USENIX Association

Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)

July 14–16, 2021

ISBN 978-1-939133-22-9

Conference Organizers

Program Co-Chairs
Angela Demke Brown, *University of Toronto*
Jay Lorch, *Microsoft Research*

Program Committee
Atul Adya, *Google*
Deniz Altinbüken, Google
Nadav Amit, *VMware Research*
Thomas Anderson, *University of Washington*
Sebastian Angel, *University of Pennsylvania*
Behnaz Arzani, *Microsoft Research*
Mahesh Balakrishnan, *Facebook*
Sujata Banerjee, *VMware Research*
Sorav Bansal, *Indian Institute of Technology Delhi*
Andrew Baumann, *Microsoft Research*
Adam Belay, *Massachusetts Institute of Technology*
Theophilus A. Benson, *Brown University*
Pramod Bhatotia, *Technische Universität Munich*
James Bornholt, *The University of Texas at Austin*
Edouard Bugnion, *EPFL*
George Candea, *EPFL*
Kang Chen, *Tsinghua University*
Rong Chen, *Shanghai Jiao Tong University*
Mosharaf Chowdhury, *University of Michigan*
Moshe Gabel, *University of Toronto*
Ada Gavrilovska, *Georgia Institute of Technology*
Manya Ghobadi, *Massachusetts Institute of Technology*
Garth A. Gibson, *Vector Institute, Carnegie Mellon University, and University of Toronto*
Ashvin Goel, *University of Toronto*
Joseph Gonzalez, *University of California, Berkeley*
Ronghui Gu, *Columbia University*
Haryadi Gunawi, *University of Chicago*
Chuanxiong Guo, *ByteDance*
Chris Hawblitzel, *Microsoft Research*
Jon Howell, *VMware Research*
Yu Hua, *Huazhong University of Science and Technology*
Ryan Huang, *Johns Hopkins University*
Michael Isard, *Google Research*
Joe Izraelevitz, *University of Colorado, Boulder*
Manos Kapritsos, *University of Michigan*
Baris Kasikci, *University of Michigan*
Sam King, *University of California, Davis*
Orran Krieger, *Boston University*
Arvind Krishnamurthy, *University of Washington*
Amit Levy, *Princeton University*
Jialin Li, *National University of Singapore*
Wyatt Lloyd, *Princeton University*
Shan Lu, *University of Chicago*
Harsha V. Madhyastha, *University of Michigan*
Petros Maniatis, *Google*
Z. Morley Mao, *University of Michigan*
Changwoo Min, *Virginia Tech*
Radhika Mittal, *University of Illinois at Urbana–Champaign*
Dushyanth Narayanan, *Microsoft Research*
Ravi Netravali, *University of California, Los Angeles*
Kay Ousterhout, *Lightstep*
Aurojit Panda, *New York University*
Gennady Pekhimenko, *University of Toronto and Vector Institute*
Amar Phanishayee, *Microsoft Research*
Peter Pietzuch, *Imperial College London*
Don Porter, *The University of North Carolina at Chapel Hill*
Oriana Riva, *Microsoft Research*
Malte Schwarzkopf, *Brown University*
Vyas Sekar, *Carnegie Mellon University*
Michael Stumm, *University of Toronto*
Lalith Suresh, *VMware Research*
Doug Terry, *Amazon*
Alexey Tumanov, *Georgia Institute of Technology*
Amin Vahdat, *Google*
Shivaram Venkataraman, *University of Wisconsin—Madison*
Rashmi Vinayak, *Carnegie Mellon University*
Marko Vukolić, *IBM Research - Zurich*
Andrew Warfield, *Amazon*
Gala Yadgar, *Technion—Israel Institute of Technology*
Junfeng Yang, *Columbia University*
Ding Yuan, *University of Toronto*
Irene Zhang, *Microsoft Research*
Yiying Zhang, *University of California, San Diego*
Wenting Zheng, *University of California, Berkeley, and Carnegie Mellon University*
Yuanyuan Zhou, *University of California, San Diego*

Preview Session Co-Chairs
Sangeetha Abdu Jyothi, *University of California, Irvine, and VMware Research*
Deniz Altinbüken, *Google*
Dilma Da Silva, *Texas A&M University*
Aurojit Panda, *New York University*

Mentoring Co-Chairs
Baris Kasikci, *University of Michigan*
Amy Ousterhout, *University of California*, *Berkeley*
Malte Schwarzkopf, *Brown University*

Networking Session Co-Chairs
Reto Achermann, *University of British Columbia*
Zsolt István, *IT University of Copenhagen*
Adriana Szekeres, *VMware Research*
Vasily Tarasov, *IBM Research - Almaden*

Steering Committee
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Jason Flinn, *Facebook*
Casey Henderson, *USENIX Association*
Jon Howell, *VMware Research*
Kimberly Keeton
Hank Levy, *University of Washington*
Shan Lu, *University of Chicago*
James Mickens, *Harvard University*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Margo Seltzer, *University of British Columbia*
Geoff Voelker, *University of California, San Diego*

External Reviewers

Adam Chlipala
Aditya Akella
Adriana Szekeres
Ana Klimovic
Carmela Troncoso
Cristina Nita-Rotaru
Harry Xu
James Mickens
James R. Wilcox
Jonathan Mace
Kim Laine
Philip Levis
Rebecca Isaacs
Ryan Stutsman
Sarah Meiklejohn
Srinath Setty
Steven Hand
Ulfar Erlingsson

Message from the OSDI '21 Program Co-Chairs

Dear colleagues,

Welcome to the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)!

This is the oddest year ever for an OSDI, thanks to the laudable efforts of several members of our community to move OSDI to an annual rather than bi-annual cadence. We hope this experiment is successful and we can continue having this higher-bandwidth and lower-latency channel for sharing the excellent research done by the OSDI community.

This year's program offers 31 exceptional papers. These papers represent the many strengths of our community and cover a wide range of topics, including systems support for machine learning, memory management, file and storage systems, data management, operating systems, hardware, security, privacy, distributed systems, correctness, and formal verification of systems.

Given the exceptionally high number of submissions received by the previous OSDI, we recruited a large PC of 75 full members including academics, industrial researchers, and industrial practitioners. We also recruited 50 people to serve as an external review committee, providing a larger pool of expertise that we could draw on when needed; we called on 18 of them to provide additional expert reviews. We are grateful to all the committee members for agreeing to serve on relatively short notice after OSDI was added to the 2021 conference calendar in late Summer 2020.

Our program committee received 165 submissions and reviewed them in two rounds. Papers received three reviews in the first round; 110 advanced to round two, where they received an additional three reviews. For a small number of papers, where opinions were divided or where a paper was particularly specialized, we solicited additional expert reviews from our external review committee. In total, the PC and external reviewers wrote more than 770,000 words in more than 850 thoughtful reviews.

The PC conducted extensive discussions to select which papers to accept. This began with a rigorous asynchronous online discussion phase across the full PC, which resulted in 14 acceptances. The 41 papers that didn't reach an accept-or-reject consensus during the asynchronous online phase were discussed during a two-day PC meeting conducted via videoconference. The PC chairs strove to ensure that all papers received full and fair consideration. All discussions reached a consensus agreement, and a PC member wrote a summary of that discussion for the authors. Across all discussion stages, our reviewers wrote over 1,800 comments in HotCRP containing nearly 200,000 words. Ultimately, the PC selected 31 papers for presentation at the conference, resulting in a 19% acceptance rate, similar to prior years. Each of the accepted papers was allocated two additional pages and shepherded by a member of the PC to help the authors address the reviewers' comments in the camera-ready version.

After finalizing the program, we created a separate committee to decide the Jay Lepreau Best Paper Awards composed of PC members with no conflicts with the papers under consideration. PC members nominated papers for these awards. We selected seven papers with at least two nominations for best paper as candidates for the award. After reading the nominated papers and considering the reviews from the full PC, the awards committee chose the Jay Lepreau Best Paper Award recipients.

OSDI '21 featured an artifact-evaluation process organized by Artifact Evaluation Committee Co-Chairs Guyue (Grace) Liu, Manuel Rigger, and Lalith Suresh. Of the 31 papers accepted at OSDI '21, 26 had artifacts submitted by their authors, and all 26 of these earned the "Available" badge. In addition, 23 artifacts earned the "Functional" badge and 20 earned the most challenging "Results Reproduced" badge. For more details, see the Message from the OSDI '21 Artifact Evaluation Committee Co-Chairs.

As PC co-chairs, we stand on the shoulders of so many who did a tremendous amount of hard work to make OSDI '21 a success. First, we thank the authors of all submitted papers for choosing to send their work to OSDI. Thanks also to the program committee for their hard work in reviewing and discussing the submissions and in shepherding the accepted papers. We're also grateful to the external reviewers who provided additional perspectives. We thank the Artifact Evaluation Committee Co-Chairs mentioned in the previous paragraph as well as all the members of the Artifact Evaluation Committee who helped conduct thorough evaluations. We thank Baris Kasikci, Amy Ousterhout, and Malte Schwarzkopf for organizing OSDI/ATC mentoring; Deniz Altınbüken, Dilma Da Silva, Sangeetha Abdu Jyothi, and Aurojit Panda for organizing the joint OSDI/ATC preview session; Reto Achermann, Zsolt István, Adriana Szekeres, and Vasily Tarasov for organizing the joint OSDI/ATC networking session; and Irina Calciu and Geoff Kuenning, the Program Committee Co-Chairs of ATC '21, for coordinating with us efficiently, productively, and enjoyably. We thank the USENIX staff, who have been fundamental in organizing OSDI '21 in an especially difficult year. The logistics of the online PC meeting were facilitated by PhD students Christina Christodoulakis, Eric Munson, and Upamanyu Sharma, whose assistance we greatly appreciate. Finally, OSDI wouldn't be what it is without our attendees—thank you for listening to our speakers, asking challenging and insightful questions, sharing your ideas with others, and networking with one another online!

We hope you will find OSDI '21 interesting, educational, and inspiring!

Angela Demke Brown, *University of Toronto*
Jay Lorch, *Microsoft Research*
OSDI '21 Program Co-Chairs

Message from the OSDI '21 Artifact Evaluation Committee Co-Chairs

We are happy to report about the OSDI '21 artifact evaluation process. This is the second time that OSDI conducted such a process and we hope to keep improving it so that artifact evaluation will become more common in our community's conferences.

Process

We continued to use the three-badge approach (vs. the single-badge approach) from OSDI '20 to evaluation and these three badges include:

- **Artifacts Available:** To earn this badge, the AEC must judge that the artifacts associated with the paper have been made available for retrieval, permanently and publicly.
- **Artifacts Functional:** To earn this badge, the AEC must judge that the artifacts conform to the expectations set by the paper in terms of functionality, usability, and relevance.
- **Results Reproduced:** To earn this badge, the AEC must judge that they can use the submitted artifacts to obtain the main results presented in the paper.

Evaluation

We had 28 reviewers and we assigned 2 or 3 artifacts for each reviewer so that each artifact was evaluated by 3 reviewers. The evaluation process had two key phases: the kick-the-tires phase and the in-depth evaluation phase. During the kick-the-tires phase, reviewers made a quick first pass over all assignments to identify and report obvious problems and communicated them with the authors. After the kick-the-tires phase, reviewers evaluated each assignment thoroughly and wrote detailed reviews. Finally, reviewers coordinated and communicated with fellow AEC members and decided which badges should be awarded to each artifact.

Results

OSDI '21 accepted 31 papers and 26 papers participated in the AE, a significant increase in the participation ratio: 84%, compared to OSDI '20 (70%) and SOSP '19 (61%).

Of the 26 submitted artifacts:

- 26 artifacts received the Artifacts Available badge (100%).
- 23 artifacts received the Artifacts Functional badge (88%).
- 20 artifacts received the Results Reproduced badge (77%).

Key Takeaways

CloudLab Resources: Our experience showed that CloudLab (https://cloudlab.us/) can effectively facilitate the evaluation process. We suggest future AEC chairs prepare and make CloudLab resources available from the beginning of the evaluation process.

Usage of Screencasts: Some artifacts could only be evaluated based on the screencasts due to various constraints. This posed a few challenges around identifying a consistent standard for evaluating screencasts. We suggest future AEC chairs provide clear guidance on screencasts to authors and announce ahead of time whether they count for different badges.

Finally, we deeply thank the authors and the AEC committee for all their efforts in making the OSDI '21 artifact evaluation possible, especially during a pandemic.

Guyue (Grace) Liu, *Carnegie Mellon University*
Manuel Rigger, *ETH Zürich*
Lalith Suresh, *VMware Research*
OSDI '21 Artifact Evaluation Committee Co-chairs

15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)

July 14–16, 2021

Wednesday, July 14

Optimizations and Scheduling for Machine Learning

Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning 1
Aurick Qiao, *Petuum, Inc. and Carnegie Mellon University;* Sang Keun Choe and Suhas Jayaram Subramanya, *Carnegie Mellon University;* Willie Neiswanger, *Petuum, Inc. and Carnegie Mellon University;* Qirong Ho, *Petuum, Inc.;* Hao Zhang, *Petuum, Inc. and UC Berkeley;* Gregory R. Ganger, *Carnegie Mellon University;* Eric P. Xing, *MBZUAI, Petuum, Inc., and Carnegie Mellon University*

Oort: Efficient Federated Learning via Guided Participant Selection 19
Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury, *University of Michigan*

PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections 37
Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, and Liyan Zheng, *Tsinghua University;* Yuanzhi Li, *Carnegie Mellon University;* Kaiyuan Rong and Yuanyong Chen, *Tsinghua University;* Zhihao Jia, *Carnegie Mellon University and Facebook*

Privacy Budget Scheduling 55
Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, and Roxana Geambasu, *Columbia University;* Mathias Lécuyer, *Microsoft Research*

Storage

Modernizing File System through In-Storage Indexing 75
Jinhyung Koo, Junsu Im, Jooyoung Song, and Juhyung Park, *DGIST;* Eunji Lee, *Soongsil University;* Bryan S. Kim, *Syracuse University;* Sungjin Lee, *DGIST*

Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes 93
Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu, *Tsinghua University*

Rearchitecting Linux Storage Stack for μs Latency and High Throughput 113
Jaehyun Hwang and Midhul Vuppalapati, *Cornell University;* Simon Peter, *UT Austin;* Rachit Agarwal, *Cornell University*

Optimizing Storage Performance with Calibrated Interrupts 129
Amy Tai, *VMware Research;* Igor Smolyar, *Technion – Israel Institute of Technology;* Michael Wei, *VMware Research;* Dan Tsafrir, *Technion – Israel Institute of Technology and VMware Research*

ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction 147
Kyuhwa Han, *Sungkyunkwan University and Samsung Electronics;* Hyunho Gwak and Dongkun Shin, *Sungkyunkwan University;* Joo-Young Hwang, *Samsung Electronics*

Data Management

DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling 163
Tanvir Ahmed Khan and Ian Neal, *University of Michigan;* Gilles Pokam, *Intel Corporation;* Barzan Mozafari and Baris Kasikci, *University of Michigan*

CLP: Efficient and Scalable Search on Compressed Text Logs 183
Kirk Rodrigues, Yu Luo, and Ding Yuan, *University of Toronto and YScope Inc.*

Polyjuice: High-Performance Transactions via Learned Concurrency Control 199
Jiachen Wang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Ding Ding, *Department of Computer Science, New York University;* Huan Wang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Conrad Christensen, *Department of Computer Science, New York University;* Zhaoguo Wang and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Jinyang Li, *Department of Computer Science, New York University*

Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing 219
Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai Artificial Intelligence Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Thursday, July 15

Operating Systems and Hardware

The nanoPU: A Nanosecond Network Stack for Datacenters 239
Stephen Ibanez, Alex Mallery, Serhat Arslan, and Theo Jepsen, *Stanford University;* Muhammad Shahbaz, *Purdue University;* Changhoon Kim and Nick McKeown, *Stanford University*

Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator 257
A.H. Hunter, *Jane Street Capital;* Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan, *Google*

Scalable Memory Protection in the PENGLAI Enclave 275
Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, and Xueqiang Jiang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Yubin Xia, Binyu Zang, and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

NrOS: Effective Replication and Sharing in an Operating System 295
Ankit Bhardwaj and Chinmay Kulkarni, *University of Utah;* Reto Achermann, *University of British Columbia;* Irina Calciu, *VMware Research;* Sanidhya Kashyap, *EPFL;* Ryan Stutsman, *University of Utah;* Amy Tai and Gerd Zellweger, *VMware Research*

Security and Privacy

Addra: Metadata-private voice communication over fully untrusted infrastructure 313
Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta, *University of California, Santa Barbara*

Bringing Decentralized Search to Decentralized Services 331
Mingyu Li, Jinhao Zhu, and Tianxu Zhang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Cheng Tan, *Northeastern University;* Yubin Xia, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Sebastian Angel, *University of Pennsylvania;* Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing 349
Youngseok Yang, *Seoul National University;* Taesoo Kim, *Georgia Institute of Technology;* Byung-Gon Chun, *Seoul National University and FriendliAI*

MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation 367
Sam Kumar, David E. Culler, and Raluca Ada Popa, *University of California, Berkeley*

Zeph: Cryptographic Enforcement of End-to-End Data Privacy 387
Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi, *ETH Zürich*

Friday, July 16

Correctness

DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols 405
Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan, *Columbia University*

GoJournal: a verified, concurrent, crash-safe journaling system 423
Tej Chajed, *MIT CSAIL;* Joseph Tassarotti, *Boston College;* Mark Theng, *MIT CSAIL;* Ralf Jung, *MPI-SWS;* M. Frans Kaashoek and Nickolai Zeldovich, *MIT CSAIL*

Storm: Refinement Types for Secure Web Applications 441
Nico Lehmann and Rose Kunkel, *UC San Diego;* Jordan Brown, *Independent;* Jean Yang, *Akita Software;* Niki Vazou, *IMDEA Software Institute;* Nadia Polikarpova, Deian Stefan, and Ranjit Jhala, *UC San Diego*

Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation 461
Shaghayegh Mardani, *UCLA;* Ayush Goel, *University of Michigan;* Ronny Ko, *Harvard University;* Harsha V. Madhyastha, *University of Michigan;* Ravi Netravali, *Princeton University*

SanRazor: Reducing Redundant Sanitizer Checks in C/C++ Programs 479
Jiang Zhang, *University of Southern California;* Shuai Wang, *HKUST;* Manuel Rigger, Pinjia He, and Zhendong Su, *ETH Zurich*

Graph Embeddings and Neural Networks

Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. . . . 495
John Thorpe, Yifan Qiao, Jonathan Eyolfson, and Shen Teng, *UCLA;* Guanzhou Hu, *UCLA and University of Wisconsin, Madison;* Zhihao Jia, *CMU;* Jinliang Wei, *Google Brain;* Keval Vora, *Simon Fraser;* Ravi Netravali, *Princeton University;* Miryung Kim and Guoqing Harry Xu, *UCLA*

GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. 515
Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding, *University of California, Santa Barbara*

Marius: Learning Massive Graph Embeddings on a Single Machine 533
Jason Mohoney and Roger Waleffe, *University of Wisconsin-Madison;* Henry Xu, *University of Maryland, College Park;* Theodoros Rekatsinas and Shivaram Venkataraman, *University of Wisconsin-Madison*

***P*^{3}: Distributed Deep Graph Learning at Scale 551**
Swapnil Gandhi and Anand Padmanabha Iyer, *Microsoft Research*

Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning

Aurick Qiao[1,2] Sang Keun Choe[2] Suhas Jayaram Subramanya[2] Willie Neiswanger[1,2]
Qirong Ho[1] Hao Zhang[1,3] Gregory R. Ganger[2] Eric P. Xing[4,1,2]

[1]*Petuum, Inc.* [2]*Carnegie Mellon University* [3]*UC Berkeley* [4]*MBZUAI*

Abstract

Pollux improves scheduling performance in deep learning (DL) clusters by adaptively co-optimizing inter-dependent factors both at the per-job level and at the cluster-wide level. Most existing schedulers expect users to specify the number of resources for each job, often leading to inefficient resource use. Some recent schedulers choose job resources for users, but do so without awareness of how DL training can be re-optimized to better utilize the provided resources.

Pollux simultaneously considers both aspects. By monitoring the status of each job during training, Pollux models how their *goodput* (a metric we introduce to combine system throughput with statistical efficiency) would change by adding or removing resources. Pollux dynamically (re-)assigns resources to improve cluster-wide goodput, while respecting fairness and continually optimizing each DL job to better utilize those resources.

In experiments with real DL jobs and with trace-driven simulations, Pollux reduces average job completion times by 37–50% relative to state-of-the-art DL schedulers, even when they are provided with ideal resource and training configurations for every job. Pollux promotes fairness among DL jobs competing for resources, based on a more meaningful measure of *useful* job progress, and reveals a new opportunity for reducing DL cost in cloud environments. Pollux is implemented and publicly available as part of an open-source project at `https://github.com/petuum/adaptdl`.

1 Introduction

Deep learning (DL) training has rapidly become a dominant workload in many shared resource environments such as datacenters and the cloud. DL jobs are resource-intensive and long-running, often demanding distributed execution using expensive hardware devices (eg. GPUs or TPUs) in order to complete within reasonable amounts of time. To meet this resource demand, dedicated clusters are often provisioned for deep learning [31, 67], with a scheduler that mediates resource sharing between many competing DL jobs.

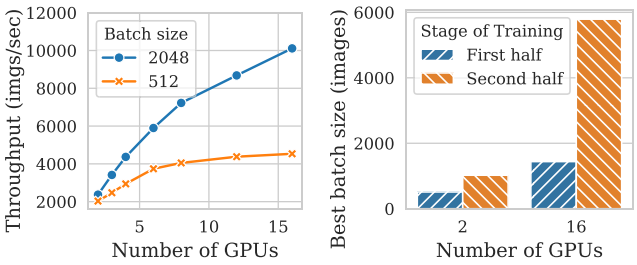
Existing schedulers require users to manually configure their jobs, which if done improperly, can greatly degrade training performance and resource efficiency. For example, allocating too many GPUs may result in long queuing times and inefficient resource usage, while allocating too few GPUs may result in long runtimes and unused resources. Such decisions are especially difficult to make in a shared-cluster setting, since optimal choices are dynamic and depend on the cluster load while a job is running.

Even though recent *elastic* schedulers can automatically select an appropriate amount of resources for each job, they do so blindly to inter-dependent training-related configurations that are just as important. For example, the *batch size* and *learning rate* of a DL job influence the amount of computation needed to train its model. Their optimal choices vary between different DL tasks and model architectures, and they have strong dependence on the job's allocation of resources.

The amount of resources, batch size, and learning rate are difficult to configure appropriately without expert knowledge about both the cluster hardware performance and DL model architecture. Due to the inter-dependence between their optimal values, they should be configured jointly with each other. Due to the dynamic nature of shared clusters, their optimal values may change over time. This creates a complex web of considerations a user must make in order to configure their job for efficient execution and resource utilization.

How can a cluster scheduler help to automatically configure user-submitted DL jobs? Fundamentally, a properly-configured DL job strikes a balance between two often opposing desires: (1) *system throughput*, the number of training examples processed per wall-clock time, and (2) *statistical efficiency*, the amount of progress made per training example processed.

System throughput can be increased by increasing the batch size, as illustrated in Fig. 1a. A larger batch size enables higher utilization of more compute resources (e.g., more GPUs). But, even with an optimally-retuned learning rate, increasing the batch size often results in a decreased statistical efficiency [46, 57]. For every distinct allocation of GPUs, there is potentially a different batch size that best balances increasing



(a) Job scalability (and thus resource utilization) depends on the batch size.

(b) The most efficient batch size depends on the allocated resources and stage of training.

Figure 1: Trade-offs between the batch size, resource scalability, and stage of training (ResNet18 on CIFAR-10). The learning rate is separately tuned for each batch size.

system throughput with decreasing statistical efficiency, as illustrated in Fig. 1b. Furthermore, how quickly the statistical efficiency decreases with respect to the batch size depends on the current training progress. A job in a later stage of training can potentially tolerate 10x or larger batch sizes without degrading statistical efficiency, than earlier during training [46].

Guided by these insights, this paper presents *Pollux*, a hybrid resource scheduler that *co-adaptively* allocates resources and tunes the batch size and learning rate for all DL jobs in a shared cluster. Pollux achieves this by jointly managing several system-level and training-related parameters, including the number of GPUs, co-location of workers, per-GPU batch size, gradient accumulation, and learning rate scaling. In particular:

★ We propose a formulation of *goodput* for DL jobs, which is a holistic measure of training performance that takes into account both system throughput and statistical efficiency.

