip resource exploitation* and *global barrier merging*. Note that *MonoNN-Base* is different from the baselines in Fig.10. *MonoNN-Base* is already a strong baseline that has many basic optimization techniques. It is a single monolithic kernel with minimal non-computation overhead, achieving better performance than TensorFlow and on par with XLA. We then build *MonoNN-O1-O3* by gradually applying the above optimizations one by one in order. *MonoNN-O3* is the full MonoNN. We observe 5%, 6%, and 12% speedup for *O1*, *O2*, and *O3* optimization on batch size 1, and 35%, 15%, and 3% speedup on batch size 16. *Context-aware instruction rescheduling* shows much more performance gain for batch size 16 because larger tensor shapes need higher parallelism, thus requiring ILP compensation more. In addition, we observe instruction rescheduling does not improve performance on *OPT-125M* model as the text generation model only produces a single token in each inference and a small tensor shape does not need a larger rescheduling factor. *Comprehensive on-chip resource exploitation* also shows higher performance gain on batch size 16 as larger tensors need more comprehensive solutions to accelerate off-chip memory access. *GTB Merging* shows larger performance gain when batch size is one because synchronization overhead is invariant to batch size and thus will take a larger portion when kernel duration is short.

7.2 MonoNN Optimization Breakdown

In this section, we dissect our optimization techniques proposed in Sec.4 and present a deep-dive into the solution generated by MonoNN with both conceptual and quantitative analysis to help understand optimizations in monolithic kernel better. Fig.13 shows the identified *CI Ops* and *SIS* in a BERT-Base model. We present a detailed analysis when the inference batch size is 16 on NVIDIA A10.



255 Registers/Thread

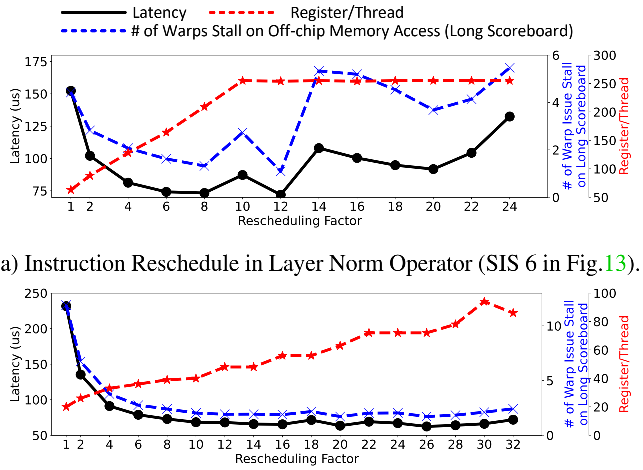
TLP: 114 Thread Blocks * 128 Threads Per Block; SMEM:49KB

CI Op 1 | SIS 1 IRF:4 | CI Op 2 | SIS 2 IRF:4 | CI Op 3 | SIS 3 IRF:16 | CI Op 4 | SIS 4 IRF:24 | CI Op 5 | SIS 5 IRF:26 | CI Op 6 | SIS 6 IRF:12

Multi-Head Attention | LN 1 | MLP 1 | GELU | MLP 2 | LN 2

Single BERT Layer

Figure 13: Identified *SIS* and *OCS* on a BERT layer, including TLP and on-chip resource usage of the monolithic kernel and instruction rescheduling factor (IRF) for each *SIS*.



(a) Instruction Reschedule in Layer Norm Operator (SIS 6 in Fig.13).

(b) Instruction Reschedule in GELU Operator (SIS 5 in Fig.13).

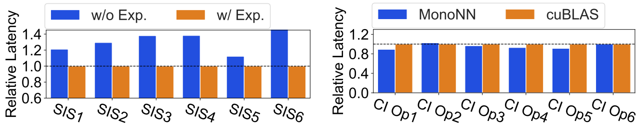
Figure 14: Context-aware instruction rescheduling analysis.