★ We show that a model of a DL job's goodput can be learned by observing its throughput and statistical behavior during training, and used for predicting the performance given different resource allocations and batch sizes.

★ We design and implement a scheduling architecture that uses such models to configure the right combination of resource allocation and training parameters for each pending and running DL job. This includes locally tuning system-level and training-related parameters for each DL job, and globally optimizing cluster-wide resource allocations. The local and global components actively communicate and cooperate with each other, operating based on the common goal of goodput maximization.

★ We evaluate Pollux on a cluster testbed using a workload derived from a Microsoft cluster trace. Compared with recent DL schedulers, Tiresias [22] and Optimus [52], Pollux reduces the average job completion time by up to 73%. Even when all jobs are manually tuned beforehand, Pollux reduces the average job completion time by 37%–50%. At the same time, Pollux improves finish-time fairness [43] by 1.5×–5.4×.

★ We show that, in cloud environments, using goodput-driven auto-scaling based on Pollux can potentially reduce the cost of training large models by 25%.

2 Background: Distributed DL Training

Training a deep learning model typically involves minimizing a *loss function* of the form

$$\mathcal{L}(w) = \frac{1}{|X|}\sum_{x_i \in X} \ell(w, x_i), \tag{1}$$

where $w \in \mathbb{R}^d$ are the model parameters to be optimized, X is the training dataset, x_i is an individual sample in X, and ℓ is the loss evaluated at a single sample.

The loss function can be minimized using stochastic gradient descent (SGD) or its variants like AdaGrad [15] and Adam [36]. For the purpose of explaining system throughput and statistical efficiency, we will use SGD as the running example. SGD repeatedly applies the following update until the loss converges to a stable value: $w^{(t+1)} = w^{(t)} - \eta \hat{g}^{(t)}$. η is known as the learning rate, which is a scalar that controls the magnitude of each update, and $\hat{g}^{(t)}$ is a stochastic gradient estimate of the loss function $\mathcal{L}$, evaluated using a random *mini-batch* $\mathcal{M}^{(t)} \subset X$ of the training data:

$$\hat{g}^{(t)} = \frac{1}{M}\sum_{x_i \in \mathcal{M}^{(t)}} \nabla \ell(w^{(t)}, x_i). \tag{2}$$

The learning rate η and batch size $M = |\mathcal{M}^{(t)}|$ are training parameters which are typically chosen by the user.

2.1 System Throughput

The *system throughput* of DL training can be defined as the number of training samples processed per unit of wall-clock time. When a DL job is distributed across several nodes, its system throughput is determined by several factors, including (1) the allocation and placement of resources (e.g. GPUs) assigned to the job, (2) the method of distributed execution and synchronization, and (3) the batch size.

Data-parallel execution. *Synchronous data-parallelism* is a popular method of distributed execution for DL training. The model parameters $w^{(t)}$ are replicated across a set of distributed GPUs $1, \ldots, K$, and each mini-batch $\mathcal{M}^{(t)}$ is divided into equal-sized partitions per node, $\mathcal{M}_1^{(t)}, \ldots, \mathcal{M}_K^{(t)}$. Each GPU k computes a local gradient estimate $\hat{g}_k^{(t)}$ using its own partition:

$$\hat{g}_k^{(t)} = \frac{1}{m}\sum_{x_i \in \mathcal{M}_k^{(t)}} \nabla \ell(w^{(t)}, x_i), \tag{3}$$

where $m = |\mathcal{M}_k^{(t)}|$ is the per-GPU batch size. These local gradient estimates are then averaged across all GPUs to obtain the desired $\hat{g}^{(t)}$. Finally, each node applies the same update using $\hat{g}^{(t)}$ to obtain the new model parameters $w^{(t+1)}$.

The run-time of each training iteration is determined by two main components. *First*, the time spent computing each $\hat{g}_k^{(t)}$, which we denote by T_{grad}. *Second*, the time spent

averaging $\hat{g}_k^{(t)}$ (e.g. using collective all-reduce [51, 56]) and/or synchronizing $w^{(t)}$ (e.g. using parameter servers [8, 11, 26, 53]) across all GPUs, which we denote by T_{sync}. T_{sync} is influenced by the size of the gradients, performance of the network, and is typically shorter when the GPUs are co-located within the same physical node or rack.

Limitations due to the batch size. When the number of GPUs is increased, T_{grad} decreases due to a smaller per-GPU batch size. On the other hand, T_{sync}, which is typically independent of the batch size, remains unchanged. By Amdahl's Law, no matter how many GPUs are used, the run-time of each training iteration is lower bounded by T_{sync}. To overcome this scalability limitation, a common strategy is to increase the batch size. Doing so causes the local gradient estimates to be computed over more training examples and thereby increasing the ratio of T_{grad} to T_{sync}. As a result, using a larger batch size enables higher system throughput when scaling to more GPUs in the synchronous data-parallel setting.

2.2 Statistical Efficiency

The *statistical efficiency* of DL training can be defined as the amount of training progress made per unit of training data processed, influenced by parameters such as *batch size* or *learning rate*; for example, a larger batch size normally decreases the statistical efficiency. The ability to predict statistical efficiency is key to improving said statistical efficiency, because we can use the predictions to better adapt the batch sizes and learning rates.

Gradient noise scale. Previous work [32, 46] relate the statistical efficiency of DL training to the *gradient noise scale* (GNS), which measures the noise-to-signal ratio of the stochastic gradient. A larger GNS means that training parameters such as the batch size and learning rate can be increased to higher values with relatively less reduction of the statistical efficiency. The GNS can vary greatly between different DL models [19]. It is also non-constant and tends to gradually increase during training, by up to $10\times$ or more [46]. Thus, it is possible to attain significantly better statistical efficiency for large batch sizes later on during training.

The gradient noise scale mathematically captures an intuitive explanation of how the batch size affects statistical efficiency. When the stochastic gradient has low noise, adding more training examples to each mini-batch does not significantly improve each gradient estimate, which lowers statistical efficiency. When the stochastic gradient has high noise, adding more training examples to each mini-batch reduces the noise of each gradient estimate, which maintains high statistical efficiency. Near convergence, the stochastic gradients have relatively lower signal than noise, and so larger batch sizes can be more useful later in training.

Learning rate scaling. When training with an increased total batch size M, the learning rate η should also be increased, otherwise the final trained model quality/accuracy can be significantly worse [57]. How to increase the learning rate varies between different models and training algorithms (e.g. SGD, Adam [36], AdamW [42]), and several well-established scaling rules may be used. For example, the linear scaling rule [21], which prescribes that η be scaled proportionally with M, or the square-root scaling rule [40, 69] (commonly used with Adam), which prescribes that η be scaled proportionally with $\sqrt{M}$. More recent scaling rules such as AdaScale [32] may scale the learning rate adaptively during training.

In addition to decreasing statistical efficiency, using large batch sizes may also degrade the final model quality in terms of validation performance [19, 35, 60], although the reasons behind this effect are not completely understood at the time of this paper. However, for each of the learning rate scaling rules mentioned above, there is usually a problem-dependent range of batch sizes that achieve similar validation performances. Within these ranges, the batch size may be chosen more freely without significantly degrading the final model quality.

2.3 Existing DL Schedulers

We broadly group existing DL schedulers into two categories, to put Pollux in context. First, *non-scale-adaptive* schedulers are agnostic to the performance scalability of DL jobs with respect to the amount of allocated resources. For example, Tiresias [22] requires users to specify the number of GPUs at the time of job submission, which will be fixed for the lifetime of the job. Gandiva [66] also requires users to specify number of GPUs, but enhances resource utilization through fine-grained time sharing and job packing. Although Gandiva may dynamically change the number of GPUs used by a job, it does so opportunistically and not based on knowledge of job scalability.

Second, *scale-adaptive* schedulers automatically decide the amount of resources allocated to each job based on how well they can be utilized to speed up the job. For example, Optimus [52] learns a predictive model for the system throughput of each job given various amounts of resources, and optimizes cluster-wide resource allocations to minimize the average job completion time. SLAQ [71], which was not evaluated on DL, uses a similar technique to minimize the average loss values for training general ML models. Gavel [48] goes further by scheduling based on a throughput metric that is comparable across different accelerator types.[1] AntMan [67] uses dynamic scaling and fine-grained GPU sharing to improve cluster utilization, resource fairness, and job completion times. Themis [43] introduces the notion of finish-time fairness, and promotes fairness between multiple DL applications with a two-level scheduling architecture.

Crucially, existing schedulers are agnostic to the statistical efficiency of DL training and the inter-dependence of resource decisions and training parameters. Pollux explicitly co-adapts these inter-dependent values to improve goodput for DL jobs.

[1] Pollux's current throughput model does not consider accelerator heterogeneity. We believe that extending with Gavel's metric would allow Pollux to co-adapt for goodput in heterogeneous DL clusters.

3 The Goodput of DL Training and Pollux

In this section, we define the *goodput*[2] of DL jobs, which is a measure of training performance that takes into account both system throughput and statistical efficiency. We then describe how the goodput can be measured during training and used as a predictive model, which is leveraged by Pollux to jointly optimize cluster-wide resource allocations and batch sizes.

Definition 3.1. (Goodput) The *goodput* of a DL training job at iteration t is the product between its system throughput and its statistical efficiency at iteration t,

$$\texttt{GOODPUT}_t(\star) = \texttt{THROUGHPUT}(\star) \times \texttt{EFFICIENCY}_t(M(\star)), \quad (4)$$

where $\star$ represents any configuration parameters that jointly influence the throughput and batch size during training, and M is the total batch size summed across all allocated GPUs.

While the above definition is general across many training systems, we focus on three configuration parameters of particular impact in the context of efficient resource scheduling, i.e. $\star = (a,m,s)$, where:

- $a \in \mathbb{Z}^N$: the *allocation vector*, where a_n is the number of GPUs allocated from node n.
- $m \in \mathbb{Z}$: the *per-GPU batch size*.
- $s \in \mathbb{Z}$: number of *gradient accumulation steps* (§3.2).

The total batch size is then defined as

$$M(a,m,s) = \texttt{SUM}(a) \times m \times (s+1).$$

Pollux's approach. An initial batch size M_0 and learning rate (LR) η_0 are selected by the user when submitting their job. Pollux will start each job using a single GPU, $m = M = M_0$, $s = 0$, and $\eta = \eta_0$. As the job runs, Pollux profiles its execution to learn and refine predictive models for both THROUGHPUT (§3.2) and EFFICIENCY (§3.1). Using these predictive models, Pollux periodically re-tunes (a,m,s) for each job, according to cluster-wide resource availability and performance (§4.2).

$\texttt{EFFICIENCY}_t$ is measured *relative to* the initial batch size M_0 and learning rate η_0, and Pollux only considers batch sizes that are at least the initial batch size, ie. $M \geq M_0$. In this scenario, $\texttt{EFFICIENCY}_t(M)$ is a fraction (between 0 and 1) relative to $\texttt{EFFICIENCY}_t(M_0)$. Therefore, goodput can be interpreted as the portion of the throughput that is useful for training progress, being equal to the throughput if and only if perfect statistical efficiency is achieved.

Plug-in Learning Rate Scaling. Recall from §2.2 that different training jobs may require different learning rate scaling rules to adjust η in response to changes in M. In order to support a wide variety of LR scaling rules, including state-of-the-art rules such as AdaScale [32], Pollux provides a plug-in interface that can be implemented using a function signature

$$\texttt{SCALE\_LR}(M_0, M) \longrightarrow \lambda.$$

SCALE_LR is called before every model update step, and λ is used by Pollux to scale the learning rate. The implementation of SCALE_LR can utilize metrics collected during training, such as the gradient noise scale. Using this interface, one can implement rules including AdaScale, square-root scaling [40], linear scaling [21] and LEGW [69].

3.1 Modeling Statistical Efficiency

We model $\texttt{EFFICIENCY}_t(M)$ as the amount of progress made per training example using M, relative to using M_0. For SGD-based training, this quantity can be expressed in terms of the gradient noise scale (GNS) [46]. To support popular adaptive variants of SGD like Adam [36] and AdaGrad [64], we use the *pre-conditioned gradient noise scale* (PGNS), derived by closely following the original derivation of the GNS ("simple" noise scale in [46]) starting from pre-conditioned SGD[3] rather than vanilla SGD. The PGNS, which we denote by φ_t, is expressed as

$$\varphi_t = \frac{\mathrm{tr}(P\Sigma P^T)}{|Pg|^2}, \quad (5)$$

where g is the true gradient, P is the pre-conditioning matrix of the adaptive SGD algorithm, and Σ is the covariance matrix of per-example stochastic gradients. The PGNS is a generalization of the GNS and is mathematically equivalent to the GNS for the special case of vanilla SGD.

Similar to the GNS (Appendix D of [46]), it takes $1+\varphi_t/M$ training iterations to make a similar amount of training progress across different batch sizes M. Therefore, we can use the PGNS φ_t to define a concrete expression for $\texttt{EFFICIENCY}_t(M)$ as

$$\texttt{EFFICIENCY}_t(M) = \frac{\varphi_t + M_0}{\varphi_t + M}. \quad (6)$$

Intuitively, Eqn. 6 measures the contribution from each training example to the overall progress. If $\texttt{EFFICIENCY}_t(M) = E$, then (1) $0 < E \leq 1$, and (2) training using batch size M will need to process $1/E$ times as many training examples to make the same progress as using batch size M_0.

During training, Pollux estimates the value of φ_t, then uses Eqn 6 to predict the $\texttt{EFFICIENCY}_t$ at different batch sizes. The measured value of φ_t varies according to the training progress at iteration t, thus $\texttt{EFFICIENCY}_t(M)$ reflects the lifetime-dependent trends exhibited by the true statistical efficiency.

[2] Our notion of goodput for DL is analogous to the traditional definition of goodput in computer networks, ie. the *useful* portion of throughput as benchmarked by training progress per unit of wall-clock time.

[3] Pre-conditioned SGD optimizes $\mathcal{L}(Pw)$ instead of $\mathcal{L}(w)$, where P is known as a pre-conditioning matrix. Adaptive variants of SGD such as Adam and AdaGrad may be viewed as vanilla SGD (with momentum) applied together with a particular pre-conditioning matrix P.

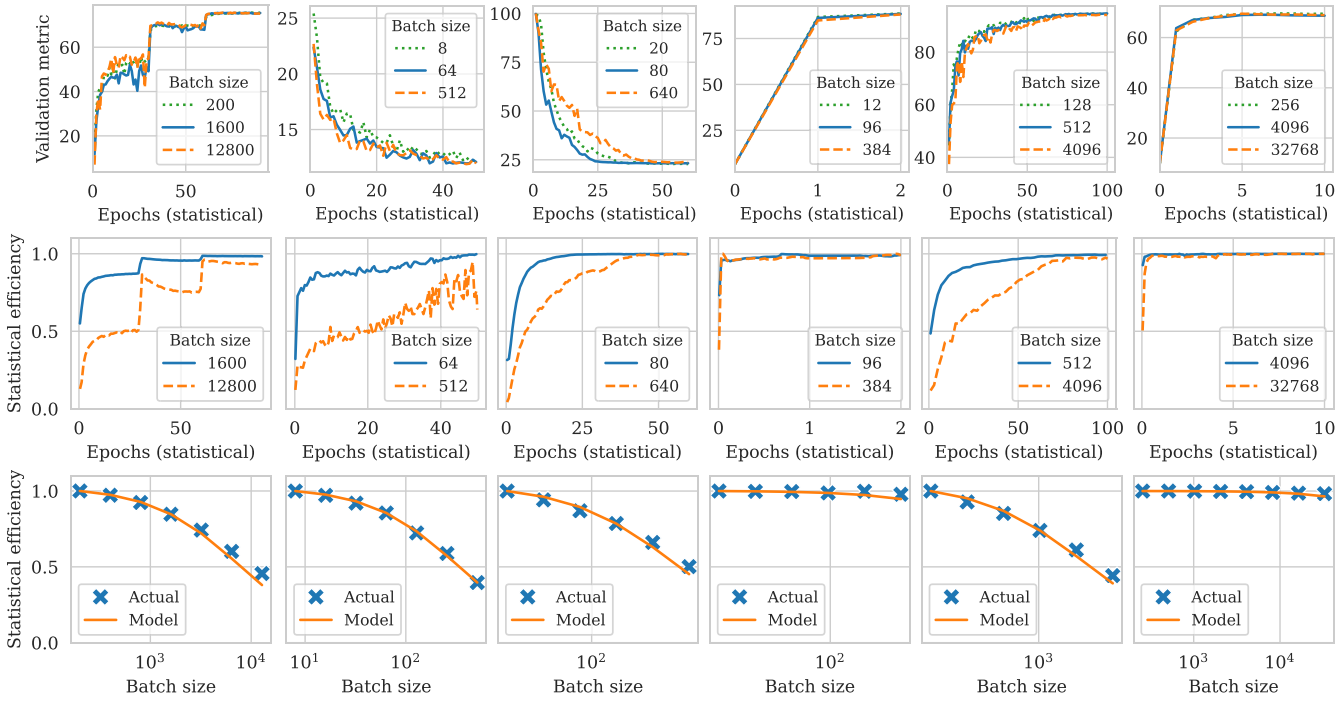


Figure 2: Statistical efficiency for all models described in Table 1. TOP: validation metric vs training progress for three different batch sizes: M_0, an intermediate batch size, and the max batch size limit we set for each DL task. Metrics are as defined in Table 1 except for YoloV3 for which validation loss is shown. MIDDLE: measured statistical efficiency vs. training progress for two different batch sizes. Training progress (x-axis) in the top two rows is shown in terms of "statistical epochs", defined as $\frac{M}{|X|}\sum_t \texttt{EFFICIENCY}_t(M)$ where $|X|$ is the size of the training dataset. BOTTOM: measured $\texttt{EFFICIENCY}_t$ vs. predicted $\texttt{EFFICIENCY}_t$ for a range of batch sizes (log-scaled), using φ_t measured using the median batch size from each range, during an early-training epoch (roughly 1/8th of the way through training).

Fig. 2 (TOP) shows the validation metrics on a held-out dataset for a variety of DL training tasks (details in Table 1) versus their training progress. "Statistical epochs"[4] is the number of training iterations normalized by $\texttt{EFFICIENCY}_t$ so that each statistical epoch makes theoretically, as projected by our model, the same training progress across different batch sizes. Thus, the degree of similarity between validation curves at different batch sizes is an indicator for the accuracy of $\texttt{EFFICIENCY}_t$ as a predictor of actual training progress.

Although there are differences in the validation curves for several DL tasks (especially in earlier epochs), they achieve similar best values across the different batch sizes we evaluated ($\pm 1\%$ relative difference for all tasks except DeepSpeech2 at $\pm 4\%$). We note that these margins are within the plateau of high-quality models expected from large-batch training [45].

Fig. 2 (MIDDLE and BOTTOM) show the measured and predicted $\texttt{EFFICIENCY}_t$ during training and for a range of different batch sizes. In general, larger batch sizes have lower $\texttt{EFFICIENCY}_t$ early in training, but close the gap later on in training. The exceptions being BERT, which is a fine-tuning task starting from an already pre-trained model, and recommendation, which uses a much smaller and shallower model architecture than the others. How $\texttt{EFFICIENCY}_t$ changes during training varies from task to task, and depends on specific properties like the learning rate schedule. For example, $\texttt{EFFICIENCY}_t$ for ImageNet, which uses step-based learning rate annealing, experiences sharp increases whenever the learning rate is annealed.

Finally, we note that the $\texttt{EFFICIENCY}_t$ function (which is supplied with estimates of φ_t by Pollux) is able to accurately model observed values at a range of different batch sizes. This means that φ_t measured using batch size M can be used by Pollux to predict the value of $\texttt{EFFICIENCY}_t$ at a different batch size M' without needing to train using M' ahead of time.

Upper batch size limit. In some cases, as the batch size increases, the chosen LR scaling rule may break down before the statistical efficiency decreases, which degrades the final model quality. To address these cases, the application may define a maximum batch size limit that will be respected by Pollux. Nevertheless, we find that a batch size up to $32\times$ larger works

[4] Similar to the notion of "scale-invariant iterations" defined in [32].

well in most cases. Furthermore, limits for common models are well-studied for popular LR scaling rules [21, 32, 57, 69]. As better LR scaling rules are developed, they may be incorporated into Pollux using its plug-in interface (§3).

Estimating φ_t. The PGNS φ_t can be estimated in a similar fashion as the GNS by following Appendix A.1 of [46], except using the pre-conditioned gradient Pg instead of the gradient g. This can be done efficiently when there are multiple data-parallel processes by using the different values of $\hat{g}_k^{(t)}$ already available on each GPU k. However, this method doesn't work when there is only a single GPU (and gradient accumulation is off, i.e. $s = 0$). In this particular situation, Pollux switches to a differenced variance estimator [63] which uses consecutive gradient estimates $\hat{g}^{(t-1)}$ and $\hat{g}^{(t)}$.

3.2 Modeling System Throughput

To model and predict the system throughput for data-parallel DL, we aim to predict the time spent per training iteration, T_{iter}, and then calculate the throughput as

$$\texttt{THROUGHPUT}(a,m,s) = M(a,m,s)/T_{iter}(a,m,s). \quad (7)$$

We start by separately modeling T_{grad}, the time in each iteration spent computing local gradient estimates, and T_{sync}, the time in each iteration spent averaging gradient estimates and synchronizing model parameters across all GPUs. We also start by assuming no gradient accumulation, i.e. $s = 0$.

Modeling T_{grad}. The local gradient estimates are computed using back-propagation, whose run-time scales linearly with the per-GPU batch size m. Thus, we model T_{grad} as

$$T_{grad}(m) = \alpha_{grad} + \beta_{grad} \cdot m, \quad (8)$$

where $\alpha_{grad}, \beta_{grad}$ are fittable parameters.

Modeling T_{sync}. When allocated a single GPU, no synchronization is needed and $T_{sync} = 0$. Otherwise, we model T_{sync} as a linear function of the number of GPUs since in data-parallelism, the amount of data sent and received from each replica is typically only dependent on the size of the gradients and/or parameters. We include a linear factor to account for performance retrogressions associated with using three or more GPUs, such as increasing likelihood of stragglers or network delays.

Co-location of GPUs on the same node reduces network communication, which can improve T_{sync}. Thus, we use different parameters depending on GPU placement. Letting $K = \texttt{SUM}(a)$ be the number of allocated GPUs,

$$T_{sync}(a,m) = \begin{cases} 0 & \text{if } K = 1 \\ \alpha_{sync}^{local} + \beta_{sync}^{local} \cdot (K-2) & \text{if } N = 1, K \geq 2 \\ \alpha_{sync}^{node} + \beta_{sync}^{node} \cdot (K-2) & \text{otherwise,} \end{cases} \quad (9)$$

where N is the number of physical nodes occupied by at least one replica. α_{sync}^{local} and β_{sync}^{local} are the constant and retrogression parameters for when all processes are co-located onto the same node. α_{sync}^{node} and β_{sync}^{node} are the analogous parameters for when at least two process are located on different nodes. Note that our model for T_{sync} can be extended to account for rack-level locality by adding a third pair of parameters.

Combining T_{grad} and T_{sync}. Modern DL frameworks can partially overlap T_{grad} and T_{sync} by overlapping gradient computation with network communication [70]. The degree of this overlap depends on structures in the specific DL model being trained, like the ordering and sizes of its layers.