7.2.1 Context-Aware Instruction Rescheduling Analysis

Context-Aware Instruction Rescheduling (Sec.4.2.2) can increase ILP with more register usage. We show the rescheduling analysis of two subgraphs in Fig.14. As demonstrated in Fig.14a, when the rescheduling factor is too low, the average number of warps per SM per cycle that stall on off-chip memory access is high due to the low parallelism (both TLP and ILP), resulting in high inference latency. On the other hand, a too-high factor will cause register pressure and even register spilling. Slight register pressure often does not indicate degradation in performance, but register spilling often results in drastic performance degradation. We observe register pressure when the rescheduling factor is 10 and register spilling when the rescheduling factor is larger than 22. The best factor for *SIS6* is 12. *SIS5* in Fig.14b has less register usage compared to *SIS6*, for which the best rescheduling factor is 26.

7.2.2 On-chip Resource Exploitation Analysis

Comprehensive On-chip Resource Exploitation (Sec.4.3) can further exploit the on-chip cache and shared memory based on the data access pattern of the subgraph. Fig.15a shows performance improvement after applying this optimization



(a) *SIS* latency after Exploitation. (b) *CI Ops* latency.

Figure 15: Operator latency breakdown.

for subgraphs corresponds to Fig.13, achieving 1.3× speedup on average. Note that this is additional performance gain over *Context-Aware Instruction Rescheduling*.

7.2.3 Performance of Compute Intensive Operators

We also detailed the performance of *CI Ops* in the monolithic kernel. All *CI Ops* need to follow the same TLP setting but the tensor shape for each *CI Op* could be different. Thus, handling different tensor shapes with a unified TLP setting is critical. MonoNN achieves this by leveraging intra-thread block tuning choices from CUTLASS. Through extensive evaluation, we found that intra-block tuning can generate satisfactory solutions for *CI Ops*. We illustrate the performance of *CI Ops* in monolithic kernel in Fig.15b. With the highly-tuned open-source vendor code (i.e., CUTLASS), all operators achieve on-par performance with the cuBLAS. Surprisingly, in some cases, the *CI Op* found by MonoNN is slightly better than cuBLAS. The reason we judiciously suspect is MonoNN performs an exhaustive search over all possible solutions whereas cuBLAS uses heuristics.

7.2.4 Global Thread Barrier Merging Analysis

GTB is necessary for a monolithic kernel to ensure correctness. But each *GTB* involves a small overhead, approximately 2*us* based on our evaluation. Thus we need to minimize such overhead with *longest-path based GTB merging* (Sec.4.4). We compare *GTB* number before and after merging optimization. We observe 516, 658, 319, 710, and 366 *GTBs* in BERT-Base, CLIP, OPT-125M, T5-Base, and ViT model respectively. The *GTB* number reduced to 146, 185, 185, 315, and 184 respectively after *GTB* merging.

7.2.5 Dissecting Non-computation Overhead

	Bert-Base	ViT	T5-Base	OPT-2
Framework	0.41	0.57	0.44	0.61
Kernel Launch	0.71	0.51	0.72	1.74

Table 2: Context switch overhead breakdown (in ms).

Framework scheduling overhead and kernel launching overhead are two major sources of non-computation overhead. Tab.2 shows the separated framework scheduling and kernel launching overhead after optimization with XLA. It shows that the kernel launch overhead accounts for 61.7% of the overhead on average, larger than that of framework overhead. To measure the two kinds of overhead, we build two XLA variants. The first variant *XLA-framework* executes all framework scheduling logic as XLA, except it does not launch GPU kernel but returns immediately for each operator. Therefore, the inference latency of *XLA-framework* is pure framework overhead. The second variant *XLA-framework-and-kernel* has the same functionality as XLA, except that it launches empty GPU kernels (GPU kernels that do nothing) rather than the original kernels. The inference latency of *XLA-framework-and-kernel* is the summation of framework overhead and kernel launch overhead.

7.3 Fusion Granularity Analysis

	EleWise	Stitch	Layer	Layer+CUDAGraph	Monolithic
OPT	0.68	0.73	0.93	0.93	**1**
MultiModal	0.49	0.61	**1.10**	**1.10**	1

Table 3: Relative performance at each fusion granularity

We further analyze how kernel fusion granularity impacts inference performance on the models we evaluated to have a better understanding of the optimization space we proposed. Specifically, we control the fusion scope of MonoNN to generate code at different fusion granularity. From small to large, 1) **EleWise**: Element-wise fusion [2, 18]. 2) **Exhaustive memory-intensive fusion (Stitch)**: perform exhaustive fusion optimization on memory-intensive subgraphs using shared memory and global memory. This scope is similar to TensorRT [11] and AStitch [42]. 3) **Layer**: each layer of the neural network will be generated into a kernel with monolithic optimization. Note that from this scope, efficient code generation is unrealistic without the techniques proposed in this work. 4) *Layer+CUDAGraph* additionally apply CUDA Graph to the generated kernels. 5) *Monolithic*: the entire neural network is fused into a single kernel.

We choose OPT-125M and a customized multimodal model and benchmark them on A10 with a batch size equal to one. The multimodal model contains a transformer-based text encoder and a CNN+transformer-based image encoder. The setting with the best performance is highlighted in bold. We observe for a regular model like OPT with repetitive layers, monolithic kernel trend to achieve the best performance because all the optimization choices are essentially the same across layers. But for the multimodal model with complex structure, we observe the text encoder and image encoder trend to explore different optimization spaces due to divergence in computation tensor shape.

7.4 MonoNN Tuning Speed

MonoNN uses a grid search tuner with caching to tune the entire network. The modern neural network usually has many

	Bert-Base	Bert-Large	T5-Small	T5-Base
MonoNN	22	70	17	68
TVM	172	220	51	116

Table 4: MonoNN end-to-end compilation time in minutes.

repetitious layers so that MonoNN avoids tuning them redundantly by caching the result from previous layers. We further apply many engineering-level optimizations to speed up tuning, which will not be highlighted in this paper. To this end, we found that MonoNN tuner can provide satisfactory tuning speed. We detailed quantitative numbers in Tab.4 collected from A10 GPU.

7.5 Comparison with Rammer

We compare MonoNN with Rammer [24] on BERT-Large inference. Rammer failed in optimizing all Huggingface public models in Fig.10 due to unsupported operators (e.g., Einsum, BroadcastTo). The only common inference model that we can find is the BERT-Large model from Rammer's official repository using fixed batch size 1. Thus, we cannot test other batch sizes on Rammer. In addition, Rammer does not support Tensor Cores. We thus compare with Rammer on NVIDIA T4 GPU after disabling Tensor Core usage for MonoNN. MonoNN shows 1.28× speedup over Rammer on BERT-Large model when using batch size equal to one and sequence length equal to 512.

8 Related Work

Most of the popular ML compilers focus on either single-operator or subgraph-level kernel generation. [16, 18, 26, 31, 32, 39, 41, 43] focus on compute-intensive operators optimization, with basic fusion support for memory-intensive ops, whereas [27, 41, 42] explore the stitch optimization of memory-intensive subgraphs. From graph level, [23, 33] explore graph transformation optimizations to accelerate neural network execution, which is orthogonal to our work.

Notably, holistic optimizations for machine learning workloads have received increased attention in recent years. VersaPipe [40] utilizes persistent-thread technique [14, 17] to execute a computation graph in a pipelined manner, in which the large kernel is spitted into several small kernels to avoid resource incompatibility problem. This approach is not sufficient to support computation graphs with massive operators, like machine learning graphs. Rammer [24] utilizes the persistent-thread technique to support large scope fusion, in which the task re-slicing and scheduling help to fill up execution units. Rammer does not touch the incompatibility problem and cannot support the monolithic optimization of an entire neural network efficiently. For example, the demoed BERT model of Rammer consists of 734 kernels on GPU. The persistent thread scheduling of VersaPipe and Rammer also introduces extra scheduling overhead, while MonoNN applies effective static scheduling to avoid such overhead. Moreover, neither VersaPipe nor Rammer explores the optimizations of on-chip resource exploitation and GTB merging like in MonoNN. BOLT [35] can fuse GEMM and its following operations into single kernels under restricted locality constrain. It cannot generate the monolithic kernel due to the incompatibility problem. Müller et al. [25] manually fuse all operators of a tiny MLP, small enough to fit on-chip, into a single GPU kernel for accelerated execution. In contrast, MonoNN explores a general approach for automatically high-performance code generation for common-sized models. There are ad hoc solutions to speed up single operator (e.g., LayerNorm) with instruction level parallelism on GPU [21]. None of the above work tackles the challenge of monolithic kernel generation.