Assuming no overlap, then $T_{iter} = T_{grad} + T_{sync}$. Assuming perfect overlap, then $T_{iter} = \max(T_{grad}, T_{sync})$. A realistic value of T_{iter} is somewhere in between these two extremes. To capture the overlap between T_{grad} and T_{sync}, we model T_{iter} as

$$T_{iter}(a,m,0) = \left(T_{grad}(a,m)^\gamma + T_{sync}(a)^\gamma\right)^{1/\gamma}, \quad (10)$$

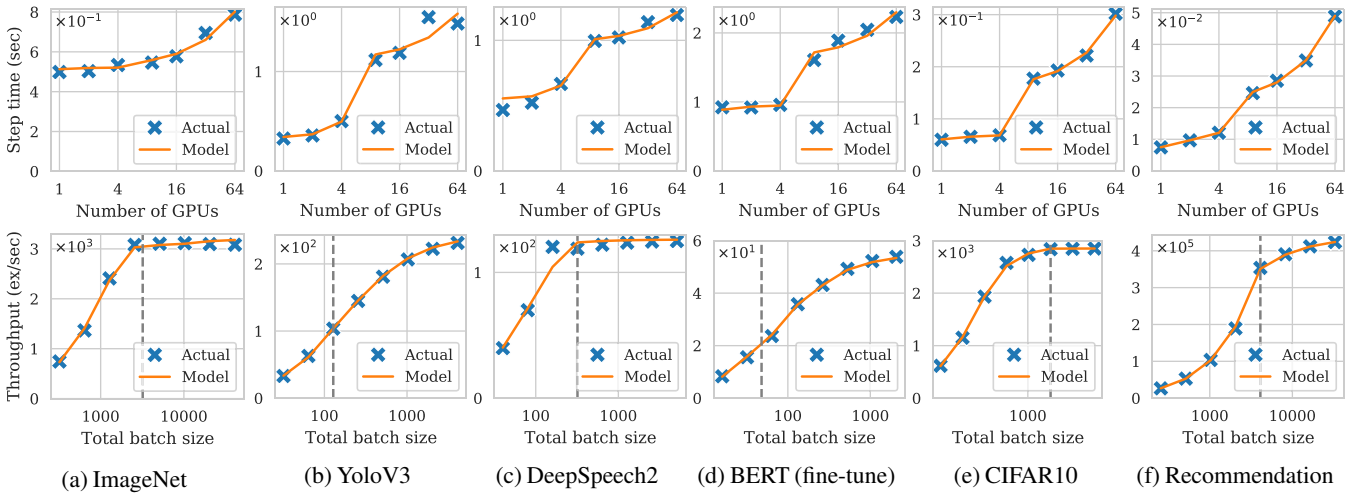
where $\gamma \geq 1$ is a learnable parameter. Eqn. 10 has the property that $T_{iter} = T_{grad} + T_{sync}$ when $\gamma = 1$, and smoothly transitions towards $T_{iter} = \max(T_{grad}, T_{sync})$ as $\gamma \to \infty$.

Gradient Accumulation. In data-parallelism, GPU memory limits the per-GPU batch size, and many DL models hit this limit before the batch size is large enough for T_{grad} to overcome T_{sync} (or experience diminishing statistical efficiency), resulting in suboptimal scalability. Several techniques exist for overcoming the GPU memory limit [9, 10, 27, 30]; we focus on gradient accumulation, which is easily implemented using popular DL frameworks. Per-GPU gradients are aggregated locally over s forward-backward passes before being synchronized across all GPUs during the $(s+1)^{\text{th}}$ pass, achieving a larger total batch size. Thus, one iteration of SGD spans s accumulation steps followed by one synchronization step, modeled as

$$T_{iter}(a,m,s) = s \times T_{grad}(a,m) + \left(T_{grad}(a,m)^\gamma + T_{sync}(a)^\gamma\right)^{1/\gamma}. \quad (11)$$

Throughput model validation. Fig. 3 shows an example of our THROUGHPUT function fit to measured throughput values for a range of resource allocations and batch sizes. Each DL task was implemented using PyTorch [51], which overlaps the backward pass' computation and communication. Gradients are synchronized with NCCL 2.7.8, which uses either ring all-reduce or tree all-reduce depending on the detected GPUs and their placements and its own internal performance estimates. Overall, we find that our model can represent the observed data closely, while varying both the amount of resources as well as the batch size. In particular, all models we measured except ImageNet exhibited high sensitivity to inter-node synchronization, indicating that they benefit from co-location of GPUs. Furthermore, YOLOv3 and BERT benefit from using gradient accumulation to increase their total batch sizes. These detailed characteristics are well-represented by our THROUGHPUT function, and can be optimized for by Pollux.

In addition to the configurations in Fig. 3, we fitted the THROUGHPUT function on a diverse set of GPU placements and batch sizes in a 64-GPU cluster. Across all DL tasks, the average error of the fitted model was at most 10%, indicating that it represents the observed throughput measurements well.



(a) ImageNet (b) YoloV3 (c) DeepSpeech2 (d) BERT (fine-tune) (e) CIFAR10 (f) Recommendation

Figure 3: System throughput for all models described in Table 1, as measured using g4dn.12xlarge instances in AWS each with 4 NVIDIA T4 GPUs and created within the same placement group. Eqn. 11 was fitted using the observed data that appeared in each plot. TOP: time per training iteration vs. the number of allocated GPUs (log-scaled), with the per-GPU batch size held constant. The GPUs are placed in as few 4-GPU nodes as possible, which causes a sharp increase beyond 4 GPUs (when inter-node network synchronization becomes required). BOTTOM: system throughput (examples per second) vs. total batch size (log-scaled), with the number of GPUs held constant. To the left of the vertical dashed line, the entire mini-batch fits within GPU memory. To the right, the total batch size is achieved using gradient accumulation.

Limits of the throughput model. Pollux models data-parallel training throughput only in the dimensions it cares about, i.e. number and co-locality of GPUs, batch size, and gradient accumulation steps. The simple linear assumptions made in Eqn. 11, although sufficiently accurate for the settings we tested, may diverge from reality for specialized hardware [33], sophisticated synchronization algorithms [7, 65, 72], different parallelization strategies [28, 47, 58, 59], at larger scales [6, 68], or hidden resource contention not related to network used for gradient synchronization. Rather than attempting to cover all scenarios with a single throughput model, we designed $\texttt{GOODPUT}_t$ (Eqn. 4) to be modular so that different equations for `THROUGHPUT` may be easily plugged in without interfering with the core functionalities provided by Pollux.

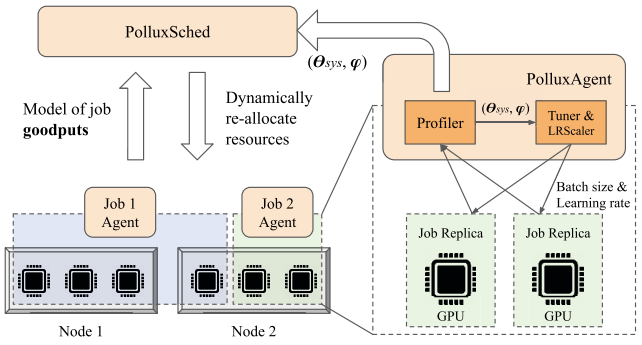


Figure 4: Co-adaptive scheduling architecture of Pollux.

4 Pollux Design and Architecture

Pollux adapts DL job execution at two distinct granularities. First, at a job-level granularity, Pollux dynamically tunes the batch size and learning rate for best utilization of the allocated resources. Second, at the cluster-wide granularity, Pollux dynamically (re-)allocates resources, driven by the goodput of all jobs sharing the cluster combined with cluster-level goals including fairness and job-completion time. To achieve this co-adaptivity in a scalable way, Pollux's design consists of two primary components, as illustrated in Fig. 4.

First, a *PolluxAgent* runs together with each job. It fits the $\texttt{EFFICIENCY}_t$ and `THROUGHPUT` functions for that job, and tunes its batch size and learning rate for efficient utilization of its current allocated resources. PolluxAgent periodically reports the goodput function of its job to the PolluxSched.

Second, the *PolluxSched* periodically optimizes the resource allocations for all jobs in the cluster, taking into account the current goodput function for each job and cluster-wide resource contention. Scheduling decisions made by PolluxSched also account for the overhead associated with resource re-allocations, slowdowns due to network interference between multiple jobs, and resource fairness.

PolluxAgent and PolluxSched *co-adapt* to each other. While PolluxAgent adapts each training job to make efficient use of its allocated resources, PolluxSched dynamically re-allocates each job's resources, taking into account the PolluxAgent's ability to tune its job.

4.1 PolluxAgent: Job-level Optimization

An instance of PolluxAgent is started with each training job. During training, it continually measures the job's gradient noise scale and system throughput, and it reports them to PolluxSched at a fixed interval. It also uses this information to determine the most efficient batch size for its job given its current resource allocations, and adapts its job's learning rate to this batch size using the appropriate plug-in LR scaling rule (e.g. AdaScale for SGD or square-root scaling for Adam).

Online model fitting. In §3.2, we defined the system throughput parameters of a training job as the 7-tuple

$$\theta_{sys} = \left(\alpha_{grad}, \beta_{grad}, \alpha_{sync}^{local}, \beta_{sync}^{local}, \alpha_{sync}^{node}, \beta_{sync}^{node}, \gamma\right), \quad (12)$$

which are required to construct the THROUGHPUT function. Together with the PGNS φ_t (for predicting $\texttt{EFFICIENCY}_t$) and initial batch size M_0, the triple $(\theta_{sys}, \varphi_t, M_0)$ specifies the GOODPUT function. While M_0 is a constant configuration provided by the user, and φ_t can be computed according to §3.1, θ_{sys} is estimated by fitting the THROUGHPUT function to observed throughput values collected about the job during training.

PolluxAgent measures the time taken per iteration, T_{iter}, and records the tuple (a, m, s, T_{iter}) for all combinations of resource allocations a, per-GPU batch size m, and gradient accumulation steps s encountered during its lifetime. Periodically, PolluxAgent fits the parameters θ_{sys} to all of the throughput data collected so far. Specifically, we minimize the root mean squared logarithmic error (RMSLE) between Eqn. 11 and the collected data triples, using L-BFGS-B [73]. We set constraints for each α and β parameter to be non-negative, and γ to be in the range $[1, 10]$. PolluxAgent then reports the updated values of θ_{sys} and φ_t to PolluxSched.

Prior-driven exploration. At the beginning of each job, throughput values have not yet been collected. To ensure that Pollux finds efficient resource allocations through systematic exploration, we impose several priors which bias θ_{sys} towards the belief that throughput scales perfectly with more resources, until such resource configurations are explored.

In particular, we set $\alpha_{sync}^{local} = 0$ while the job had not used more than one GPU, $\alpha_{sync}^{local} = \beta_{sync}^{local} = 0$ while the job had not used more than one node, and $\beta_{sync}^{local} = \beta_{sync}^{node} = 0$ while the job had not used more than two GPUs. This creates the following behavior: each job starts with a single GPU and is initially assumed to scale perfectly to more GPUs. PolluxSched is then encouraged to allocate more GPUs and/or nodes to the job, naturally as part of its resource optimization (§4.2), until the PolluxAgent can estimate θ_{sys} more accurately. Finally, to prevent a job from being immediately scaled out to arbitrarily many GPUs, we restrict the maximum number of GPUs that can be allocated to at most twice the maximum number of GPUs the job has been allocated in its lifetime.

Although other principled approaches to exploration can be applied (e.g., Bayesian optimization), we find that this simple prior-driven strategy is sufficient in our experiments. Sec. 5.3.2 shows that prior-driven exploration performs close (within 2-5%) to an idealized scenario in which the model is fitted offline for each job before being submitted to the cluster.

Training job tuning. With θ_{sys}, φ_t, and M_0, which fully specify the DL job's GOODPUT function at its current training progress, PolluxAgent determines the most efficient per-GPU batch size and gradient accumulation steps,

$$(m^*, s^*) = \underset{m,s}{\operatorname{argmax}}\, \texttt{GOODPUT}(a, m, s), \quad (13)$$

where a is the job's current resource allocation.

Once a new configuration is found, the job will use it for its subsequent training iterations, using the plug-in LR scaling rule to adapt its learning rate appropriately. As the job's $\texttt{EFFICIENCY}_t$ function changes over time, PolluxAgent will periodically re-evaluate the most efficient configuration.

4.2 PolluxSched: Cluster-wide Optimization

The PolluxSched periodically allocates (and re-allocates) resources for every job in the cluster. To determine a set of efficient cluster-wide resource allocations, it maximizes a *fitness function* that is defined as a generalized (power) mean across speedups for each job:

$$\texttt{FITNESS}_p(A) = \left(\frac{1}{J}\sum_{j=1}^{J} \texttt{SPEEDUP}_j(A_j)^p\right)^{1/p}. \quad (14)$$

A is an *allocation matrix* with each row A_j being the allocation vector for a job j, thus A_{jn} is the number of GPUs on node n allocated to job j, and J is the total number of running and pending jobs sharing the cluster. We define the speedup of each job as the factor of goodput improvement using a given resource allocation over using a fair-resource allocation, ie.

$$\texttt{SPEEDUP}_j(A_j) = \frac{\max_{m,s} \texttt{GOODPUT}_j(A_j, m, s)}{\max_{m,s} \texttt{GOODPUT}_j(a_f, m, s)}, \quad (15)$$

where $\texttt{GOODPUT}_j$ is the goodput of job j at its current training iteration, and a_f is a fair resource allocation for the job, defined to be an exclusive $1/J$ share of the cluster.[5]

In §3, we described how the GOODPUT function can be fitted to observed metrics during training and then be evaluated as a predictive model. PolluxSched leverages this ability to predict GOODPUT to maximize FITNESS via a search procedure, and then it applies the outputted allocations to the cluster.

Fairness and the effect of p. When $p = 1$, $\texttt{FITNESS}_p$ is the average of SPEEDUP values across all jobs. This causes PolluxSched to allocate more GPUs to jobs that achieve a high SPEEDUP when provided with many GPUs (i.e., jobs that scale

[5] We note that SPEEDUP has similarities with *finish-time fairness* [43]. But, SPEEDUP is related to training performance at a moment in time, whereas finish-time fairness is related to end-to-end job completion time.

well). However, as $p \to -\infty$, $\texttt{FITNESS}_p$ smoothly approaches the minimum of SPEEDUP values, in which case maximizing $\texttt{FITNESS}_p$ promotes equal SPEEDUP between training jobs, but ignores the overall cluster goodput and resource efficiency.

Thus, p can be considered a "fairness knob", with larger negative values being more fair. A cluster operator may select a suitable value, based on organizational priorities. In our experience and results in §5, we find that $p = -1$ achieves most goodput improvements and reasonable fairness.

Re-allocation penalty. Each time a job is re-allocated to a different set of GPUs, it incurs some delay to re-configure the training process. Using the the popular checkpoint-restart method, we measured between 15 and 120 seconds of delay depending on the size of the model being trained and other initialization tasks in the training code. To prevent an excessive number of re-allocations, when PolluxSched evaluates the fitness function for a given allocation matrix, it applies a penalty for every job that needs to be re-allocated,

$$\texttt{SPEEDUP}_j(A_j) \longleftarrow \texttt{SPEEDUP}_j(A_j) \times \texttt{REALLOC\_FACTOR}_j(\delta).$$

We define $\texttt{REALLOC\_FACTOR}_j(\delta) = (T_j - R_j\delta)/(T_j + \delta)$, where T_j is the age of the training job, R_j is the number of re-allocations incurred by the job so far, and δ is an estimate of the re-allocation delay. Intuitively, $\texttt{REALLOC\_FACTOR}_j(\delta)$ scales $\texttt{SPEEDUP}_j(A_j)$ according to the assumption that the historical average rate of re-allocations for job j will continue indefinitely into the future. Thus, a job that has historically experienced a higher rate of re-allocations will be penalized more for future re-allocations.

Interference avoidance. When multiple distributed DL jobs share a single node, their network usage while synchronizing gradients and model parameters may interfere with each other, causing both jobs to slow down [31]; Xiao et al. [66] report up to 50% slowdown for DL jobs which compete with each other for network resources. PolluxSched mitigates this issue by disallowing different distributed jobs (each using GPUs across multiple nodes) from sharing the same node.

Interference avoidance is implemented as a constraint in Pollux's search algorithm, by ensuring at most one distributed job is allocated to each node. We study the effects of interference avoidance in §5.3.2.

Supporting non-adaptive jobs. In certain cases, a user may want to run a job with a fixed batch size, i.e. $M = M_0$. These jobs are well-supported by PolluxSched, which simply fixes $\texttt{EFFICIENCY}_t$ for that job to 1 and can continue to adapt its resource allocations based solely on its system throughput.

4.3 Implementation

PolluxAgent is implemented as a Python library that is imported into DL training code. We integrated PolluxAgent with PyTorch [51], which uses all-reduce as its gradient synchronization algorithm. PolluxAgent inserts performance profiling code that measures the time taken for each iteration of training, as well as calculating the gradient noise scale. At a fixed time interval, PolluxAgent fits the system throughput model (Eqn. 10) to the profiled metrics collected so far, and reports the fitted system throughput parameters, along with the latest gradient statistics, to PolluxSched. After reporting to PolluxSched, PolluxAgent updates the job's per-GPU batch size and gradient accumulation steps, by optimizing its now up-to-date goodput function (Eqn. 4) with its currently allocated resources.

PolluxSched is implemented as a service in Kubernetes [2]. At a fixed time interval, PolluxSched runs its search algorithm, and then applies the resultant allocation matrix by creating and terminating Kubernetes Pods that run the job workers. To find a good allocation matrix, PolluxSched uses a population-based search algorithm that perturbs and combines candidate allocation matrices to produce higher-value allocation matrices, and finally modifies them to satisfy node resource constraints and interference avoidance. The allocation matrix with the highest fitness score is applied to the jobs running in the cluster.

Both PolluxAgent and PolluxSched require a sub-procedure that optimizes $\texttt{GOODPUT}_t(a, m, s)$ given a fixed a (Eqn. 13). We implemented this procedure by first sampling a range of candidate values for the total batch size M, then finding the smallest s such that $m = \lceil M/s \rceil$ fits into GPU memory according to a user-defined upper-bound, and finally taking the configuration which results in the highest GOODPUT value.

5 Evaluation

We compare Pollux with two state-of-the-art DL schedulers using a testbed cluster with 64 GPUs. Although one primary advantage of Pollux is automatically selecting the configurations for each job, we find that Pollux still reduces average job completion times by 37–50% even when the baseline schedulers are supplied with well-tuned job configurations (a scenario that strongly favors the baseline schedulers). Pollux is able to dynamically adapt each job by trading-off between high-throughput/low-efficiency and low-throughput/high-efficiency modes of training, depending on the current cluster state and training progress.

Using a cluster simulator, we evaluate the impact of specific settings on Pollux, including the total workload intensity, prior-driven exploration, scheduling interval, and interference avoidance. With its fairness knob, Pollux can improve finish-time fairness [43] by 1.5–5.4× compared to baseline DL schedulers. We also reveal a new opportunity for auto-scaling in the cloud by showing that a Pollux-based auto-scaler can potentially reduce the cost of training large models (e.g. ImageNet) by 25%.

5.1 Experimental Setup

Testbed. We conduct experiments using a cluster consisting of 16 nodes and 64 GPUs. Each node is an AWS EC2 `g4dn.12xlarge` instance with 4 NVIDIA T4 GPUs, 48 vCPUs, 192GB memory, and a 900GB SSD. All instances

are launched within the same placement group. We deployed Kubernetes 1.18.2 on this cluster, along with CephFS 14.2.8 to store checkpoints for checkpoint-restart elasticity.

Synthetic Workload Construction. We randomly sampled 160 jobs from the busiest 8-hour range (hours 3–10) in the deep learning cluster traces published by Microsoft [31]. Each job in the orginal trace has information on its submission time, number of GPUs, and duration. However, no information is provided on the model architectures being trained or dataset characteristics. Instead, our synthetic workload consists of the models and datasets described in Table 1.

We categorized each job in the trace and in Table 1 based on their total GPU-time: Small (0–1 GPU-hours), Medium (1–10 GPU-hours), Large (10–100 GPU-hours), and XLarge (100–1000 GPU-hours). For each job in the trace, we picked a training job from Table 1 that is in the same category.

Manually-tuned jobs for baseline DL schedulers. We manually tuned the number of GPUs and batch sizes for each job in our synthetic workload, as follows. We measured the time per training iteration for each model in Table 1 using a range of GPU allocations and batch sizes, and fully trained each model using a range of different batch sizes (see §5.3 for details). We considered a number of GPUs *valid* if using the optimal batch size for that number of GPUs achieves 50% – 80% of the ideal (i.e., perfectly linear) scalability versus using the optimal batch size on a single GPU. For each job submitted from our synthetic workload, we selected its number of GPUs and batch size randomly from its set of valid configurations.

Our job configurations assume that the users are highly rational and knowledgeable about the scalability of the models they are training. Less than 50% of the ideal scalability would lead to under-utilization of resources, and more than 80% of the ideal scalability means the job can still utilize more GPUs efficiently. We emphasize that this assumption of uniformly sophisticated users is unrealistically biased in favor of the baseline schedulers and only serves for comparing Pollux with the ideal performance of baseline systems.

Comparison of DL schedulers. We compare Pollux to two recent deep learning schedulers, Tiresias [22] and Optimus [52], as described in §2.3. Whereas Pollux dynamically co-adapts the number of GPUs and batch sizes of DL training jobs, Optimus only adapts the number of GPUs, and Tiresias adapts neither. To establish a fair baseline for comparison, for all three schedulers, we scale the learning rate using AdaScale for SGD, and the square-root scaling rule for Adam and AdamW.

Pollux. We configured PolluxSched to use a 60s scheduling interval, and compute `REALLOC_FACTOR`(δ) using $\delta = 30$s. PolluxAgent reports its most up-to-date system throughput parameters and gradient statistics every 30s. Unless otherwise specified, the default fairness knob value of $p = -1$ is used.

Tiresias. We configured Tiresias as described in the testbed experiments of Gu et al. [22], with two priority queues and the `PromoteKnob` disabled. We manually tuned the queue threshold to perform well for our synthetic workload. Whenever possible, we placed jobs onto as few different nodes as possible to promote worker locality.

Optimus+Oracle. Optimus leverages a throughput prediction model that is specific to jobs using the parameter server architecture. To account for differences due to the performance model, our implementation of Optimus uses our own throughput model as described in §3.2. Furthermore, Optimus predicts the number of training iterations until convergence by fitting a simple function to the model's convergence curve. Since this method does not work consistently for all models in our synthetic workload, we run each job ahead of time and provide Optimus with the exact number of iterations until completion. We call this version of Optimus *Optimus+Oracle*.

For each job, Tiresias uses the number of GPUs and batch size specified in our synthetic workload. Optimus+Oracle uses the batch size specified, but determines the number of GPUs dynamically. Each job uses gradient accumulation if they are allocated too few GPUs to support the specified batch size.

5.2 Testbed Macrobenchmark Experiments

Table 2 summarizes the results of our testbed experiments for seven configurations: Pollux compared with, first, baseline schedulers using well-tuned job configurations; second, baseline schedulers using more realistic job configurations; third, Pollux using two alternate values for its fairness knob.

Comparisons using well-tuned job configurations. Even when Optimus+Oracle and Tiresias are given well-tuned job configurations as described in §5.1, they are still significantly behind Pollux. In this setting, Pollux (with $p = -1$) achieved 50% and 37% shorter average JCT, 27% and 27% shorter tail (99th percentile) JCT, and 20% and 33% shorter makespan, in comparison to Optimus+Oracle+TunedJobs and Tiresias+TunedJobs, respectively. As we previously noted, this setting highly favors the baseline schedulers, essentially mimicking users who possess expert knowledge about system throughput, statistical efficiency, and how their values change with respect to resource allocations and batch sizes.

One key source of improvement for Pollux is its ability to trade-off between high-throughput/low-efficiency and low-throughput/high-efficiency modes during training. Fig. 5 shows the total number of allocated GPUs and average $\texttt{EFFICIENCY}_t$ during the execution of our synthetic workload. During periods of low cluster contention, Pollux can allocate more GPUs (indicated by **(A)**) and use larger batch sizes to boost training throughput, even at the cost of lower statistical efficiency, because doing so results in an overall higher goodput. On the other hand, during periods of high cluster contention, Pollux may instead use smaller batch sizes to increase statistical efficiency (indicated by **(B)**).

Comparisons using realistic job configurations. Without assistance from a system like Pollux, users are likely to try various numbers of GPUs and batch sizes, before finding a configuration that is efficient. Other users may not invest time

Task	Dataset	Model	Optimizer	LR Scaler	M_0	Validation	Size	Frac. Jobs
Image Classification	ImageNet [12]	ResNet-50 [24]	SGD	AdaScale	200 imgs	75% top1 acc.	XL	2%
Object Detection	PASCAL-VOC [16]	YOLOv3 [55]	SGD	AdaScale	8 imgs	84% mAP	L	6%
Speech Recognition	CMU-ARCTIC [38]	DeepSpeech2 [3]	SGD	AdaScale	20 seqs	25% word err.	M	10%
Question Answering	SQuAD [54]	BERT (finetune) [14]	AdamW	Square-Root	12 seqs	88% F1 score	M	10%
Image Classification	Cifar10 [39]	ResNet18 [24]	SGD	AdaScale	128 imgs	94% top1 acc.	S	36%
Recommendation	MovieLens [23]	NeuMF [25]	Adam	Square-Root	256 pairs	69% hit rate	S	36%

Table 1: Models and datasets used in our evaluation workload. Each training task achieves the provided validation metrics. The fraction of jobs from each category are chosen according to the public Microsoft cluster traces.