9 Conclusion

We reveal that the kernel-by-kernel execution scheme is no longer effective in fully utilizing modern GPUs for various machine learning workloads, causing notable non-computation overhead and off-chip memory traffic. We propose the *monolithic kernel* execution scheme to tackle these problems, providing a vast new optimization space. We propose *context-aware instruction rescheduling* and *comprehensive on-chip resource exploitation* techniques to cope with the incompatibility problem between compute-intensive and memory-intensive operators. We systematically abstract the monolithic optimization space and propose *schedule-independent group tuning* approach to compress the extremely large tuning space. We develop a compiler integrating the optimizations automatically. Extensive evaluation on a set of inference tasks demonstrates that MonoNN outperforms state-of-the-art optimizers with on average 2.01× speedup.

ACKNOWLEDGMENT

We appreciate the guidance from the OSDI reviewers and our shepherd during the review and revision process. We also extend our gratitude to Minmin Sun, Feiwen Zhu, Kai Zhu, Zaifeng Pan, Bohua Chen, and Wenyi Zhao at Alibaba Group for their valuable suggestions. This research is funded by Alibaba Group through the Alibaba Innovative Research Program. Donglin Zhuang and Haojun Xia are affiliated with the School of Computer Science at the University of Sydney, Australia, and conducted this work during their internship at Alibaba. Shuaiwen Leon Song is the corresponding author of this paper.

References

[1] Cutlass: Cuda templates for linear algebra subroutines. https://github.com/nvidia/cutlass.

[2] Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla.

[3] Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[4] Nvidia turing gpu architecture. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[5] Nvidia ampere ga102 gpu architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

[6] Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[7] Accelerating inference in tf-trt. https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html, Cited Dec 2022.

[8] Amd cdna architecture. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, Cited Dec 2022.

[9] Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, Cited Dec 2022.

[10] Multi-process service (mps). https://docs.nvidia.com/deploy/mps/index.html, Cited Dec 2022.

[11] Tensorrt. https://developer.nvidia.com/tensorrt, Cited Dec 2022.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[14] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149, 2009.

[15] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[16] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[17] Michael Boyer, David Tarjan, Scott T Acton, and Kevin Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In *2009 IEEE international symposium on parallel & distributed processing*, pages 1–12. IEEE, 2009.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2021.

[21] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[22] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. Cortex: A compiler for recursive deep learning models. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[25] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *ACM Trans. Graph.*, 40(4):36:1–36:16, 2021.

[26] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 883–898. ACM, 2021.

[27] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. Recom: A compiler approach to accelerating recommendation model inference with massive embedding columns. In Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 268–286. ACM, 2023.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[29] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[32] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[33] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: optimizing tensor programs with partially equivalent transformations and automated corrections. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 37–54. USENIX Association, 2021.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[35] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap be-

tween auto-tuners and hardware-native performance. *arXiv preprint arXiv:2110.15238*, 2021.

[36] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 681–699. USENIX Association, 2023.

[37] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching cuda kernels. *ICPP19*, 2019.

[38] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022.

[39] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[40] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599. IEEE, 2017.

[41] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proc. ACM Manag. Data*, 1(3):206:1–206:29, 2023.

[42] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–373, 2022.

[43] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: fast and efficient tensor compilation for deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 233–248. USENIX Association, 2022.