Policy	Job Completion Time		Makespan
	Average	99%tile	
Pollux ($p=-1$)	**0.76h**	11h	**16h**
Optimus+Oracle+TunedJobs	1.5h	15h	20h
Tiresias+TunedJobs	1.2h	15h	24h
Optimus+Oracle	2.7h	22h	28h
Tiresias	2.8h	25h	31h
Pollux ($p=+1$)	0.83h	**10h**	**16h**
Pollux ($p=-10$)	0.84h	12h	18h

Table 2: Summary of testbed experiments.

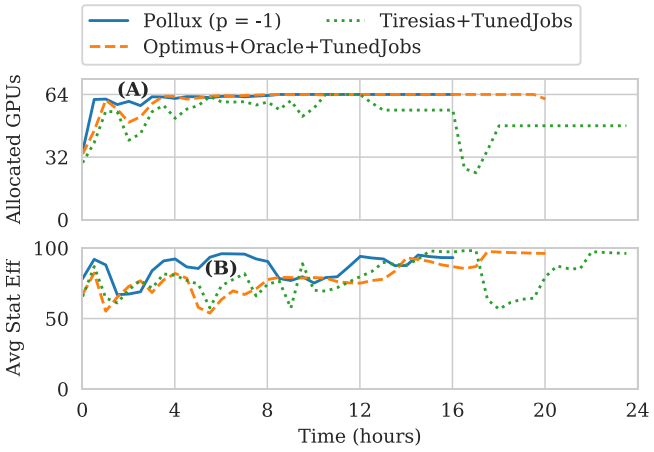


Figure 5: Comparison between Pollux ($p=-1$), Optimus, and Tiresias while executing our synthetic workload (with tuned jobs). TOP: average cluster-wide allocated GPUs over time. BOTTOM: average cluster-wide statistical efficiency over time. Tiresias+TunedJobs dips between hours 16 and 20 due to a 24-GPU job blocking a 48-GPU job from running.

into configuring their jobs well in the first place.

To set a more realistically configured baseline, we ran Optimus+Oracle and Tiresias on a version of our synthetic workload with the number of GPUs exactly as specified in the Microsoft cluster trace. The batch size was chosen to be the baseline batch size M_0 times the number of GPUs, which is how we expect most users to initially configure their distributed training jobs. We find that these jobs typically use fewer GPUs and smaller batch sizes than their well-configured counterparts.

Using this workload, we find that Pollux has 72% and 73% shorter average JCT, 50% and 56% shorter tail JCT, and 43% and 48% shorter makespan, in comparison to Optimus+Oracle and Tiresias, respectively. Even though Optimus+Oracle can dynamically increase the GPU allocation of each job, it still only slightly outperforms Tiresias because it does not also increase the batch size to better utilize those additional GPUs.

A closer look at co-adapted job configurations. Fig. 6 (LEFT) shows the configurations chosen by Pollux for one ImageNet training job as the synthetic workload progresses. **(A)** during the initial period of low cluster contention, more GPUs are allocated to ImageNet, causing a larger batch size to be used and lowering statistical efficiency. **(B)** during the subsequent period of high cluster contention, fewer GPUs are allocated to ImageNet, causing a smaller batch size to be used and raising statistical efficiency. **(C)** when the cluster contention comes back down, ImageNet continues to be allocated more GPUs and uses a larger batch size. However, we note that the batch size per GPU is much higher than in the first low-contention period, since the job is now in its final, high-statistical-efficiency phase of training. We see similar trade-offs being made over time for two YOLOv3 jobs (RIGHT).

Effect of the fairness knob. We ran Pollux using three values of the fairness knob, $p = 1, -1, -10$. Compared with no fairness ($p = 1$), introducing a moderate degree of fairness ($p = -1$) improved the average job completion time (JCT) but degraded the tail JCT. This is because[6], in our synthetic workload, the tail JCT comprises of long but scalable jobs (i.e. ImageNet), which take a large number of GPUs away from other jobs in the absence of fairness ($p=1$). However, further increasing fairness ($p=-10$) degraded performance in average JCT, tail JCT, and makespan. In §5.3.1, we present a more detailed analysis of the impact of p on scheduling fairness.

System overheads. During each 60s scheduling interval, PolluxSched spent an average of 1 second on 1 vCPU computing the cluster allocations by optimizing the $\texttt{FITNESS}_p$ function. On average, each job was re-allocated resources once every 7 minutes, resulting in an average 8% run-time overhead due to checkpoint-restarts. Each PolluxAgent fits its throughput model parameters on its latest observed metrics every 30 seconds, taking an average of 0.2 seconds each time. Finding the

[6]We note that $p = -1$ (harmonic mean over speedups) may be more suitable than $p=1$ (arithmetic mean) when optimizing for the average JCT.

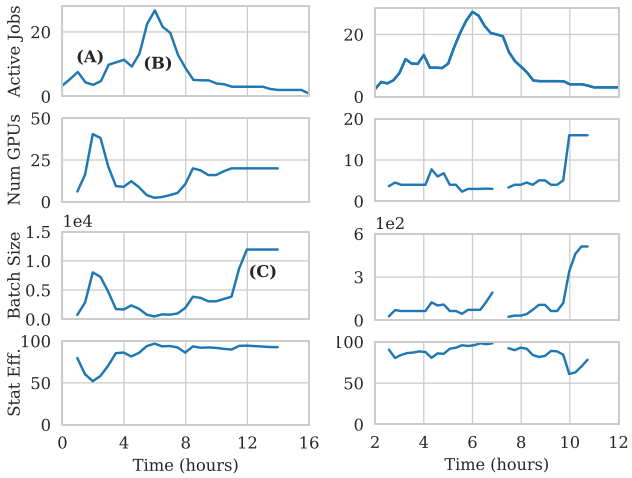


Figure 6: Co-adaptation over time of one ImageNet job (LEFT) and two YOLOv3 jobs (RIGHT) using Pollux ($p=-1$). ROW 1: number of jobs actively sharing the cluster. ROW 2: number of GPUs allocated to the job. ROW 3: batch size (images) used. ROW 4: statistical efficiency (%).

optimal per-GPU batch size and gradient accumulation steps by optimizing $\texttt{GOODPUT}_t$ takes an average of 0.4 milliseconds.

5.3 Simulator Experiments

We built a discrete-time cluster simulator in order to evaluate a broader set of workloads and settings. Our simulator is constructed by measuring the performance and gradient statistics of each model in Table 1, under many different resource and batch size configurations, and re-playing them for each simulated job. This way, we are able to simulate both the system throughput and statistical efficiency of the jobs in our workload.

Unless stated otherwise, each experiment in this section is repeated on 8 different workload traces generated using the same duration, number of jobs, and job size distributions as in §5.2, and we report the average results across all 8 traces.

Simulator construction. For each job in Table 1, we measured the time per training iteration for 146 different GPU allocations+placements in our testbed cluster of 16 nodes and 64 total GPUs. For each allocation, we measured a range of batch sizes up to the GPU memory limit. To simulate the throughput for a job, we queried a multi-dimensional linear interpolation on the configurations we measured. For each model, we also measured the (pre-conditioned) gradient noise scale during training using a range of batch sizes, and across every epoch. To simulate the statistical efficiency for a job using a certain batch size, we linearly interpolated its value of the PGNS between the two nearest batch sizes we measured.

Simulator fidelity. The data we collected about each job enables our simulator to reproduce several system effects, including the performance impact of different GPU placements. We also simulate the overhead of checkpoint-restarts by injecting a 30-second delay for each job that has its resources re-allocated. Unless stated otherwise, we do not simulate any network interference between different jobs. We study the effects of interference in more detail in §5.3.2.

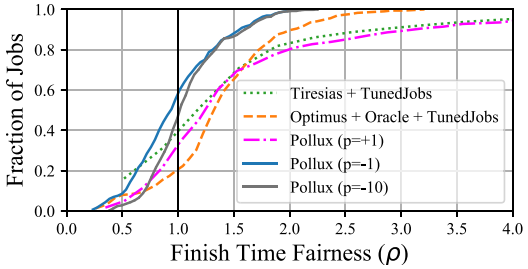


Figure 7: CDF of Finish Time Fairness (ρ).

Compared with our testbed experiments in §5.2, we find that our simulator obtains similar factors of improvement, showing that Pollux reduces the average JCT by 48% and 32% over Optimus+Oracle+TunedJobs and Tiresias+TunedJobs.

5.3.1 Scheduling Fairness

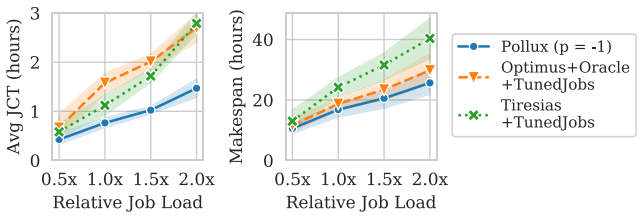
We evaluate the scheduling fairness of Pollux using *finish-time fairness* [43] (denoted by ρ), which is defined to be the ratio of a job's JCT running on shared resources to that of the job running in an isolated and equally-partitioned cluster. Under this metric, jobs with $\rho < 1$ have been treated better-than-fair by the cluster scheduler, while jobs with $\rho > 1$ have been treated worse-than-fair.

In Fig. 7, we compare the finish-time fairness of Pollux with Optimus+Oracle+TunedJobs and Tiresias+TunedJobs. Pollux with $p = 1$ results in poor fairness, similar to Tiresias+TunedJobs, which is apparent as a long tail of jobs with $\rho > 4$. Optimus+Oracle+TunedJobs obtains better fairness due to its allocation algorithm which attempts to equalize the JCT improvement for each job. Pollux with $p = -1$ provides the best fairness, with 99% of jobs achieving $\rho < 2$, and does so while still providing significant performance increases (Table 2). For $p = -10$, we observe slightly worse fairness overall, caused by PolluxSched incurring a larger number of re-allocations due to ignoring the cost in favor of equalizing speedups at all times.

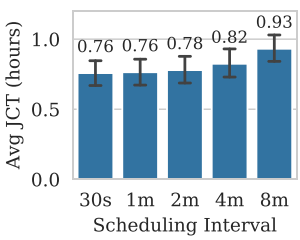
To provide context, we note that the curves for Tiresias and Optimus are consistent with those reported (for different workloads) by Mahajan et al. [43]. Although their Themis system is not available for direct comparison, the ρ range for Pollux with $p = -1$ is similar to the range reported for Themis. The max-ρ improvements (1.5× and 5.4×) over Tiresias and Optimus are also similar.

5.3.2 Other Effects on Scheduling

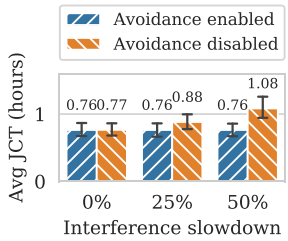
Sensitivity to job load. We compare the performance of Pollux, Optimus+Oracle+TunedJobs, and Tiresias+TunedJobs



(a) Varying the workload intensity.



(b) Varying scheduling interval.



(c) Varying job interference.

Figure 8: Effects of various parameters on Pollux, error bars and bands represent 95% confidence intervals.

for increasing workload intensity in terms of rate of job submissions. Fig. 8a shows the results. As expected, all three scheduling policies suffer longer average JCT and makespan as the load is increased. Across all job loads, Pollux maintains similar relative improvements over the baseline schedulers.

Impact of prior-driven exploration. Pollux explores GPU allocations for each DL job from scratch during training (Sec. 4.1). We evaluated the potential improvement from more efficient exploration by seeding each job's throughput models using historical data collected offline. We observed minor (2–5%) reduction in JCT for short jobs like CIFAR10, but no significant change for longer running jobs, indicating low overhead from Pollux's prior-driven exploration.

Impact of scheduling interval. We ran Pollux using a range of values for its scheduling interval, as shown in Fig. 8b. We find that Pollux performs similarly well in terms of average JCT for intervals up to 2 minutes, while longer intervals result in performance degradation. Since newly-submitted jobs can only start during the next scheduling interval, we would expect an increase in the average queuing time due to longer scheduling intervals. However, we find that queuing contributed to roughly half of the performance degradation observed, indicating that Pollux still benefits from a relatively frequent adjustment of resource allocations.

Impact of interference avoidance. To evaluate the impact of PolluxSched's interference avoidance constraint, we artificially inject various degrees of slowdown for distributed jobs sharing the same node. Fig. 8c shows the results. With interference avoidance enabled, the average JCT is unaffected by even severe slowdowns, because network contention is completely mitigated. However, without interference avoidance, the average JCT is 1.4× longer when the interference slowdown is 50%. On the other hand, in the ideal scenario when there is zero slowdown due to interference, PolluxSched performs similarly whether or not interference avoidance is enabled. This indicates that PolluxSched is still able to find efficient cluster allocations while obeying the interference avoidance constraint.

5.4 More Applications of Pollux

5.4.1 Cloud Auto-scaling

In cloud environments, computing resources can be obtained and released as required, and users pay for the duration they hold onto those resources. Goodput-driven scheduling presents a unique opportunity: when a DL model's statistical efficiency increases during training, it may be more cost-effective to provision more cloud resources and use larger batch sizes during the later epochs of a large training job, rather than earlier on. We present some preliminary evidence using our cluster simulator, and note that a full design of an auto-scaling system based on goodput may be the subject of future work.

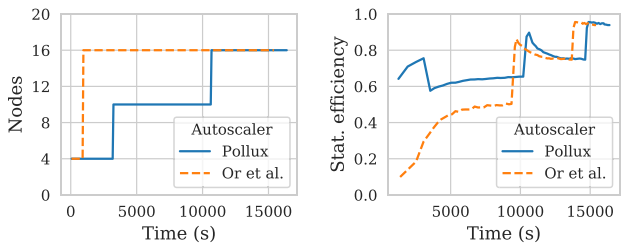
Auto-scaling ImageNet training. We implemented a simple auto-scaling policy using Pollux's goodput function. During training, we scaled up the number of nodes whenever $\max_{m,s} \texttt{GOODPUT}_t(a,m,s)/\texttt{SUM}(a) > U \cdot \max_{m,s} \texttt{GOODPUT}_t(1,m,s)$, i.e. the goodput exceeds some fraction U of the predicted ideal goodput assuming perfect scalability. We set $U = 2/3$, and increased to a number of nodes such that the predicted goodput is approximately $L = 1/2$ of the predicted ideal goodput.

Fig. 9 compares our Pollux-based auto-scaler with the auto-scaler proposed by Or *et al.* [50], which allows the batch size to be increased during training, but models job performance using the system throughput rather than the goodput. Since the system throughput does not change with training progress, throughput-based autoscaling (Or *et al.*) quickly scales out to more nodes and a larger batch size (Fig. 9a), which remains constant thereafter. On the other hand, Pollux starts with a small number of nodes, and gradually increases the number of nodes as the effectiveness of larger batch sizes improves over time. Fig. 9b shows that Pollux maintains a high statistical efficiency throughout training. Overall, compared to Or *et al.*'s throughput-based auto-scaling, Pollux trains ImageNet with 25% cheaper cost, with only a 6% longer completion time.

5.4.2 Hyper-parameter Optimization (HPO)

Hyper-parameter optimization (HPO) is an important DL workload. In HPO, the user defines a *search space* over relevant model hyper-parameters. A HPO algorithm (aka a trial scheduler) submits many training jobs (trials) to evaluate the effectiveness of particular hyper-parameters, in terms of objectives such as model accuracy or energy efficiency.

Different HPO algorithm types manage trials differently. For example, Bayesian optimization algorithms [37, 62] may submit a few training jobs at a time, and determine future trials based on the fully-trained results of previous trials.



(a) Number of nodes over time. (b) Statistical efficiency over time.

Figure 9: Goodput-based auto-scaling (Pollux) vs throughput-based auto-scaling (Or et al.) for ImageNet training.

Policy	Accuracy (Top 5 trials)	Avg JCT	Makespan
Pollux	95.4±0.2	25min	10h
Baseline	95.5±0.3	34min	14h

Table 3: Summary of HPO experiments.

Bandit-based algorithms [41] may launch a large number of trials at once and early-stop ones that appear unpromising.

A full evaluation on how Pollux affects different HPO algorithm types is future work. Table 3 shows results from tuning a ResNet18 model trained on the CIFAR10 dataset, using a popular Bayesian optimization-based HPO algorithm known as the Tree-structured Parzen Estimator (TPE) [5]. The search space covers the learning rate and annealing, momentum, weight decay, and network width hyper-parameters. We configured TPE so that 4 trials run concurrently with each other, and 100 trials are run in total. The testbed consists of two NVIDIA DGX A100 nodes, each with 8 A100 GPUs. The baseline scheduler assigns a static allocation of 4 GPUs (all on the same node) to each trial and uses a fixed per-GPU batch size for every trial. As expected, similar accuracy values are achieved, but Pollux completes HPO 30% faster due to adaptive (re-)allocation of resources as trials progress and adaptive batch sizes.

5.5 Artifact

We provide an artifact containing the full implementation of Pollux, benchmark model implementations (Table 1), testbed experiment scripts (Sec. 5.2), cluster simulator implementation and results (Sec. 5.3), available at `https://github.com/petuum/adaptdl/tree/osdi21-artifact`. The raw testbed experiment (Sec. 5.2) logs and analysis scripts are provided at `https://github.com/petuum/pollux-results`.

6 Additional Related Work

Prior DL schedulers are discussed in §2.3.

Adaptive batch size training. Recent work on DL training algorithms have explored dynamically adapting batch sizes for better efficiency and parallelization. AdaBatch [13] increases the batch size at pre-determined iterations during training, while linearly scaling the learning rate. Smith et al. [61] suggest that instead of decaying the learning rate during training, the batch size should be increased instead. CABS [4] adaptively tunes the batch size and learning rate during training using similar gradient statistics as Pollux.

These works have a common assumption that extra computing resources are available to parallelize larger batch sizes whenever desired, which is rarely true inside shared-resource environments. Pollux complements existing adaptive batch size strategies by adapting the batch size and learning rate in conjunction with the amount of resources currently available. Alternatively, anytime minibatch [17] adapts the batch size to mitigate stragglers in distributed training.

KungFu [44] supports adaptive training algorithms, including adaptive batch sizes, by allowing applications to define custom adaptation policies and enabling efficient adaptation and monitoring during training. Although KungFu is directed at single-job training and Pollux at cluster scheduling, we believe KungFu offers useful tools which can be used to implement the adaptive policies used by the PolluxAgent.

Hyper-parameter tuning. A large body of work focuses on tuning the hyper-parameters for ML and DL models [5, 18, 29, 34, 49], which typically involves many training jobs [1, 20] as discussed earlier. Although batch size and learning rate are within the space of hyper-parameters often optimized by these systems, Pollux's goal is fundamentally different. Whereas HPO algorithms search for the highest model quality, Pollux adapts the batch size and learning rate for the most efficient execution for each job, while not degrading model quality.

7 Conclusion

Pollux is a DL cluster scheduler that co-adaptively allocates resources, while at the same time tuning each training job to best utilize those resources. We present a formulation of goodput that combines system throughput and statistical efficiency for distributed DL training. Based on the principle of goodput maximization, Pollux automatically and jointly tunes the resource allocations, batch sizes, and learning rates for DL jobs, which can be particularly difficult for users to configure manually. Pollux outperforms and is more fair than recent DL schedulers, even if users can configure their jobs well, and provides even bigger benefits with more realistic user knowledge.

8 Acknowledgements

We thank our shepherd, Michael Isard, and the anonymous OSDI reviewers for their insightful comments and suggestions that improved our work. We also thank our colleagues from Petuum — Omkar Pangarkar, Richard Fan, Peng Wu, Jayesh Gada, and Vishnu Vardhan — for their invaluable contributions toward the open source implementation of Pollux.

References

[1] Introduction to katib. https://www.kubeflow.org/docs/components/hyperparameter-tuning/overview/. Accessed: 2020-05-18.

[2] Production-grade container orchestration - kubernetes. https://kubernetes.io/. Accessed: 2020-05-18.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 173–182. JMLR.org, 2016.

[4] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. *CoRR*, abs/1612.05086, 2016.

[5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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 H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[7] J. Canny and Huasha Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *SDM*, 2013.

[8] 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. *CoRR*, abs/1512.01274, 2015.

[9] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[10] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[13] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *CoRR*, abs/1712.02029, 2017.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[16] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[17] Nuwan Ferdinand, Haider Al-Lawati, Stark Draper, and Matthew Nokleby. ANYTIME MINIBATCH: EXPLOITING STRAGGLERS IN ONLINE DISTRIBUTED OPTIMIZATION. In *International Conference on Learning Representations*, 2019.

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[19] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *CoRR*, abs/1811.12941, 2018.

[20] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[22] 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.

[23] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[25] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[26] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[27] 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 '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[29] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[30] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020.

[31] 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.

[32] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. Adascale {sgd}: A scale-invariant algorithm for distributed training, 2020.

[33] N. Jouppi, C. Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, R. Bajwa, Sarah Bates, Suresh Bhatia, N. Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, M. Dau, J. Dean, Ben Gelb, T. Ghaemmaghami, R. Gottipati, William Gulland, R. Hagmann, C. Ho, Doug Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, N. Kumar, Steve Lacy, J. Laudon, James Law, Diemthu Le, Chris Leary, Z. Liu, Kyle A. Lucke, Alan Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, K. Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, J. Ross, Matt Ross, Amir Salek, E. Samadiani, C. Severn,

G. Sizikov, Matthew Snelham, J. Souter, D. Steinberg, Andy Swing, Mercedes Tan, G. Thorson, Bo Tian, H. Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, W. Wang, Eric Wilcox, and D. Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[34] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *arXiv preprint arXiv:1903.06694*, 2019.

[35] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[37] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, 2017.

[38] John Kominek and Alan Black. The cmu arctic speech databases. *SSW5-2004*, 01 2004.

[39] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[40] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[42] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[43] 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.

[44] 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. USENIX Association, November 2020.

[45] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.

[46] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018.

[47] 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 '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[48] 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.

[49] Willie Neiswanger, Kirthevasan Kandasamy, Barnabas Poczos, Jeff Schneider, and Eric Xing. Probo: a framework for using probabilistic programming in bayesian optimization. *arXiv preprint arXiv:1901.11515*, 2019.

[50] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. In *Proceedings of Machine Learning and Systems 2020*, pages 400–411. 2020.

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward 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 H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[52] 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.

[53] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed

machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, Boston, MA, July 2018. USENIX Association.

[54] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[55] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[56] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[57] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[58] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[59] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019.

[60] S. L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. *ArXiv*, abs/1710.06451, 2018.

[61] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[62] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25:2951–2959, 2012.

[63] WenWu Wang and Ping Yu. Asymptotically optimal differenced estimators of error variance in nonparametric regression. *Computational Statistics & Data Analysis*, 105:125 – 143, 2017.

[64] Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes. In *International Conference on Machine Learning*, pages 6677–6686. PMLR, 2019.

[65] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394, 2015.

[66] 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.

[67] 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.

[68] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds, 2019.

[69] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. *CoRR*, abs/1901.08256, 2019.

[70] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association.

[71] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 390–404, New York, NY, USA, 2017. Association for Computing Machinery.

[72] Huasha Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014.

[73] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, December 1997.

Oort: Efficient Federated Learning via Guided Participant Selection

Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

Federated Learning (FL) is an emerging direction in distributed machine learning (ML) that enables in-situ model training and testing on edge data. Despite having the same end goals as traditional ML, FL executions differ significantly in scale, spanning thousands to millions of participating devices. As a result, data characteristics and device capabilities vary widely across clients. Yet, existing efforts randomly select FL participants, which leads to poor model and system efficiency.

In this paper, we propose Oort to improve the performance of federated training and testing with guided participant selection. With an aim to improve time-to-accuracy performance in model training, Oort prioritizes the use of those clients who have both data that offers the greatest utility in improving model accuracy and the capability to run training quickly. To enable FL developers to interpret their results in model testing, Oort enforces their requirements on the distribution of participant data while improving the duration of federated testing by cherry-picking clients. Our evaluation shows that, compared to existing participant selection mechanisms, Oort improves time-to-accuracy performance by 1.2×-14.1× and final model accuracy by 1.3%-9.8%, while efficiently enforcing developer-specified model testing criteria at the scale of millions of clients.

1 Introduction

Machine learning (ML) today is experiencing a paradigm shift from cloud datacenters toward the edge [18, 40]. Edge devices, ranging from smartphones and laptops to enterprise surveillance cameras and edge clusters, routinely store application data and provide the foundation for machine learning beyond datacenters. With the goal of not exposing raw data, large companies such as Google and Apple deploy *federated learning (FL)* for computer vision (CV) and natural language processing (NLP) tasks across user devices [2, 24, 30, 77]; NVIDIA applies FL to create medical imaging AI [49]; smart cities perform in-situ image training and testing on AI cameras to avoid expensive data migration [32, 38, 51]; and video streaming and networking communities use FL to interpret and react to network conditions [10, 76].

Although the life cycle of an FL model is similar to that in traditional ML, the underlying execution in FL is spread across thousands to millions of devices in the wild. Similar to traditional ML, the FL developer often first prototypes model architectures and hyperparameters with a proxy dataset. After selecting a suitable configuration, she can use federated training to improve model performance by training across a crowd of participants [18, 40]. The wall clock time for training a model to reach an accuracy target (i.e., time-to-accuracy) is still a key performance objective even though it may take significantly longer than centralized training [40]. To circumvent biased or stale proxy data in hyperparameter tuning [57], to inspect these models being trained, or to validate deployed models after training [75,76], developers may want to perform federated testing on the real-life client data, wherein enforcing their requirements on the testing set (e.g., N samples for each category or following the representative categorical distribution[1]) is crucial for them to reason about model performance under different data characteristics [20, 57].

Unfortunately, clients may not all be simultaneously available for FL training or testing [40]; they may have heterogeneous data distributions and system capabilities [18, 34]; and including too many may lead to wasted work and suboptimal performance [18] (§2). Consequently, a fundamental problem in practical FL is the *selection of a "good" subset of clients as participants*, where each participant locally processes its own data, and only their results are collected and aggregated at a (logically) centralized coordinator.

Existing works optimize for *statistical model efficiency* (i.e., better training accuracy with fewer training rounds) [22, 47, 59, 72] or *system efficiency* (i.e., shorter rounds) [54, 68], while randomly selecting participants. Although random participant selection is easy to deploy, unfortunately, it results in poor performance of federated training because of large heterogeneity in device speed and/or data characteristics. Worse, random participant selection can lead to biased testing sets and loss of confidence in results. As a result, developers often resort to more participants than perhaps needed [57, 73].

We present Oort for FL developers to enable guided participant selection throughout the life cycle of an FL model (§3). Specifically, Oort cherry-picks participants to improve time-to-accuracy performance for federated training, and it enables

[1] A categorical distribution is a discrete probability distribution showing how a random variable can take the result from one of K possible categories.

developers to specify testing criteria for federated model testing. It makes informed participant selection by relying on the information already available in existing FL solutions [40] with little modification.

Selecting participants for federated training is challenging because of the trade-off between heterogeneous system and statistical model utilities both across clients and of any specific client over time (as the trained model changes). First, simply picking clients with high statistical utility can lead to longer training rounds due to the coupled nature of client data and system performance. The challenge is further exacerbated by the large population, as capturing the latest utility of all clients is impractical. As such, we identify clients with high statistical utility, which is measured in terms of their most recent aggregate training loss, adjusted for spatiotemporal variations, and penalize the utility of a client if her system speed is likely to elongate the duration necessary to complete global aggregation. To navigate the sweet point of jointly maximizing statistical and system efficiency, we adaptively allow for longer training rounds to admit clients with higher statistical utility. We then employ an online exploration-exploitation strategy to probabilistically select participants among high-utility clients for robustness to outliers. Our design can accommodate diverse selection criteria (e.g., fairness), and deliver improvements while respecting privacy (§4).

Although FL developers often have well-defined requirements on their testing data, satisfying these requirements is not straightforward. Similar to traditional ML, developers may request a testing dataset that follows the global distribution to avoid testing on all clients [35, 57]. However, clients' data characteristics in some private FL scenarios may not be available [27, 77]. To preserve the deviation target of participant data from the global, Oort performs participant selection by bounding the number of participants needed. Second, for cases where clients' data characteristics are provided [51], developers can specify specific distribution of the testing set to debug model efficiency (e.g., using balanced distribution) [15, 78]. At scale, satisfying this requirement in FL suffers large overhead. Therefore, we propose a scalable heuristic to efficiently enforce developer requirements, while optimizing the duration of testing (§5).

We have integrated Oort with PySyft (§6) and evaluated it across various FL tasks with real-world workloads (§7). [2] Compared to the state-of-the-art selection techniques used in today's FL deployments [21, 73, 77], Oort improves time-to-accuracy performance by 1.2×-14.1× and final model accuracy by 1.3%-9.8% for federated model training, while achieving close to upper-bound statistical performance. For federated model testing, Oort can efficiently respond to developer-specified data distribution across millions of clients, and improves the end-to-end testing duration by 4.7× on average over state-of-the-art solutions.

[2] Oort is available at https://github.com/SymbioticLab/Oort.

Overall, we make the following contributions in this paper:

1. We highlight the tension between statistical and systems efficiency when selecting FL participants and present Oort to effectively navigate the tradeoff.
2. We propose participant selection algorithms to improve the time-to-accuracy performance of training and to scalably enforce developers' FL testing criteria.
3. We implement and evaluate these algorithms at scale in Oort, showing both statistical and systems performance improvements over the state-of-the-art.

2 Background and Motivation

We start with a quick primer on federated learning (§2.1), followed by the challenges it faces based on our analysis of real-world datasets (§2.2). Next, we highlight the key shortcomings of the state-of-the-art that motivate our work (§2.3).

2.1 Federated Learning

Training and testing play crucial roles in the life cycle of an FL model, whereas they have different criteria.

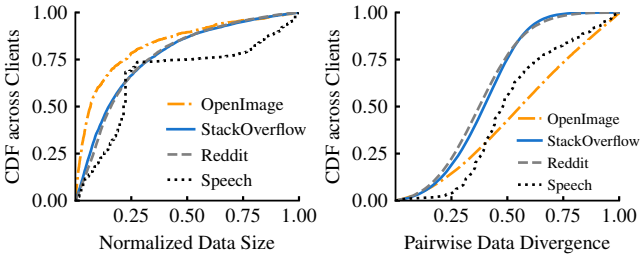
Federated model training aims to learn an accurate model across thousands to potentially millions of clients. Because of the large population size and diversity of user data and their devices in FL, training runs on a subset of clients (hundreds of participants) in each round, and often takes hundreds of rounds (each round lasts a few minutes) and several days to complete. For example, in Gboard keyboard, Google runs federated training of NLP models over weeks across 1.5 million end devices [4, 77]. For a given model, achieving a target model accuracy with less wall clock time (i.e., time-to-accuracy) is still the primary target [47, 63].

To inspect a model's accuracy during training (e.g., to detect cut-off accuracy), to validate the trained model before deployment [21, 73, 77], or to circumvent biased proxy data in hyperparameter tuning [15, 62], FL developers sometimes test model's performance on real-life datasets. Similar to traditional ML, developers often request the representativeness of the testing set with requirements like *"50k representative samples"* [15], or *"x samples of class y"* to investigate model performance on specific categories [78]. When the data characteristics of participants are not available, coarse-grained yet non-trivial requests, such as *"a subset with less than X% data deviation from the global"* are still informative [53, 57].

2.2 Challenges in Federated Learning

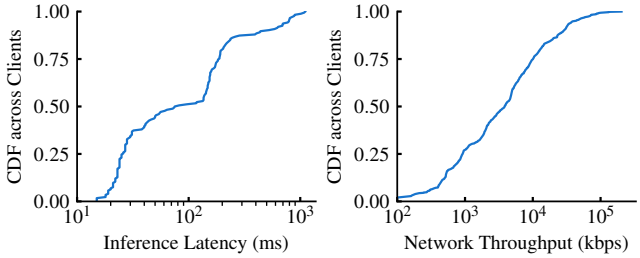
Apart from the challenges faced in traditional ML, FL introduces new challenges in terms of data, systems, and privacy.

Heterogeneous statistical data. Data in each FL participant is typically generated in a distributed manner under different contexts and stored independently. For example, images collected by cameras will reflect the demographics of each camera's location. This breaks down the widely-accepted assumption in traditional ML that samples are independent and identically distributed (i.i.d.) from a data distribution.



(a) Unbalanced data size. (b) Heterogeneous data distribution.

Figure 1: ***Client data differs in size and distribution greatly.***



(a) Heterogeneous compute capacity. (b) Heterogeneous network capacity.

Figure 2: ***Client system performance differs significantly.***

We analyze four real-world datasets for CV (OpenImage [3]) and NLP (StackOverflow [9], Reddit [8] and Google Speech [74]) tasks. Each consists of thousands or up to millions of clients and millions of data points. In each individual dataset, we see a high statistical deviation across clients not only in the quantity of samples (Figure 1(a)) but also in the data distribution (Figure 1(b)).[3]

Heterogeneous system performance. As individual data samples are tightly coupled with the participant device, in-situ computation on this data experiences significant heterogeneity in system performance. We analyze the inference latency of MobileNet [65] across hundreds of mobile phones used in a real-world FL deployment [77], and their available bandwidth. Unlike the homogeneous setting in datacenter ML, system performance across clients exhibits an order-of-magnitude difference in both computational capabilities (Figure 2(a)) and network bandwidth (Figure 2(b)).

Enormous population and pervasive uncertainty. While traditional ML runs in a well-managed cluster with a number of machines, federated learning often involves up to millions of clients, making it challenging for the coordinator to efficiently identify and manage valuable participants. During execution, devices often vary in system performance [18,40] – they may slow down or drop out – and the model performance varies in FL training as the model updates over rounds.

Privacy concerns. Inquiring about the privacy-sensitive information of clients (e.g., raw data or even data distribution) can alienate participants in contributing to FL [24, 66, 67].

[3]We report the pairwise deviation of categorical distributions between two clients, using the popular L1-divergence metric [55].

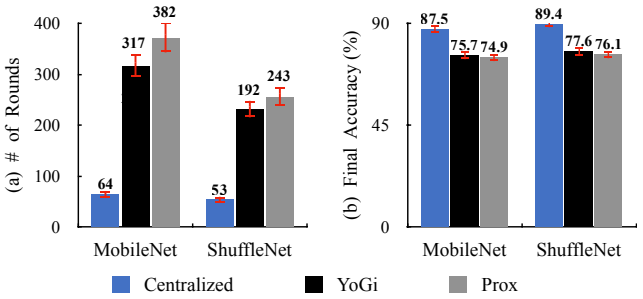


Figure 3: ***Existing works are suboptimal in: (a) round-to-accuracy performance and (b) final model accuracy. (a) reports number of rounds required to reach the highest accuracy of Prox on MobileNet (i.e., 74.9%). Error bars show standard deviation.***

Hence, realistic FL solutions have to seek efficiency improvements but with limited information available in practical FL, and their deployments must be non-intrusive to clients.

2.3 Limitations of Existing FL Solutions

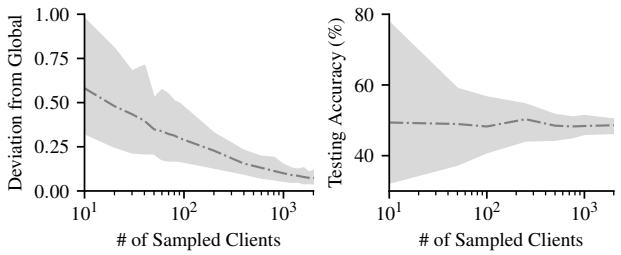
While existing FL solutions have made considerable progress in tackling some of the above challenges (§8), they mostly rely on hindsight – given a pool of participants, they optimize model performance [48,59] or system efficiency [54] to tackle data and system heterogeneity. However, the potential for curbing these disadvantages by cherry-picking participants before execution has largely been overlooked. For example, FL training and testing today still rely on randomly picking participants [18], which leaves large room for improvements.

Suboptimality in maximizing efficiency. We first show that today's participant selection underperforms for FL solutions. Here, we train two popular image classification models tailored for mobile devices (i.e., MobileNet [65] and ShuffleNet [80]) with 1.6 million images of the OpenImage dataset, and randomly pick 100 participants out of more than 14k clients in each training round. We consider a performance *upper bound* by creating a hypothetical centralized case where images are evenly distributed across only 100 clients, and train on all 100 clients in each round. As shown in Figure 3, even with state-of-the-art optimizations, such as YoGi [63] and Prox [47],[4] the round-to-accuracy and final model accuracy are both far from the upper-bound. Moreover, overlooking the system heterogeneity can elongate each round, further exacerbating the suboptimality of time-to-accuracy performance.

Inability to enforce data selection criteria. While an FL developer often fine-tunes her model by understanding the input dataset, existing solutions do not provide any systems support for her to express and reason about what data her FL model was trained or tested on. Even worse, existing participant selection not only inflates the execution, but can lead to bias and loss of confidence in results [20, 34].

To better understand how existing works fall short, we

[4]These two adapt traditional stochastic gradient descent algorithms to tackle the heterogeneity of the client datasets.



(a) Data deviation vs. participant size. (b) Accuracy vs. participant size.

Figure 4: ***Participant selection today leads to (a) deviations from developer requirements, and thus (b) affects testing result. Shadow indicates the [min, max] range of y-axis values over 1000 runs given the same x-axis input; each line reports the median.***

take the global categorical distribution as an example requirement, and experiment with the above pre-trained ShuffleNet model. Figure 4(a) shows that: (i) even for the same number of participants, random selection can result in noticeable data deviations from the target distribution; (ii) while this deviation decreases as more participants are involved, it is non-trivial to quantify how it varies with different number of participants, even if we ignore the cost of enlarging the participant set. Worse, when even selecting many participants, developers can not enforce other distributions (e.g., balanced distribution for debugging [15]) with random selection. One natural effect of violating developer specification is bias in results (Figure 4(b)), where we test the accuracy of the same model on these participants. We observe that a biased testing set results in high uncertainties in testing accuracy.

3 Oort Overview

Oort improves FL training and testing performance by judiciously selecting participants while enabling FL developers to specify data selection criteria. In this section, we provide an overview of how Oort fits in the FL life cycle to help the reader follow the subsequent sections.

3.1 Architecture

At its core, Oort is a participant selection framework that identifies and cherry-picks valuable participants for FL training and testing. It is located inside the coordinator of an FL framework and interacts with the driver of an FL execution (e.g., PySyft [7] or Tensorflow Federated [11]). Given developer-specified criteria, it responds with a list of participants, whereas the driver is in charge of initiating and managing execution on the Oort-selected remote participants.

Figure 5 shows how Oort interacts with the developer and FL execution frameworks. ① *Job submission*: the developer submits and specifies the participant selection criteria to the FL coordinator in the cloud. ② *Participant selection*: the coordinator enquires the clients meeting eligibility properties (e.g., battery level), and forwards their characteristics (e.g., liveness) to Oort. Given the developer requirements (and execution feedbacks in case of training ②a), Oort selects participants based on the given criteria and notifies the coordinator of this participant selection (②b). ③ *Execution*: the coordinator distributes relevant profiles (e.g., model) to these participants, and then each participant independently computes results (e.g., model weights in training) on her data; ④ *Aggregation*: when participants complete the computation, the coordinator aggregates updates from participants.

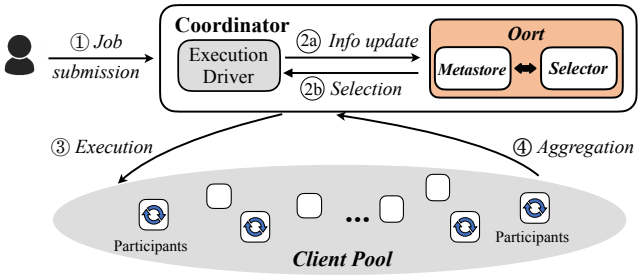


Figure 5: ***Oort architecture. The driver of the FL framework interacts with Oort using a client library.***

During federated training, where the coordinator initiates the next training round after aggregating updates from enough number of participants [18], it iterates over ②-④ in each round. Every few training rounds, federated testing is often used to detect whether the cut-off accuracy has been reached.

3.2 Oort Interface

Oort employs two distinct selectors that developers can access via a client library during FL training and testing.

Training selector. This selector aims to improve the time-to-accuracy performance of federated training. To this end, it captures the utility of clients in training, and efficiently explores and selects high-utility clients at runtime.

```
1  import Oort
2
3  def federated_model_training():
4    selector = Oort.create_training_selector(config)
5
6    # Train to target testing accuracy
7    while federated_model_testing() < target:
8
9      # Train 50 rounds before testing
10     for _ in range(50):
11       # Collect feedbacks of last round
12       feedbacks = engine.get_participant_feedback()
13
14       # Update the utility of clients
15       for clientId in feedbacks:
16         selector.update_client_util(
17                     clientId, feedbacks[clientId])
18
19       # Pick 100 high-utility participants
20       participants = selector.select_participant(100)
21       ... # Activate training on remote clients
```

Figure 6: ***Code snippet of Oort interaction during FL training.***

Figure 6 presents an example of how FL developers and frameworks interact with Oort during training. In each training round, Oort collects feedbacks from the engine driver, and updates the utility of individual clients (Line 15-17). Thereafter, it cherry-picks high-utility clients to feed the underlying execution (Line 20). We elaborate more on client utility and

the selection mechanism in Section 4.

Testing selector. This selector currently supports two types of selection criteria. When the individual client data characteristics (e.g., categorical distribution) are not provided, the testing selector determines the number of participants needed to cap the data deviation of participants from the global. Otherwise, it cherry-picks participants to serve the exact developer-specified requirements on data while minimizing the duration of testing. We elaborate more on selection for federated testing in Section 5.

4 Federated Model Training

In this section, we first outline the trade-off in selecting participants for FL training (§4.1), and then describe how Oort quantifies the client utility while respecting privacy (§4.2 and §4.3), how it selects high-utility clients at scale despite staleness in client utility as training evolves (§4.4).

4.1 Tradeoff Between Statistical and System Efficiency

Time-to-accuracy performance of FL training relies on two aspects: (i) *statistical efficiency*: the number of rounds taken to reach target accuracy; and (ii) *system efficiency*: the duration of each training round. The data stored on the client and the speed with which it can perform training determine its utility with respect to statistical and system efficiency, which we respectively refer to as statistical and system utility.

Due to the coupled nature of client data and system performance, cherry-picking participants for better time-to-accuracy performance requires us to jointly consider both forms of efficiency. We visualize the trade-off between these two with our breakdown experiments on the MobileNet model with OpenImage dataset (§7.2.1). As shown in Figure 7, while optimizing the system efficiency ("Opt-Sys. Efficiency") can reduce the duration of each round (e.g., picking the fastest clients), it can lead to more rounds than random selection as that client data may have already been overrepresented by other participants over past rounds. On the other hand, using a client with high statistical utility ("Opt-Stat. Efficiency") may lead to longer rounds if that client turns out to be the system bottleneck in global model aggregation.

Challenges. To improve time-to-accuracy performance, Oort aims to find a sweet spot in the trade-off by associating with every client its *utility* toward optimizing each form of efficiency (Figure 7). This leads to three challenges:

- In each round, how to determine which clients' data would help improve the statistical efficiency of training the most while respecting client privacy (§4.2)?
- How to take a client's system performance into account to optimize the global system efficiency (§4.3)?
- How to account for the fact that we don't have up-to-date utility values for all clients during training (§4.4)?

Next, we integrate system designs with ML principles to tackle the heterogeneity, the massive scale, the runtime uncertainties and privacy concerns of clients for practical FL.

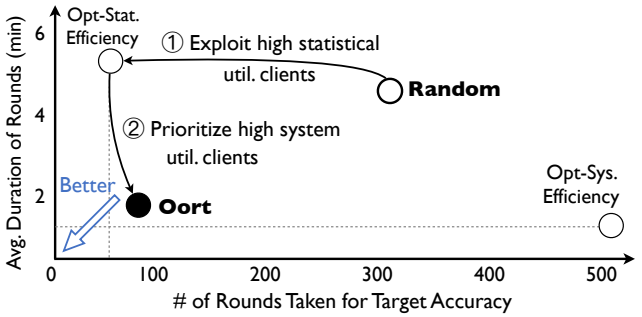


Figure 7: ***Existing FL training randomly selects participants, whereas Oort navigates the sweet point of statistical and system efficiency to optimize their circled area (i.e., time to accuracy). Numbers are from the MobileNet on OpenImage dataset (§7.2.1).***

4.2 Client Statistical Utility

An ideal design of statistical utility should be able to efficiently capture the client data utility toward improving model performance for various training tasks, and respect privacy.

To this end, we leverage importance sampling used in the ML literature [41, 81]. Say each client i has a bin B_i of training samples locally stored. Then, to improve the round-to-accuracy performance via importance sampling, the optimal solution would be to pick bin B_i with a probability proportional to its importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \| \nabla f(k) \|^2}$, where $\| \nabla f(k) \|$ is the L2-norm of the unique sample k's gradient $\nabla f(k)$ in bin B_i. Intuitively, this means selecting the bin with larger aggregate gradient norm across all of its samples.

However, taking this importance as the statistical utility is impractical, since it requires an extra time-consuming pass over the client data to generate the gradient norm of every sample,[5] and this gradient norm varies as the model updates.

To avoid extra cost, we introduce a pragmatic approximation of statistical utility instead. At its core, the gradient is derived by taking the derivative of training loss with respect to current model weights, wherein training loss measures the estimation error between model predictions and the ground truth. Our insight is that a larger gradient norm often attributes to a bigger loss [39]. Therefore, we define the statistical utility $U(i)$ of client i as $U(i) = |B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}$, where the training loss $Loss(k)$ of sample k is automatically generated during training with negligible collection overhead. As such, we consider clients that currently accumulate a bigger loss to be more important for future rounds.

Our statistical utility can capture the heterogeneous data utility across and within categories and samples for various tasks. We present the theoretical proof for its effectiveness over random sampling in our technical report [45], and empirically show its close-to-optimal performance (§7.2.2).

[5] ML models generate the training loss of each sample during training, but calculate the gradient of the mini-batch instead of individual samples.

How Oort respects privacy? Training loss measures the prediction confidence of a model without revealing the raw data and is often collected in real FL deployments [30, 77]. We further provide three ways to respect privacy. First, we rely on *aggregate* training loss, which is computed locally by the client across *all* of her samples without revealing the loss distribution of individual samples either. Second, when even the aggregate loss raises a privacy concern, clients can add noise to their loss value before uploading, similar to existing local differential privacy [27]. Third, we later show that Oort can flexibly accommodate other definitions of statistical utility used in our generic participant selection framework (§4.4). We provide detailed theoretical analyses for each strategy (e.g., using gradient norm of batches) of how Oort can respect privacy (e.g., amenable under noisy utility value) in our technical report [45], while empirically showing its superior performance even under noisy utility value (§7.2.3).

4.3 Trading off Statistical and System Efficiency

Simply selecting clients with high statistical utility can hamper the system efficiency. To reconcile the demand for both efficiencies, we should maximize the statistical utility we can achieve per unit time (i.e., the division of statistical utility and its round duration). As such, we formulate the utility of client i by associating her statistical utility with a global system utility in terms of the duration of each training round:

$$Util(i) = \underbrace{|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}}_{\text{Statistical utility } U(i)} \times \underbrace{\left(\frac{T}{t_i}\right)^{\mathbb{1}(T<t_i)\times\alpha}}_{\text{Global sys utility}} \quad (1)$$

where T is the developer-preferred duration of each round, t_i is the amount of time that client i takes to process the training, which has already been collected by today's coordinator from past rounds,[6] and $\mathbb{1}(x)$ is an indicator function that takes value 1 if x is true and 0 otherwise. This way, the utility of those clients who may be the bottleneck of the desired speed of current round will be penalized by a developer-specified factor α, but we do not reward the non-straggler clients because their completions do not impact the round duration.

This formulation assumes that all samples at a client are processed in that training round. Even if the estimated t_i for a client is greater than the desired round duration T, Oort might pick that client if the statistical utility outweighs its slow speed. Alternatively, if the developer wishes to cap every round at a certain duration [54], then either only clients with $t_i < T$ can be considered (e.g., by setting $\alpha \to \infty$) or a subset of a participant's samples can be processed [47, 63], and only the aggregate training loss of those trained data in that round is considered in measuring the statistical utility.

[6] We only care whether a client can complete by the expected duration T. So, a client can even mask its precise speed by deferring its report.

Navigating the trade-off. Determining the preferred round duration T in Equation (1), which strikes the trade-off between the statistical and system efficiency in aggregations, is non-trivial. Indeed, the total statistical utility (i.e., $\sum U(i)$) achieved by picking high utility clients can decrease round by round, because the training loss decreases as the model improves over time. If we persist in suppressing clients with high statistical utility but low system speed, the model may converge to suboptimal accuracy (§7.2.2).

To navigate the optimal trade-off – maximizing the total statistical utility achieved without greatly sacrificing the system efficiency – Oort employs a pacer to determine the preferred duration T at runtime. The intuition is that, when the accumulated statistical utility in the past rounds decreases, the pacer allows a larger $T \leftarrow T + \Delta$ by Δ to bargain with the statistical efficiency again. We elaborate more in Algorithm 1.

4.4 Adaptive Participant Selection

Given the above definition of client utility, we need to address the following practical concerns in order to select participants with the highest utility in each training round.

- *Scalability*: a client's utility can only be determined after it has participated in training; how to choose from clients at scale without having to try all clients once?
- *Staleness*: since not every client participates in every round, how to account for the change in a client's utility since its last participation?
- *Robustness*: how to be robust to outliers in the presence of corrupted clients (e.g., with noisy data)?

To tackle these challenges, we develop an exploration-exploitation strategy for participant selection (Algorithm 1).

Online exploration-exploitation of high-utility clients. Selecting participants out of numerous clients can be modeled as a multi-armed bandit problem, where each client is an "arm" of the bandit, and the utility obtained is the "reward" [14]. In contrast to sophisticated designs (e.g., reinforcement learning), the bandit model is scalable and flexible even when the solution space (e.g., number of clients) varies dramatically over time. Next, we adaptively balance the exploration and exploitation of different arms to maximize the long-term reward.

Similar to the bandit design, Oort efficiently explores potential participants under spatial variation, while intelligently exploiting observed high-utility participants under temporal variation. At the beginning of each selection round, Oort receives the feedback of the last training round, and updates the statistical utility and system performance of clients (Line 6). For the explored clients, Oort calculates their client utility and narrows down the selection by exploiting the high-utility participants (Line 9-15). Meanwhile, Oort samples $\varepsilon(\in [0, 1])$ fraction of participants to explore potential participants that had not been selected before (Line 16), which turns to full exploration as $\varepsilon \to 1$. Although we cannot learn the statistical

Input: Client set $\mathbb{C}$, sample size K, exploitation factor ε, pacer step Δ, step window W, penalty α
Output: Participant set $\mathbb{P}$

/* Initialize global variables. */
1 $\mathbb{E} \leftarrow \emptyset$; $\mathbb{U} \leftarrow \emptyset$ ▷ Explored clients and statistical utility.
2 $\mathbb{L} \leftarrow \emptyset$; $\mathbb{D} \leftarrow \emptyset$ ▷ Last involved round and duration.
3 $R \leftarrow 0$; $T \leftarrow \Delta$ ▷ Round counter and preferred round duration.

/* Participant selection for each round. */
4 **Function** SelectParticipant ($\mathbb{C}$, K, ε, T, α)
5 $Util \leftarrow \emptyset$; $R \leftarrow R+1$
/* Update and clip the feedback; blacklist outliers. */
6 UpdateWithFeedback($\mathbb{E}$, $\mathbb{U}$, $\mathbb{L}$, $\mathbb{D}$)
/* Pacer: Relaxes global system preference T if the statistical utility achieved decreases in last W rounds. */
7 **if** $\sum \mathbb{U}(R-2W:R-W) > \sum \mathbb{U}(R-W:R)$ **then**
8 $T \leftarrow T+\Delta$
/* Exploitation #1: Calculate client utility. */
9 **for** *client* $i \in \mathbb{E}$ **do**
10 $Util(i) \leftarrow \mathbb{U}(i) + \sqrt{\frac{0.1 \log R}{\mathbb{L}(i)}}$ ▷ Temporal uncertainty.
11 **if** $T < \mathbb{D}(i)$ **then** ▷ Global system utility.
12 $Util(i) \leftarrow Util(i) \times (\frac{T}{\mathbb{D}(i)})^{\alpha}$
/* Exploitation #2: admit clients with greater than c% of cut-off utility; then sample $(1-\varepsilon)K$ clients by utility. */
13 $Util \leftarrow$ SortAsc(Util)
14 $\mathbb{W} \leftarrow$ CutOffUtil($\mathbb{E}$, $c \times Util((1-\varepsilon) \times K)$)
15 $\mathbb{P} \leftarrow$ SampleByUtil($\mathbb{W}$, $Util$, $(1-\varepsilon) \times K$)
/* Exploration: sample unexplored clients by speed. */
16 $\mathbb{P} \leftarrow \mathbb{P} \cup$ SampleBySpeed($\mathbb{C}-\mathbb{E}$, $\varepsilon \times K$)
17 **return** $\mathbb{P}$

Alg. 1: Participant selection w/ exploration-exploitation.

utility of not-yet-tried clients, one can decide to prioritize the unexplored clients with faster system speed when possible (e.g., by inferring from device models), instead of performing random exploration (Line 16).

Exploitation under staleness in client utility. Oort employs two strategies to account for the dynamics in client utility over time. First, motivated by the confidence interval used to measure the uncertainty in bandit reward, we introduce an incentive term, which shares the same shape of the confidence in bandit solutions [37], to account for the staleness (Line 10), whereby we gradually increase the utility of a client if she has been overlooked for a long time. So those clients accumulating high utility since their last trial can still be repurposed again. Second, instead of picking clients with top-k utility deterministically, we allow a confidence interval c on the cut-off utility (95% by default in Line 13-14). Namely, we admit clients whose utility is greater than the c% of the top $((1-\varepsilon) \times K)$-th participant. Among this high-utility pool, Oort samples participants with probability proportional to their utility (Line 15). This adaptive exploitation mitigates the uncertainties in client utility by prioritizing participants opportunistically, thus relieving the need for accurate estimations of utility as we do not require the exact ordering among clients, while preserving a high quality as a whole.

```python
def federated_model_testing():
  selector = Oort.create_testing_selector()

  # Type 1: subset w/ < X deviation from the global
  participants = selector.select_by_deviation(
    dev_target, range_of_capacity, total_num_clients)

  # Provide individual client data characteristics
  selector.update_client_info(client_id, client_info)
  # Type 2: [5k, 5k] samples of category [i, j]
  participants = selector.select_by_category(
                 request_list, testing_config)
```

Figure 8: ***Key Oort APIs for supporting federated testing.***

Robust exploitation under outliers. Simply prioritizing high utility clients can be vulnerable to outliers in unfavorable settings. For example, corrupted clients may have noisy data, leading to high training loss, or even report arbitrarily high training loss intentionally. For robustness, Oort (i) removes the client in selection after she has been picked over a given number of rounds. This helps to remove the perceived outliers in terms of participation (Line 6); (ii) clips the utility value of a client by capping it to no more than an upper bound (e.g., 95% value in utility distributions). With probabilistic participant selection among the high-utility client pool (Line 15), the chance of selecting outliers is significantly decreased under the scale of clients in FL. We show that Oort outperforms existing mechanisms while being robust (§7.2.3).

Accommodation to diverse selection criteria. Our adaptive participant selection is generic for different utility definitions of diverse selection criteria. For example, developers may hope to reconcile their demand for time-to-accuracy efficiency and fairness, so that some clients are not underrepresented (e.g., more fair resource usage across clients) [40, 48]. Although developers may have various fairness criterion $fairness(\cdot)$, Oort can enforce their demands by replacing the current utility definition of client i with $(1-f) \times Util(i) + f \times fairness(i)$, where $f \in [0,1]$ and Algorithm 1 will naturally prioritize clients with the largest fairness demand as $f \to 1$. For example, $fairness(i) = max\_resource\_usage - resource\_usage(i)$ motivates fair resource usage for each client i. Note that existing participant selection provides no support for fairness, and we show that Oort can efficiently enforce diverse developer-preferred fairness while improving performance (§7.2.3).

5 Federated Model Testing

Enforcing developer-defined requirements on data distribution is a first-order goal in FL testing, whereas existing mechanisms lead to biased testing results (§2.3). In this section, we

elaborate on how Oort serves the two primary types of queries. As shown in Figure 8, we start with how Oort preserves the representativeness of testing set even without individual client data characteristics (§5.1), and how it efficiently enforces developer's testing criteria for specific data distribution when the individual information is provided (§5.2).

5.1 Preserving Data Representativeness

Learning the individual data characteristics (e.g., categorical distribution) can be too expensive or even prohibited [25, 64]. Without knowing data characteristics, the developer has to be conservative and selects many participants to gain more confidence for query *"a testing set with less than X% data deviation from the global"*, as selecting too few can lead to a biased testing result (§2.3). However, admitting too many may inflate the budget and/or take too long because of the system heterogeneity. Next, we show how Oort can enable guided participant selection by determining the number of participants needed to guarantee this deviation target.

We consider the deviation of the data formed by all participants from the global dataset (i.e., representative) using L1-distance, a popular distance metric in FL [34, 35, 57]. For category X, its L1-distance ($|\bar{X} - E[\bar{X}]|$) captures how the average number of samples of all participants (i.e., empirical value $\bar{X}$) deviates from that of all clients (i.e., expectation $E[\bar{X}]$). Note that the number of samples X_n that client n holds is independent across clients. Namely, the number of samples that one client holds will not be affected by the selection of any other clients at that time, so it can be viewed as a random instance sampled from the distribution of variable X.

Given the developer-specified tolerance ε on data deviation and confidence interval δ (95% by default [56]), our goal is to estimate the number of participants needed such that the deviation from the representative categorical distribution is bounded (i.e., $Pr[|\bar{X} - E[\bar{X}]| < \varepsilon] > \delta$). To this end, we formulate it as a problem of sampling stochastic variables, and apply the Hoeffding bound [16] to capture how this data deviation varies with different number of participants. We attach our theoretical results and proof in our technical report [45].

Estimating the number of participants to cap deviation. Even when the individual data characteristics are not available, the developer can specify her tolerance ε on the deviation from the global categorical distribution, whereby Oort outputs the number of participants needed to preserve this preference. To use our model, the developer needs to input the global range (i.e., global maximum - global minimum) of the number of samples that one client can hold, and the total number of clients. Learning this global information securely is well-established [23, 64], and the developer can assume a plausible limit (e.g., according to the capacity of device models) too.

Our model does not require any collection of the distribution of global or participant data. As a straw-man participant selection design, the developer can randomly distribute her model to this Oort-determined number of participants. After collecting results from this number of participants, she can confirm the representativeness of computed data.

5.2 Enforcing Diverse Data Distribution

When the individual data characteristics are provided (e.g., FL across enterprise AI cameras [35, 51]), Oort can enforce the exact data preference on specific categorical distribution, and improve the duration of testing by cherry-picking participants.

Satisfying queries like *"[5k, 5k] samples of class [x, y]"* can be viewed as a multi-dimensional bin covering problem, where a subset of data bins (i.e., participants) are selected to cover the requested quantity of data. For each category $i(\in I)$ of interest, the developer has preference p_i (preference constraint), and an upper limit B (referred to as budget) on how many participants she can have [15]. Each participant $n(\in N)$ can contribute n_i samples out of her capacity ${c_n}^i$ (capacity constraint). Given her compute speed s_n, the available bandwidth b_n and the size of data transfers d_n, we aim to minimize the duration of model testing:

$$\min \left\{ \max_{n \in N} \left(\frac{\sum_{i \in I} n_i}{s_n} + \frac{d_n}{b_n} \right) \right\} \quad \triangleright \text{ Minimize duration}$$

$$\text{s.t.} \quad \forall i \in I, \sum_{n \in N} n_i = p_i \quad \triangleright \text{ Preference Constraint}$$

$$\forall i \in I, \forall n \in N, n_i \leq c_n^i \quad \triangleright \text{ Capacity Constraint}$$

$$\forall i \in I, \sum_{n \in N} \mathbb{1}(n_i > 0) \leq B \quad \triangleright \text{ Budget Constraint}$$

The max-min formulation stems from the fact that testing completes after aggregating results from the last participant. While this mixed-integer linear programming (MILP) model provides high-quality solutions, it has prohibitively high computational complexity for large N.

Scalable participant selection. For better scalability, we present a greedy heuristic to scale down the search space of this strawman. We (1) first group a subset of feasible clients to satisfy the preference constraint. To this end, we iteratively add to our subset the client which has the most number of samples across all not-yet-satisfied categories, and deduct the preference constraint on each category by the corresponding capacity of this client. We stop this greedy grouping until the preference is met, or request a new budget if we exceed the budget; and (2) then optimize job duration with a simplified MILP among this subset of clients, wherein we have removed the budget constraint and reduced the search space of clients. We show that our heuristic can outperform the straw-man MILP model in terms of the end-to-end duration of model testing owing to its small overhead (§7.3.2).

6 Implementation

We have implemented Oort as a Python library, with 2617 lines of code, to friendly support FL developers. Oort provides simple APIs to abstract away the problem of participant selection, and developers can import Oort in their application

codebase and interact with FL engines (e.g., PySyft [7] or TensorFlow Federated [11]).

We have integrated Oort with PySyft. Oort operates on and updates its client metadata (e.g., data distribution or system performance) fed by the FL developer and PySyft at runtime. The metadata of each client in Oort is an object with a small memory footprint. Oort caches these objects in memory during executions and periodically backs them up to persistent storage. In case of failures, the execution driver will initiate a new Oort selector, and load the latest checkpoint to catch up. We employ Gurobi solver [5] to solve the MILP. The developer can also initiate a Oort application beyond coordinators to avoid resource contention. We use *xmlrpc* library to connect to the coordinator, and these updates will activate Oort to write these updates to its metastore. In the coordinator, we use the PySyft API *model.send(client_id)* to direct which client to run given the Oort decision, and *model.get(client_id)* to collect the feedback.

7 Evaluation

We evaluate Oort's effectiveness for four different ML models on four CV and NLP datasets. We organize our evaluation by the FL activities with the following key results.

FL training results summary:

- Oort outperforms existing random participant selection by 1.2×-14.1× in time-to-accuracy performance, while achieving 1.3%-9.8% better final model accuracy (§7.2.1).
- Oort achieves close-to-optimal model efficiency by adaptively striking the trade-off between statistical and system efficiency with different components (§7.2.2).
- Oort outperforms its counterpart over a wide range of parameters and different scales of experiments, while being robust to outliers (§7.2.3).

FL testing results summary:

- Oort can serve testing criteria on data deviation while reducing costs by bounding the number of participants needed without individual data characteristics (§7.3.1).
- With the individual information, Oort improves the testing duration by 4.7× w.r.t. Mixed Integer Linear Programming (MILP) solver, and is able to efficiently enforce developer preferences across millions of clients (§7.3.2).

7.1 Methodology

Experimental setup. Oort is designed to operate in large deployments with potentially millions of edge devices. However, such a deployment is not only prohibitively expensive, but also impractical to ensure the reproducibility of experiments. As such, we resort to a cluster with 68 NVIDIA Tesla P100 GPUs, and emulate up to 1300 participants in each round. We simulate real-world heterogeneous client system performance and data in both training and testing evaluations using an open-source FL benchmark [43]: (1) Heterogeneous device runtimes of different models, network throughput/connectivity, device model and availability are emulated using data from AI Benchmark [1] and Network Measurements on mobiles [6]; (2) We distribute each real dataset to clients following the corresponding raw placement (e.g., using <*authors_ID*> to allocate OpenImage), where client data can vary in quantities, distribution of outputs and input features; (3) The coordinator communicates with clients using the parameter server architecture. These follow the PySyft and real FL deployments. To mitigate stragglers, we employ the widely-used mechanism specified in real FL deployments [18], where we collect updates from the first K completed participants out of $1.3K$ participants in each round, and K is 100 by default. We report the simulated clock time of clients in evaluations.

Dataset	# of Clients	# of Samples
Google Speech [74]	2,618	105,829
OpenImage-Easy [3]	14,477	871,368
OpenImage [3]	14,477	1,672,231
StackOverflow [9]	315,902	135,818,730
Reddit [8]	1,660,820	351,523,459

Table 1: ***Statistics of the dataset in evaluations.***

Datasets and models. We run three categories of applications with four real-world datasets of different scales, and Table 1 reports the statistics of each dataset:

- *Speech Recognition*: the small-scale Google speech dataset [74]. We train a convolutional neural network model (ResNet-34 [31]) to recognize the command among 35 categories.
- *Image Classification*: the middle-scale OpenImage [3] dataset, with 1.5 million images spanning 600 categories, and a simpler dataset (OpenImage-Easy) with images from the most popular 60 categories. We train MobileNet [65] and ShuffleNet [80] models to classify the image.
- *Language Modeling*: the large-scale StackOverflow [9] and Reddit [8] dataset. We train next word predictions with Albert model [46] using the top-10k popular words.

These applications are widely used in real end-device applications [75], and these models are designed to be lightweight.

Parameters. The minibatch size of each participant is 16 in speech recognition, and 32 in other tasks. The initial learning rate for Albert model is 4e-5, and 0.04 for other models. These configurations are consistent with those reported in the literature [29]. In configuring the training selector, Oort uses the popular time-based exploration factor [14], where the initial exploration factor is 0.9, and decreased by a factor 0.98 after each round when it is larger than 0.2. The step window of pacer W is 20 rounds. We set the pacer step Δ in a way that it can cover the duration of next $W \times K$ clients in the descending order of explored clients' duration, and the straggler penalty α to 2. We remove a client from Oort's exploitation list once

Task	Dataset	Accuracy Target	Model	Speedup for Prox [47]			Speedup for YoGi [63]		
				Stats.	Sys.	Overall	Stats.	Sys.	Overall
Image Classification	OpenImage-Easy [3]	74.9%	MobileNet [65]	3.8×	3.2×	12.1×	2.4×	2.4×	5.7×
			ShuffleNet [80]	2.5×	3.5×	8.8×	1.9×	2.7×	5.1×
	OpenImage [3]	53.1%	MobileNet	4.2×	3.1×	13.0×	2.3×	1.5×	3.3×
			ShuffleNet	4.8×	2.9×	14.1×	1.8×	3.2×	5.8×
Language Modeling	Reddit [8]	39 perplexity	Albert [46]	1.3×	6.4×	8.4×	1.5×	4.9×	7.3×
	StackOverflow [9]	39 perplexity	Albert	2.1×	4.3×	9.1×	1.8×	4.4×	7.8×
Speech Recognition	Google Speech [74]	62.2%	ResNet-34 [31]	1.1×	1.1×	1.2×	1.2×	1.1×	1.3×

Table 2: ***Summary of improvements on time to accuracy.***[7] ***We tease apart the overall improvement with statistical and system ones, and take the highest accuracy that Prox can achieve as the target, which is moderate due to the high task complexity and lightweight models.***

she has been selected over 10 times.

Metrics. We care about the *time-to-accuracy* performance and *final model accuracy* of model training tasks on the testing set. For model testing, we measure the *end-to-end* testing duration, which consists of the computation overhead of the solution and the duration of actual computation.

For each experiment, we report the mean value over 5 runs, and error bars show the standard deviation.

7.2 FL Training Evaluation

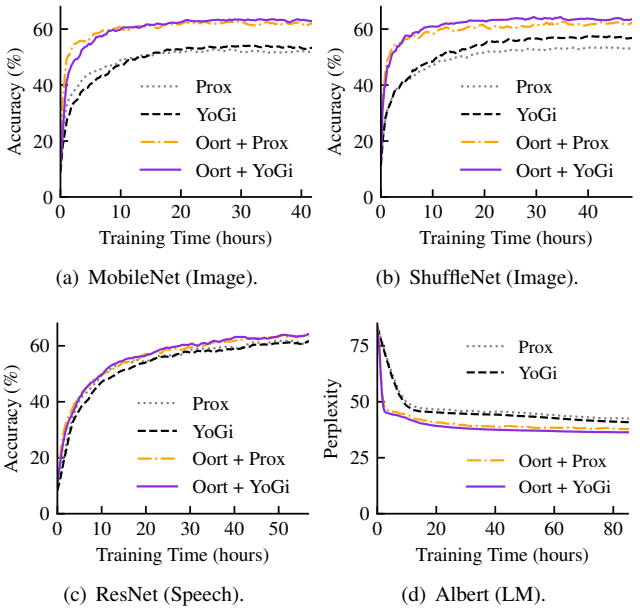
In this section, we evaluate Oort's performance on model training, and employ Prox [47] and YoGi [63]. We refer Prox as Prox running with existing random participant selection, and Prox + Oort is Prox running atop Oort. We use a similar denotation for YoGi. Note that Prox and YoGi optimize the statistical model efficiency for the given participants, while Oort cherry-picks participants to feed them.

7.2.1 End-to-End Performance

Table 2 summarizes the key time-to-accuracy performance of all datasets. In the rest of the evaluations, we report the ShuffleNet and MobileNet performance on OpenImage, and Albert performance on Reddit dataset for brevity. Figure 9 reports the timeline of training to achieve different accuracy.

Oort improves time-to-accuracy performance. We notice that Oort achieves large speedups to reach the target accuracy (Table 2). Oort reaches the target 3.3×-14.1× faster in terms of wall clock time on the middle-scale OpenImage dataset; speedup on the large-scale Reddit and StackOverflow dataset is 7.3×-9.1×. Understandably, these benefits decrease when the total number of clients is small, as shown on the small-scale Google Speech dataset (1.2×-1.3×).

[7] We set the target accuracy to be the highest achievable accuracy by all used strategies, which turns out to be Prox accuracy. Otherwise, some may never reach that target.



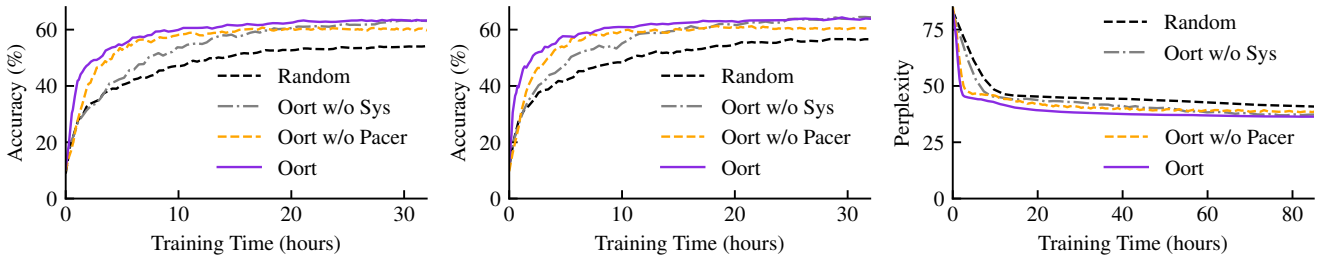
(a) MobileNet (Image). (b) ShuffleNet (Image).

(c) ResNet (Speech). (d) Albert (LM).

Figure 9: ***Time-to-Accuracy performance. A lower perplexity is better in the language modeling (LM) task.***

These time-to-accuracy improvements stem from the comparable benefits in statistical model efficiency and system efficiency (Table 2). Oort takes 1.8×-4.8× fewer training rounds on OpenImage dataset to reach the target accuracy, which is better than that of language modeling tasks (1.3×-2.1×). This is because real-life images often exhibit greater heterogeneity in data characteristics than the language dataset, whereas the large population of language datasets leaves a great potential to prioritize clients with faster system speed.

Oort improves final model accuracy. When the model converges, Oort achieves 6.6%-9.8% higher final accuracy on OpenImage dataset, and 3.1%-4.4% better perplexity on Reddit dataset (Figure 9). Again, this improvement on Google Speech dataset is smaller (1.3% for Prox and 2.2% for YoGi)



(a) MobileNet (Image). (b) ShuffleNet (Image). (c) Albert (LM).

Figure 10: ***Breakdown of Time-to-Accuracy performance with YoGi, when using different participant selection strategies.***

due to the small scale of clients. These improvements attribute to the exploitation of high statistical utility clients. Specifically, the statistical model accuracy is determined by the quality of global aggregation. Without cherry-picking participants in each round, clients with poor statistical model utility can dilute the quality of aggregation. As such, the model may converge to suboptimal performance. Instead, models running with Oort concentrate more on clients with high statistical utility, thus achieving better final accuracy.

7.2.2 Performance Breakdown

We next delve into the improvement on middle- and large-scale datasets, as they are closer to real FL deployments. We break down our knobs designed for striking the balance between statistical and system efficiency: (i) (*Oort w/o Pacer*): We disable the pacer that guides the aggregation efficiency. As such, it keeps suppressing low-speed clients, and the training can be restrained among low-utility but high-speed clients; (ii) (*Oort w/o Sys*): We further totally remove our benefits from system efficiency by setting α to 0, so Oort blindly prioritizes clients with high statistical utility. We take YoGi for analysis, because it outperforms Prox most of the time.

Breakdown of time-to-accuracy efficiency. Figure 10 reports the breakdown of time-to-accuracy performance, where Oort achieves comparable improvement from statistical and system optimizations. Taking Figure 10(b) as an example, (i) At the beginning of training, both Oort and (Oort w/o Pacer) improve the model accuracy quickly, because they penalize the utility of stragglers and select clients with higher statistical utility and system efficiency. In contrast, (Oort w/o Sys) only considers the statistical utility, resulting in longer rounds. (ii) As training evolves, the pacer in Oort gradually relaxes the constraints on system efficiency, and admits clients with relatively low speed but higher statistical utility, which ends up with the similar final accuracy of (Oort w/o Sys). However, (Oort w/o Pacer) relies on a fixed system constraint and suppresses valuable clients with high statistical utility but low speed, leading to suboptimal final accuracy.

Oort achieves close to upper-bound statistical performance. We consider an *upper-bound* statistical efficiency by creating a centralized case, where all data are evenly distributed to K participants. Using the target accuracy in Table 2, Oort can efficiently approach this upper bound by incorporating different components (Figure 11). Oort is within $2\times$ of the upper-bound to achieve the target accuracy, and (Oort w/o Sys) performs the best in statistical model efficiency, because (Oort w/o Sys) always grasps clients with higher statistical utility. However, it is suboptimal in our targeted time-to-accuracy performance because of ignoring the system efficiency. Moreover, by introducing the pacer, Oort achieves 2.4%-3.1% better accuracy than (Oort w/o Pacer), and is merely about 2.7%-3.3% worse than the upper-bound final model accuracy (Figure 12).

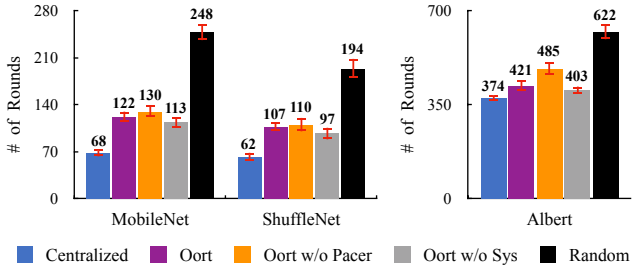


Figure 11: ***Number of rounds to reach the target accuracy.***

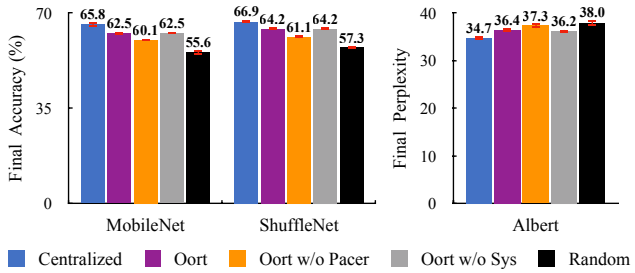
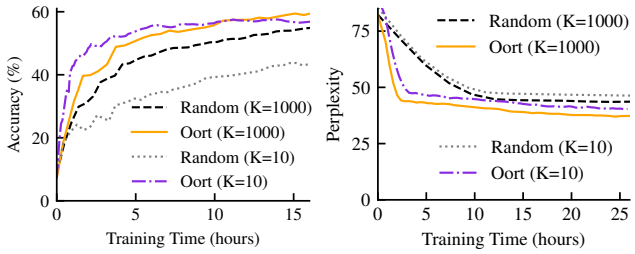


Figure 12: ***Breakdown of final model accuracy.***

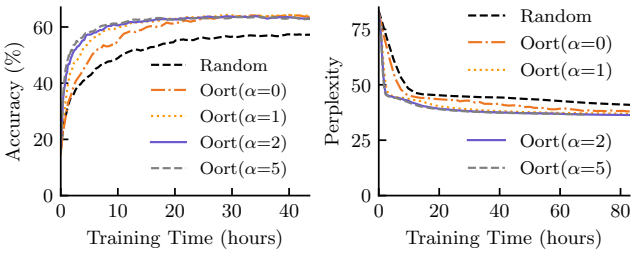
7.2.3 Sensitivity Analysis

Impact of number of participants K. We evaluate Oort across different scales of participants in each round, where we cut off the training after 200 rounds given the diminishing rewards. We observe that Oort improves time-to-accuracy efficiency across different number of participants (Figure 13), and having more participants in FL indeed receives diminishing rewards. This is because taking more participants (i) is



(a) ShuffleNet (Image). (b) Albert (LM).

Figure 13: ***Oort outperforms in different scales of participants.***



(a) ShuffleNet (Image). (b) Albert (LM).

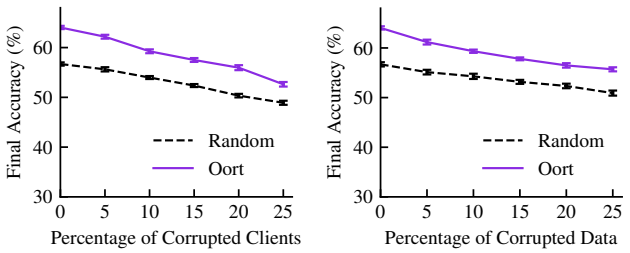
Figure 14: ***Oort improves performance across penalty factors.***

similar to having a large batch size, which is confirmed to be even negative to round-to-accuracy performance [50]; (ii) can lead to longer rounds due to stragglers when the number of clients is limited (e.g., K=1000 on OpenImage dataset).

Impact of penalty factor α on stragglers. Oort uses the penalty factor α to penalize the utility of stragglers in participant selection, whereby it adaptively prioritizes high system efficiency participants. Figure 14 shows that Oort outperforms its counterparts across different α. Note that Oort orchestrates its components to automatically navigate the best performance across parameters: larger α (i.e., overemphasizing system efficiency) drives the Pacer to relax the system constraint T more frequently to admit clients with higher statistical efficiency, and vice versa. As such, Oort achieves similar performance across all non-zero α.

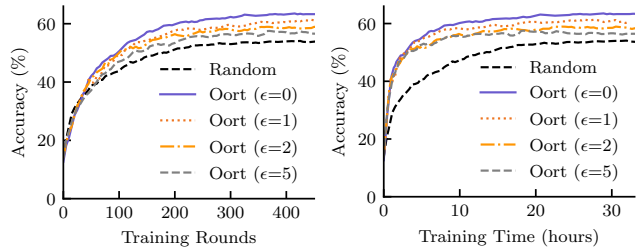
Impact of outliers. We investigate the robustness of Oort by introducing outliers manually. Following the popular adversarial ML setting [26], we randomly flip the ground-truth data labels of the OpenImage dataset to any other categories, resulting in artificially high utility. We consider two practical scenarios with the ShuffleNet model: (i) Corrupted clients: labels of all training samples on these clients are flipped (Figure 15(a)); (ii) Corrupted data: each client uniformly flips a subset of her training samples (Figure 15(b)). We notice Oort still outperforms across all degrees of corruption.

Impact of noisy utility. We next show the superior performance of Oort over its counterparts under noisy utility value. In this experiment, we add noise from the Gaussian distribution $Gaussian(0,\sigma^2)$, and investigate Oort's performance with different σ. Similar to differential FL [27], we

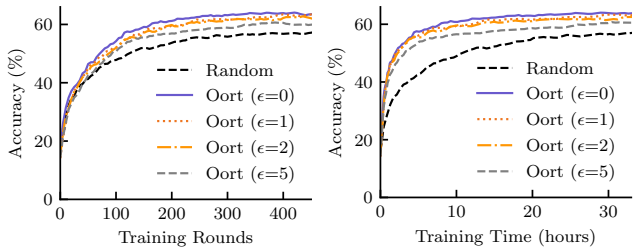


(a) Corrupted clients. (b) Corrupted data.

Figure 15: ***Oort still improves performance under outliers.***



(a) Round to accuracy (MobileNet). (b) Time to accuracy (MobileNet).



(c) Round to accuracy (ShuffleNet). (d) Time to accuracy (ShuffleNet).

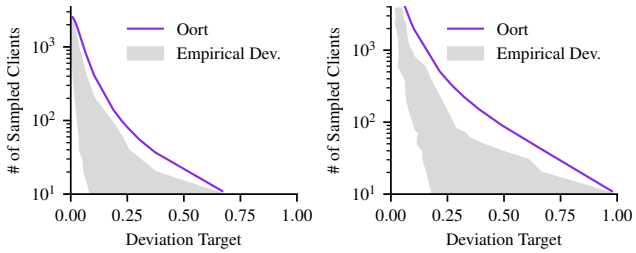
Figure 16: ***Oort improves performance even under noise.***

define $\sigma = \varepsilon \times Mean(real\_value)$, where $Mean(real\_value)$ is the average real value without noise. Note that we take this $real\_value$ as reference for the ease of presentations, and developers can refer to other values. As such, a large ε implies larger variance in noise, thus providing better privacy by disturbing the real value significantly. We report the statistical efficiency after adding noise to the statistical utility (Fig 16(a) and Fig 16(c)), as well as the time-to-accuracy performance (Fig 16(b) and Fig 16(d)). We observe that Oort still improves performance across different amount of noise, and is robust even when the noise is large (e.g., $\varepsilon = 5$ is often considered to be very large noise [12]).

Oort can respect developer-preferred fairness. In this experiment, we expect all clients should have participated training with the same number of rounds (Table 3), implying a fair resource usage [40]. We train ShuffleNet model on OpenImage dataset with YoGi. To this end, we sweep different knobs f to accommodate the developer demands for the time-to-accuracy efficiency and fairness. Namely, we replace the current utility definition of client i with $(1-f) \times Util(i) + f \times fairness(i)$, where $fairness(i) =$

Strategy	TTA (h)	Final Accuracy (%)	Var. (Rounds)
Random	36.3	57.3	0.39
$f = 0$	5.8	64.2	6.52
$f = 0.25$	6.1	62.4	5.1
$f = 0.5$	13.1	59.7	2.03
$f = 0.75$	25.4	58.6	0.65
$f = 1$	30.1	57.2	0.31

Table 3: ***Oort improves time to accuracy (TTA) across different fairness knobs (f). Random reports the performance of random participant selection. The variance of rounds reports how fairness is enforced in terms of the number of participating rounds across clients. A smaller variance implies better fairness.***



(a) Google Speech. (b) Reddit.

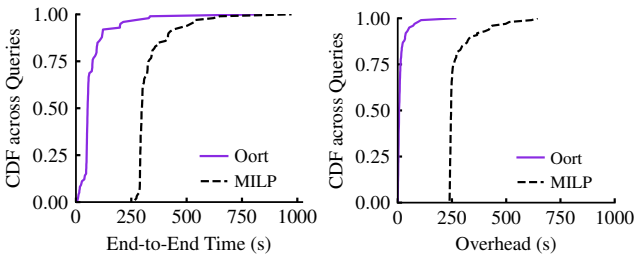
Figure 17: ***Oort can cap data deviation for all targets. Shadow indicates the empirical [min, max] range of the x-axis values over 1000 runs given the y-axis input.***

$max\_resource\_usage - resource\_usage(i)$. Understandably, time-to-accuracy efficiency significantly decreases as $f \to 1$, since we gradually end up with round-robin participant selection, totally ignoring the utility of clients. Note that Oort still achieves better time-to-accuracy even when $f \to 1$ as it prioritizes high system utility clients at the beginning of training, thus achieving shorter rounds. Moreover, Oort can enforce different fairness preferences while improving efficiency across fairness knobs.

7.3 FL Testing Evaluation

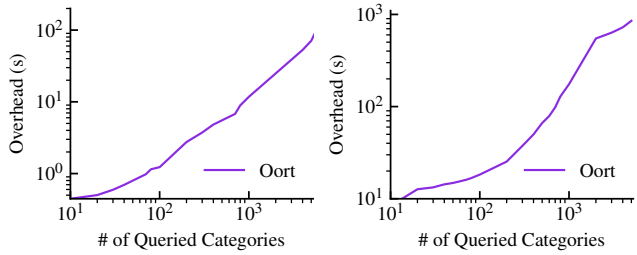
7.3.1 Preserving Data Representativeness

Oort can cap data deviation. Figure 17 reports Oort's performance on serving different deviation targets, with respect to the global distribution. We sweep the number of selected clients from 10 to 4k, and randomly select each given number of participants over 1k times to empirically search their possible deviation. We notice that for a given deviation target, (i) different workloads require distinct number of participants. For example, to meet the target of 0.05 divergence, the Speech dataset uses 6× less participants than the Reddit attributing to its smaller heterogeneity (e.g., tighter range of the number of samples); (ii) with the Oort-determined number of participants, no empirical deviation exceeds the target, showing the effectiveness of Oort in satisfying the deviation target, whereby Oort reduces the cost of expanding participant set arbitrarily and improves the testing duration.



(a) OpenImage (Testing duration). (b) OpenImage (Overhead).

Figure 18: ***Oort outperforms MILP in clairvoyant FL testing.***



(a) StackOverflow (0.3M clients). (b) Reddit (1.6M clients).

Figure 19: ***Oort scales to millions of clients, while MILP did not complete on any query.***

7.3.2 Enforcing Diverse Data Distribution

Oort outperforms MILP. We start with the middle-scale OpenImage dataset and compare the end-to-end testing duration of Oort and MILP. Here, we generate 200 queries using the form "Give me X representative samples", where we sweep X from 4k to 200k and budget B from 100 participants to 5k participants. We report the validation time of MobileNet on participants selected by these strategies.

Figure 18(a) shows the end-to-end testing duration. We observe Oort outperforms MILP by 4.7× on average. This is because Oort suffers little computation overhead by greedily reducing the search space of MILP. As shown in Figure 18(b), MILP takes 274 seconds on average to complete the participant selection, while Oort only takes 15 seconds.

Oort is scalable. We further investigate Oort's performance on the large-scale StackOverflow and Reddit dataset with millions of clients, where we take 1% of the global data as the requirement, and sweep the number of interested categories from 1 to 5k. Figure 19 shows even though we gradually magnify the search space of participant selection by introducing more categories, Oort can serve our requirement in a few minutes at the scale of millions of clients, while MILP fails to generate the solution decision for any query.

8 Related Work

Federated Learning Federated learning [40] is a distributed machine learning paradigm in a network of end devices, wherein Prox [47] and YoGi [63] are state-of-the-art optimizations in tackling data heterogeneity. Recent efforts in FL have been focusing on improving communication effi-

ciency [33,54] or compression schemes [13], ensuring privacy by leveraging multi-party computation (MPC) [19] and differential privacy [27], or tackling heterogeneity by reinventing ML algorithms [48,72]. However, they underperform in FL because of the suboptimal participant selection they rely on, and lack systems supports for developers to specify their participant selection criteria.

Datacenter Machine Learning Distributed ML in datacenters has been well-studied [36,58,60], wherein they assume relatively homogeneous data and workers [28,52]. While developer requirements and models can still be the same, the heterogeneity of client system performance and data distribution makes FL much more challenging. We aim at enabling them in FL. To accelerate traditional model training, some techniques bring up importance sampling to prioritize important training samples in selecting mini-batches for training [39,41,81]. While bearing some resemblance in prioritizing data, Oort adaptively considers both statistical and system efficiency in formulating the client utility at scale.

Geo-distributed Data Analytics Federated data analytics has been a topic of interest in geo-distributed storage [69] and data processing systems [44,79] that attempt to reduce latency [70] and/or save bandwidth [42,61,71]. Gaia [33] reduces network traffics for model training across datacenters, while Sol [44] enables generic federated computation on data with sub-second latency in the execution layer. These work back up Oort with cross-layer system support, whereas Oort cherry-picks participants before execution.

Privacy-preserving Data Analytics To gather sensitive statistics from user devices, several differentially private systems add noise to user inputs locally to ensure privacy [25], but this can reduce the accuracy. Some assume a trusted third party, which only adds noise to the aggregated raw inputs [17], or use MPC to enable global differential privacy without a trusted party [64]. While our goal is not to address the security and privacy issue in these solutions, Oort enables informed participant selection by leveraging the information already available in today's FL, and can reconcile with them (e.g., to deliver improvement under outliers while respecting privacy).

9 Conclusion

While today's FL efforts have been optimizing the statistical model and system efficiency by reinventing traditional ML designs, the participant selection mechanisms they rely on underperform for federated training and testing, and fail to enforce diverse data selection criteria. In this paper, we present Oort to enable guided participant selection for FL developers. Compared to existing mechanisms, Oort achieves large speedups in time-to-accuracy performance for federated training by picking clients with high statistical and system utility, and it allows developers to specify their selection criteria on data while efficiently serving their requirements on data distribution during testing even at the scale of millions of clients. The artifacts of Oort are available at https://github.com/SymbioticLab/Oort.

Acknowledgments

Special thanks go to the entire ConFlux team and CloudLab team for making Oort experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Gennady Pekhimenko, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1900665 and CNS-1909067.

References

[1] AI Benchmark: All About Deep Learning on Smartphones. http://ai-benchmark.com/ranking_deeplearning_detailed.html.

[2] Federated AI Technology Enabler. https://www.fedai.org/.

[3] Google Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.

[4] Google's Sundar Pichai: Privacy Should Not Be a Luxury Good. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html.

[5] Gurobi. https://www.gurobi.com/.

[6] MobiPerf. https://www.measurementlab.net/tests/mobiperf/.

[7] PySyft. https://github.com/OpenMined/PySyft.

[8] Reddit Comment Data. https://files.pushshift.io/reddit/comments/.

[9] Stack Overflow Data. https://cloud.google.com/bigquery/public-data/stackoverflow.

[10] Stanford Puffer. https://puffer.stanford.edu/.

[11] TensorFlow Federated. https://www.tensorflow.org/federated.

[12] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, 2016.

[13] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient sgd via gradient quantization and encoding. In *NeurIPS*, 2017.

[14] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, 2002.

[15] Sean Augenstein, H Brendan McMahan, Daniel Ramage, Swaroop Ramaswamy, Peter Kairouz, Mingqing Chen, Rajiv Mathews, et al. Generative models for effective ML on private, decentralized datasets. In *ICLR*, 2020.

[16] Rémi Bardenet and Odalric-Ambrym Maillard. *Concentration inequalities for sampling without replacement*. Bernoulli Society for Mathematical Statistics and Probability, 2015.

[17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[18] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. In *MLSys*, 2019.

[19] Keith Bonawitz, Vladimir Ivanov, and et al. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, 2017.

[20] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data validation for machine learning. In *MLSys*, 2019.

[21] Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. Federated learning of out-of-vocabulary words. In *arxiv.org/abs/1903.10635*, 2019.

[22] Mingqing Chen, Ananda Theertha Suresh, Rajiv Mathews, Adeline Wong, Cyril Allauzen, Francoise Beaufays, and Michael Riley. Federated learning of n-gram language models. In *ACL*, 2019.

[23] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[24] Apple Differential Privacy Team. Learning with privacy at scale. In *Apple Machine Learning Journal*, 2017.

[25] Ulfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[26] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[27] Robin C. Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. In *NeuIPS*, 2017.

[28] 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 *NSDI*, 2019.

[29] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. In *arxiv.org/abs/1811.03604*, 2018.

[30] Florian Hartmann, Sunah Suh, Arkadiusz Komarzewski, Tim D. Smith, and Ilana Segall. Federated learning for ranking browser history suggestions. In *arxiv.org/abs/1911.11807*, 2019.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[32] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[33] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[34] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip B. Gibbons. The Non-IID data quagmire of decentralized machine learning. In *ICML*, 2020.

[35] Harry Hsu, Hang Qi, and Matthew Brown. Federated visual classification with real-world data distribution. In *ECCV*, 2020.

[36] 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 *SOSP*, 2019.

[37] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM*, 2016.

[38] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. Networked cameras are the new big data clusters. In *HotEdgeVideo*, 2019.

[39] Tyler B. Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. In *NeurIPS*, 2018.

[40] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. In *Foundations and Trends in Machine Learning*, 2021.

[41] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

[42] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for inter-cloud transfers? In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[43] Fan Lai, Yinwei Dai, Xiangfeng Zhu, and Mosharaf Chowdhury. FedScale: Benchmarking model and system performance of federated learning. In *arxiv.org/abs/2105.11367*, 2021.

[44] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In *NSDI*, 2020.

[45] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *arxiv.org/abs/2010.06081*, 2020.

[46] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR*, 2020.

[47] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *MLSys*, 2020.

[48] Tian Li, Manzil Zaheer, Ahmad Beirami, and Virginia Smith. Fair resource allocation in federated learning. In *ICLR*, 2020.

[49] Wenqi Li, Fausto Milletari, and Daguang Xu. Privacy-preserving federated brain tumour segmentation. In *Machine Learning in Medical Imaging*, 2019.

[50] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local SGD. In *ICLR*, 2020.

[51] Jiahuan Luo, Xueyang Wu, Yun Luo, Anbu Huang, Yunfeng Huang, Yang Liu, and Qiang Yang. Real-world image datasets for federated learning. In *arxiv.org/abs/1910.11089*, 2019.

[52] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[53] Yishay Mansour, Mehryar Mohri, Jae Ro, and Ananda Suresh. Three approaches for personalization with applications to federated learning. In *arxiv.org/abs/2002.10619*, 2020.

[54] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[55] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[56] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[57] Mehryar Mohri, Gary Sivek, and Ananda Theertha Suresh. Agnostic federated learning. In *ICML*, 2019.

[58] 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 *SOSP*, 2019.

[59] Xingchao Peng, Zijun Huang, Yizhe Zhu, and Kate Saenko. Federated adversarial domain adaptation. In *ICLR*, 2020.

[60] 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 *SOSP*, 2019.

[61] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low latency Geo-distributed data analytics. In *SIGCOMM*, 2015.

[62] Swaroop Ramaswamy, Om Thakkar, Rajiv Mathews, Galen Andrew, H. Brendan McMahan, and Françoise Beaufays. Training production language models without memorizing user data. In *arxiv.org/abs/2009.10031*, 2020.

[63] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. In *ICLR*, 2021.

[64] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *SOSP*, 2019.

[65] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[66] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of GDPR on database systems. In *VLDB*, 2020.

[67] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. The seven sins of personal-data processing systems under GDPR. In *HotCloud*, 2019.

[68] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In *ICML*, 2017.

[69] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *NSDI*, 2020.

[70] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[71] Ashish Vulimiri, Carlo Curino, B Godfrey, J Padhye, and G Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[72] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. In *MLSys*, 2019.

[73] Kangkang Wang, Rajiv Mathews, Chloe Kiddon, Hubert Eichner, Francoise Beaufays, and Daniel Ramage. Federated evaluation of on-device personalization. In *arxiv.org/abs/1910.10252*, 2019.

[74] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. In *arxiv.org/abs/1804.03209*, 2018.

[75] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *WWW*, 2019.

[76] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *NSDI*, 2020.

[77] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving Google keyboard query suggestions. In *arxiv.org/abs/1812.02903*, 2018.

[78] Felix X. Yu, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. Federated learning with only positive labels. In *ICML*, 2020.

[79] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. AWStream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

[80] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[81] Peilin Zhao and Tong Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *ICML*, 2015.

PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections

Haojie Wang Jidong Zhai Mingyu Gao Zixuan Ma Shizhi Tang
Liyan Zheng Yuanzhi Li[†] Kaiyuan Rong Yuanyong Chen Zhihao Jia[†‡]
Tsinghua University *Carnegie Mellon University*[†] *Facebook*[‡]

Abstract

High-performance tensor programs are critical for efficiently deploying deep neural network (DNN) models in real-world tasks. Existing frameworks optimize tensor programs by applying fully equivalent transformations, which maintain equivalence on every element of output tensors. This approach misses possible optimization opportunities as transformations that only preserve equivalence on subsets of the output tensors are excluded.

We propose PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET discovers and applies program transformations that improve computation efficiency but only maintain partial functional equivalence. PET then automatically corrects results to restore full equivalence. We develop rigorous theoretical foundations to simplify equivalence examination and correction for partially equivalent transformations, and design an efficient search algorithm to quickly discover highly optimized programs by combining fully and partially equivalent optimizations at the tensor, operator, and graph levels. Our evaluation shows that PET outperforms existing systems by up to 2.5×, by unlocking previously missed opportunities from partially equivalent transformations.

1 Introduction

Existing deep neural network (DNN) frameworks represent DNN computations as *tensor programs*, which are direct acyclic computation graphs describing the operations applied to a set of tensors (i.e., *n*-dimensional arrays). The operators in tensor programs are mostly linear algebra computations such as matrix multiplication and convolution. Although tensor programs are specified based on the high-level insights of today's DNN algorithms, such constructions do not necessarily offer the best runtime performance. Current practice to optimize tensor programs in existing DNN frameworks is to leverage *program transformations*, each of which identifies a subprogram that matches a specific pattern and replaces it with another subprogram that offers improved performance.

To preserve the statistical behavior of DNN models, existing frameworks only consider *fully equivalent* program transformations, where the new subprogram is mathematically equivalent to the original subprogram for *arbitrary* inputs. For example, TensorFlow, PyTorch, TensorRT, TVM, and Ansor all use rule-based optimization strategies that directly apply manually designed program transformations whenever applicable [3, 6, 26, 32, 34]. TASO automatically generates and verifies transformations by taking operator specifications as inputs, but is still limited to fully equivalent transformations [15].

Despite the wide use of equivalent program transformations in conventional compilers and modern DNN frameworks, they only exhibit limited opportunities for performance optimization, especially for tensor programs. Unlike traditional programs whose primitives are scalars or simple arrays of scalars, tensor programs operate on high-dimensional tensors with up to millions of elements. Many transformations can improve the runtime performance of a tensor program but do not preserve full equivalence on *all* elements of the output tensors. We call such transformations *partially equivalent*. Examples of performance-optimizing partially equivalent transformations include (1) changing the shape or linearization ordering of input tensors to improve computational efficiency, (2) replacing less efficient operators with more optimized operators with similar mathematical behavior, and (3) transforming the graph structure of a program to enable subsequent performance optimizations.

Partially equivalent transformations, despite their high potential, are not exploited in existing DNN frameworks due to several challenges. First, directly applying partially equivalent transformations would violate the functional equivalence to an input program and potentially decrease the model accuracy. It is necessary to correct any non-equivalent regions of output tensors, to preserve transparency to higher-level algorithms. However, quickly *examining equivalence* to identify these regions and effectively generating the required *correction kernels* are difficult tasks. Second, when partially equivalent transformations are applied, the design space is substantially

enlarged compared to existing frameworks under equivalence constraint. Theoretically, any program transformation, regardless of how different the result is from the original one, becomes a potential candidate. The *generation algorithm* for partially equivalent transformations should carefully manage its computational complexity. The *optimizer* must balance the benefits and overhead and be able to combine fully and partially equivalent transformations to obtain performant tensor programs.

In this paper, we explore a radically different approach to optimize tensor programs, by exploiting partially equivalent transformations. We develop rigorous theorems that simplify equivalence examination and correction kernel generation, allowing us to easily restore functional equivalence and provably preserve the DNN models' statistical behavior. With a significantly larger search space of program optimizations that includes both fully and partially equivalent transformations, our approach can discover highly optimized tensor programs that existing approaches miss. Based on these techniques, we propose PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET consists of three main components:

Mutation generator. To discover partially equivalent transformations automatically for an input subprogram, PET uses a *mutation generator* to construct potential program *mutants*. Each mutant takes the same input tensors as in the original subprogram and produces output tensors with the same shapes. This ensures that a mutant can replace the input subprogram and therefore constitutes a potential transformation.

Mutation corrector. The generated mutants of an input subprogram may produce different results on some regions of the output tensors, thus affecting the model accuracy. To preserve its statistical behavior, PET's *mutation corrector* examines the equivalence between an input subprogram and its mutant and automatically generates *correction kernels*. These are subsequently applied to the output tensors to maintain an end-to-end equivalence to the input subprogram. To reduce the overhead and heterogeneity introduced by the correction kernels, PET opportunistically *fuses* the correction kernels with other tensor computation kernels.

Examining and correcting a partially equivalent transformation is difficult, since the output tensors of a program include up to millions of elements, and each one must be verified against a large number of input elements. A key contribution of PET is a set of rigorous theoretical foundations that significantly simplify this verification process. Rather than examining program equivalence for *all* positions in the output tensors, PET needs to test only a few representative positions.

Program optimizer. PET uses a *program optimizer* to identify mutant candidates with high performance, by effectively balancing the benefits from using better mutants and the overheads of extra correction kernels. We first split an arbitrarily large input program into multiple small subprograms at the positions of non-linear operators. Each subprogram then contains only linear operators and can be independently mutated. We support mutations on various subsets of operators in the subprogram, and can iteratively apply mutations to obtain mutants that are more complex. Finally, we apply a series of post-optimizations across subprogram boundaries, including redundancy elimination and operator fusion.

We evaluate PET on five real-world DNN models. Even for common and heavily optimized models in existing frameworks such as Resnet-18 [14], PET can still improve the performance by 1.2×. For new models such as CSRNet [20] and BERT [12], PET is up to 2.5× faster than the state-of-the-art frameworks. The significant performance improvement is enabled by combining fully and partially equivalent transformations at the tensor, operator, and graph levels.

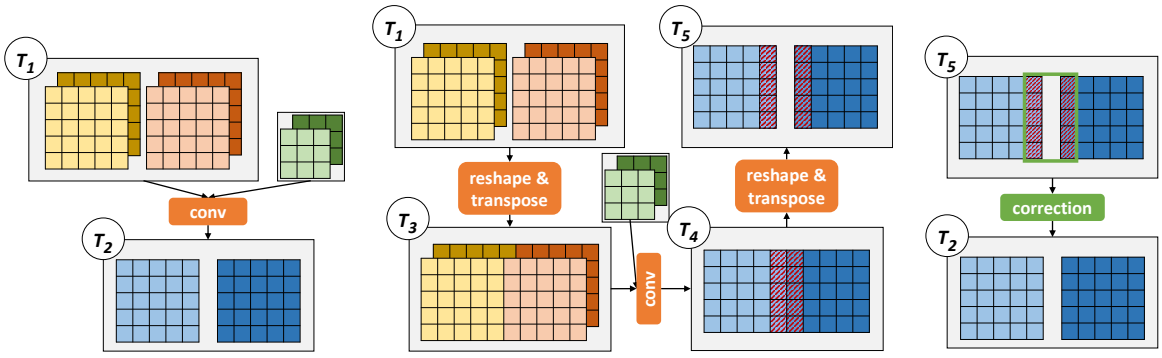
This paper makes the following contributions.

- We present the first attempt in tensor program optimization to exploit partially equivalent transformations with automated corrections. We explore a significantly larger search space than existing DNN frameworks.
- We develop rigorous theoretical foundations that simplify the equivalence examination and correction kernel generation, making it practical to preserve statistical behavior even with partially equivalent transformations.
- We propose efficient generation and optimization approaches to explore the large design space automatically with both fully and partially equivalent transformations.
- We implement the above techniques into an end-to-end framework, PET, and achieve up to 2.5× speedup compared to state-of-the-art frameworks.

2 Background and Motivation

To generate high-performance tensor programs, a common form of optimization in existing DNN frameworks (e.g., TensorFlow [3], TensorRT [32], and TVM [6]) is fully equivalent transformations that improve the performance of a tensor program while preserving its mathematical equivalence. Examples of current fully equivalent transformations include operator fusion [2, 6], layout transformations [18], and automated generation of graph substitutions [15]. Though effective at improving performance, fully equivalent transformations explore only a limited space of program optimizations.

In contrast, Figure 1 shows an example of a partially equivalent transformation for a convolution operator. It concatenates two individual images into a larger one along the width dimension to improve performance. This is because a larger width, which is typically the innermost dimension for convolution on modern accelerators like GPUs, provides more parallelism and improves computation locality. However, the new program after this transformation (shown in Figure 1(b)) produces different results on a sub-region of the output tensor along the boundary of the concatenation (shown as the shaded boxes in Figure 1(b)), resulting in partial non-equivalence.



(a) Input program. (b) A partially equivalent transformation. (c) Correcting results.

Figure 1: A partially equivalent transformation that improves the performance of convolution by manipulating tensor shape and linearization. The shaded boxes in (b) highlight non-equivalent elements between two programs in the transformation. The correction kernel in (c) is applied to these elements to recover the functional equivalence of the input program.

In addition to the above example that optimizes a tensor program by changing the shape and linearization of its tensors, partially equivalent transformations also include replacing less efficient operators with more optimized ones with similar semantics, and modifying the graph structure of a tensor program to enable additional optimizations. We provide more such examples in §4.2 and evaluate them in §8.3.

Although partially equivalent transformations exhibit high potential for performance improvement, they are not considered in current DNN frameworks due to their possible impact on model accuracy. Manually implementing such partially equivalent transformations is prohibitive. First, it requires evaluating a large amount of potential partially equivalent transformations to discover promising ones. Second, to apply partially equivalent transformations while preserving model accuracy, we need correction kernels to fix the results for non-equivalent parts (see Figure 1(c)). Overall, more *automated* approaches are needed to discover performance-optimizing partially equivalent transformations and correct the results, which are the main focus of this work.

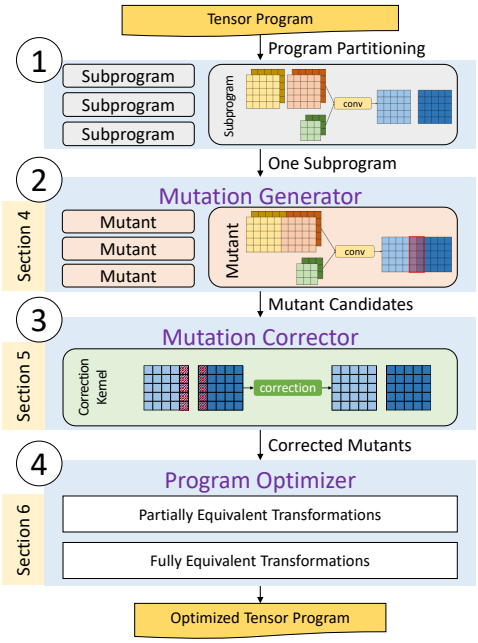


Figure 2: PET overview.

3 Design Overview

PET is the first framework to optimize tensor programs by exploiting partially equivalent transformations and correcting their results automatically. To realize this, PET leverages the multi-linearity of tensor programs.

Multi-linear tensor programs (MLTPs). We first define multi-linear tensor operators. An operator op with n input tensors $I_1,\ldots,I_n$ is *multi-linear* if op is linear to all inputs I_k:

$$op(I_1,\ldots,I_{k-1},X,\ldots,I_n)+op(I_1,\ldots,I_{k-1},Y,\ldots,I_n)$$
$$=op(I_1,\ldots,I_{k-1},X+Y,\ldots,I_n)$$
$$\alpha\cdot op(I_1,\ldots,I_{k-1},X,\ldots,I_n)=op(I_1,\ldots,I_{k-1},\alpha\cdot X,\ldots,I_n)$$

where X and Y are arbitrary tensors with the same shape as I_k, and α is an arbitrary scalar. DNN computation generally consists of multi-linear tensor operators (e.g., matrix multiplication, convolution) and element-wise non-linear operators (e.g., ReLU [23] and sigmoid). The linear operators consume the majority of the computation time, due to their high computational complexity. A program $\mathcal{P}$ is a *multi-linear tensor program* (MLTP) if all operators $op \in \mathcal{P}$ are multi-linear.

PET overview. Figure 2 shows an overview of PET. The input to PET is a tensor program to be optimized. Similar to prior work [6,34], PET first splits an input program into smaller subprograms to reduce the exploration space of each subprogram without sacrificing performance improvement opportunities. For each subprogram, PET's *mutation generator* discovers partially equivalent transformations by generating possible *mutants* for MLTPs in the subprogram. Each mutant has the

Table 1: Multi-linear tensor operators used in PET.

Operator	Description
`add`	Element-wise addition
`mul`	Element-wise multiplication
`conv`	Convolution
`groupconv`	Grouped convolution
`dilatedconv`	Dilated convolution
`batchnorm`	Batch normalization
`avgpool`	Average pooling
`matmul`	Matrix multiplication
`batchmatmul`	Batch matrix multiplication
`concat`	Concatenate multiple tensors
`split`	Split a tensor into multiple tensors
`transpose`	Transpose a tensor's dimensions
`reshape`	Decouple/combine a tensor's dimensions

same input and output shapes as the original MLTPs, thus constitutes a partially equivalent transformation (§4).

To maintain the end-to-end equivalence to an input program, PET's *mutation corrector* examines the equivalence between a mutant and its original MLTP, and automatically generates *correction kernels* to fix the outputs of the mutant. PET leverages rigorous theoretical foundations to simplify such challenging tasks (§5).

The corrected mutants are sent to PET's *program optimizer*, which combines existing fully equivalent transformations with partially equivalent ones to construct a comprehensive search space of program optimizations. The optimizer evaluates a rich set of mutants for each subprogram and applies post-optimizations across their boundaries, in order to discover highly optimized candidates in the search space (§6).

4 Mutation Generator

This section describes the mutation generator in PET, which takes an MLTP as input and automatically generates possible *mutants* to replace the input MLTP. The generation algorithm discovers valid mutants up to a certain size. Each generated mutant does not necessarily preserve mathematical equivalence to the input program on the entire output tensors. To restore functional equivalence, the mutation corrector (§5) automatically generates correction kernels.

4.1 Mutation Generation Algorithm

We call an MLTP $\mathcal{P}_1$ a *mutant* of another MLTP $\mathcal{P}_0$ if $\mathcal{P}_1$ and $\mathcal{P}_0$ have the same number of inputs (and outputs) and each input (and output) has the same shape. The computations of $\mathcal{P}_0$ and $\mathcal{P}_1$ are not necessarily equivalent. Intuitively, if $\mathcal{P}_0$ is a subprogram in a tensor program, then replacing $\mathcal{P}_0$ with $\mathcal{P}_1$ yields a valid but potentially non-equivalent tensor program.

For a given MLTP $\mathcal{P}_0$, PET generates potential mutants of $\mathcal{P}_0$ using a given set of multi-linear operators O as the basic building blocks. Table 1 lists the operators used in our evaluation. The list covers a variety of commonly used tensor operators, including compute-intensive operators (`conv`, `matmul`, etc.), element-wise operators (`add`, `mul`, etc.), and tensor manipulation (`split`, `transpose`, etc.). This set can also be extended to include new DNN operators.

Algorithm 1 MLTP mutation generation algorithm.

Input: A set of operators O; an input MLTP $\mathcal{P}_0$
Output: A set of valid program mutants $\mathcal{M}$ for $\mathcal{P}_0$
I_0 = the set of input tensors in $\mathcal{P}_0$
$\mathcal{M} = \varnothing$
BUILD(1, $\varnothing$, I_0)
// Depth-first search to construct mutants
function BUILD(n, $\mathcal{P}$, I)
  if $\mathcal{P}$ and $\mathcal{P}_0$ have the same input/output shapes **then**
    $\mathcal{M} = \mathcal{M} + \{\mathcal{P}\}$
  if $n <$ *depth* **then**
    for $op \in O$ **do**
      for $i \in I$ and i is a valid input to op **do**
        Add operator op into program $\mathcal{P}$
        Add the output tensors of op into I
        BUILD($n+1$, $\mathcal{P}$, I)
        Remove operator op from $\mathcal{P}$
        Remove the output tensors of op from I
return $\mathcal{M}$

Algorithm 1 shows a *depth-first search* algorithm for constructing potential mutants of an MLTP $\mathcal{P}_0$. PET starts from an empty program with no operator and only the set of original input tensors to $\mathcal{P}_0$. PET iteratively adds a new operator to the current program $\mathcal{P}$ by enumerating the type of operator from O and the input tensors to the operator. The input tensors can be the initial input tensors to $\mathcal{P}_0$ (i.e., I_0 in Algorithm 1) or the output tensors of previous operators. The depth-first search algorithm enumerates all potential MLTPs up to a certain size (called the mutation *depth*). For each mutant $\mathcal{P}$, PET checks whether $\mathcal{P}$ and $\mathcal{P}_0$ have the same number and shapes of inputs/outputs. $\mathcal{P}$ is a valid mutant if it passes this test.

4.2 Example Mutant Categories

While the above mutation generation algorithm is general enough to explore a sufficiently large design space, we emphasize that several mutant categories are of particular importance to PET and lead to mutants with improved performance. Note that PET does not rely on manually specified categories. Rather, these categories are discovered by PET automatically.

Reshape and transpose. It is widely known that the in-memory *layouts* of tensors play an important role in optimizing tensor programs [6]. PET leverages the `reshape` and `transpose` operators to transform the shapes of input tensors and the linearization ordering of tensor dimensions to generate mutants with better performance. A `reshape` operator changes the shape of a tensor by decoupling a single dimen-

sion into multiple ones or combining multiple dimensions into one. E.g., a `reshape` can transform a vector with four elements into a 2×2 matrix. A `transpose` operator modifies the linearization ordering of a tensor's dimensions, such as converting a row-major matrix to a column-major one.

`Reshape` and `transpose` are generally applied jointly to transform the tensor layouts. For example, Figure 1 shows a potential mutant of a convolution operator that concatenates two separate images (i.e., $T_1 \rightarrow T_3$ in Figure 1(b)) along the width dimension to improve the performance of convolution: typically a larger width exhibits more parallelism to be exploited on modern accelerators such as GPUs. This concatenation involves a combination of three `reshape` and `transpose` operators. First, a `reshape` operator splits the batch dimension of T_0 into an inner dimension that groups every two consecutive images, and an outer dimension that is half the size of the original. Then, a `transpose` operator moves the newly created inner dimension next to the width dimension and updates the tensor's linearization ordering accordingly, so each row of the two images in the same group is stored consecutively in memory. Finally, another `reshape` operator combines the two images.

The mutation generator usually *fuses* multiple consecutive `reshape` and `transpose` operators into a single compound operator, namely `reshape & transpose`. This fusion reduces the size of the generated mutants and allows for exploring much larger and more sophisticated mutants.

Single-operator mutants. PET can also generate mutants that replace an inefficient operator in a tensor program with a different and more performant operator. Several standard tensor operators, such as convolution and matrix multiplication, have been extensively optimized either manually or automatically on modern hardware backends. In contrast, their variants, such as strided or dilated convolutions [20], are not as efficiently supported. There are performance-related benefits to mutating them into their standard counterparts with highly optimized kernels. As an example, Figure 3 shows a mutant that transforms a dilated convolution into a regular convolution by reorganizing the linearization ordering of the input tensor based on the given dilation. However, the mutant is not fully equivalent to the input program and requires corrections afterward to restore functional equivalence.

Multi-operator mutants. PET also supports substituting a subgraph of multiple operators with another more efficient set of operators. For example, a few independent convolutions with similar tensor shapes may be combined into a single larger convolution to improve GPU utilization and reduce kernel launch overhead. This requires manipulating tensor shapes and adding proper padding (see the examples in §8.3.3).

5 Mutation Corrector

While the mutants generated by PET have potentially higher performance than the original programs, they may produce different mathematical results on some regions of the output tensors, potentially leading to accuracy loss. To maintain transparency at the application level, PET chooses to preserve the statistical behavior of the input program and guarantees the same model accuracy, with the help of a *mutation corrector*. Specifically, the mutation corrector takes as inputs an MLTP $\mathcal{P}_0$ and one of its mutants $\mathcal{P}$, and automatically generates *correction kernels* that are applied to the output tensors of $\mathcal{P}$ to maintain functional equivalence to $\mathcal{P}_0$.

The goal of the mutation corrector is twofold. First, for any given MLTP and its mutant, the corrector analyzes the two programs and identifies all the regions of the output tensors on which the two programs provide identical results and therefore do not need any correction. Second, for the remaining regions where the two outputs are different, the corrector automatically generates kernels to fix the output of the mutant and preserve functional equivalence.

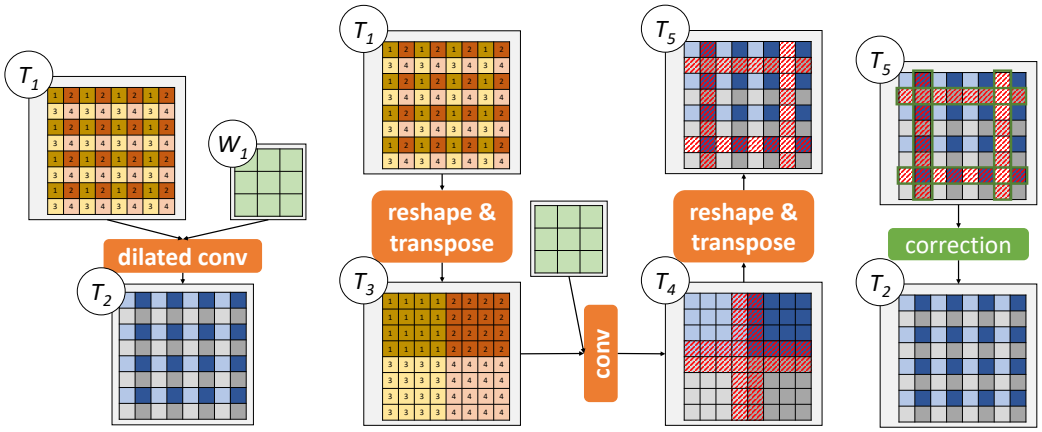
Designing the mutation corrector requires addressing two challenges. First, the output tensors may be very large, involving up to many millions of elements that all require equivalence verification. It is infeasible to verify *every* single element of the output tensors individually. Second, the verification of each output element may depend on a large number of input variables in many tensor operators. For example, each output element of a matrix multiplication is the inner product of one row and one column of the two input matrices, both with sizes up to several thousand. Numerically enumerating all possible values for this many input variables is impractical.

Two theorems that significantly simplify the verification tasks are central to the PET mutation corrector. Rather than verifying all output positions with respect to all input value combinations, PET only needs to verify a few representative output positions with a small number of randomly generated input values. This dramatically reduces the verification workload. We describe these theoretical foundations in §5.1 and introduce our mutation correction algorithm in §5.2.

5.1 Theoretical Foundations

To simplify our analysis, we assume an input MLTP $\mathcal{P}_0$ and its mutant $\mathcal{P}$ each has one output. Our results can be generalized to programs with multiple outputs by sequentially analyzing each one. Let $\mathcal{P}(I)$ denote the output tensor of running $\mathcal{P}$ on n input tensors $I = (I_1, ..., I_n)$. Let $\mathcal{P}(I)[\vec{v}]$ denote the output value at position $\vec{v}$, and let $I_j[\vec{u}]$ denote the input value at position $\vec{u}$ of I_j. With these definitions, the computation for a single output position of an MLTP $\mathcal{P}$ is represented as

$$\mathcal{P}(I_1, ..., I_n)[\vec{v}] = \sum_{\vec{r} \in \mathcal{R}(\vec{v})} \prod_{j=1}^{n} I_j[\mathbf{L}_j(\vec{v}, \vec{r})]$$



(a) Input program. (b) A partially equivalent transformation. (c) Correcting results.

Figure 3: An example mutant that transforms a dilated convolution to a standard convolution. The red-shaded boxes in (b) highlight non-equivalent elements between the two programs, which are fixed by the correction kernel in (c).

where $\mathcal{R}(\vec{v})$ is the *summation interval* of $\vec{v}$, which is iterated over when computing $\mathcal{P}(I)[\vec{v}]$, and $\vec{u} = \mathbf{L}_j(\vec{v},\vec{r})$ is a linear mapping from $(\vec{v},\vec{r})$ to a position $\vec{u}$ of the j-th input tensor I_j. For example, a convolution with a kernel size of 3×3 and zero padding is defined as

$$conv(I_1,I_2)[c,h,w] = \sum_{d=0}^{D-1}\sum_{x=\max(-1,-h)}^{\min(H-1-h,1)}\sum_{y=\max(-1,-w)}^{\min(W-1-w,1)} I_1[d,h+x,w+y]\times I_2[d,c,x,y] \quad (1)$$

where D, H, and W refer to the number of channels, height, and width of the input image I_1, respectively. The numbers below and above the summation symbols respectively denote the lower and upper bounds of the summation interval. The two linear mappings can be represented as $\mathbf{L}_1(\vec{v},\vec{r}) = (d,h+x,w+y)$ and $\mathbf{L}_2(\vec{v},\vec{r}) = (d,c,x,y)$, where $\vec{v} = (c,h,w)$ and $\vec{r} = (d,x,y)$.

Different positions of an output tensor may have different summation intervals. For the convolution operator defined above, computing the top-left output position (i.e., $h = 0$, $w = 0$) only involves a 2×2 kernel (i.e., $0 \le x \le 1, 0 \le y \le 1$) since that position does not have a left or top neighbor, as shown in Figure 4. We group the output positions with an identical summation internal into a *box*. Formally, a box is a region of an output tensor whose elements all have the same summation internal. This convolution has nine boxes overall, which are depicted in Figure 4.

All output positions in the same box have an identical summation internal and share similar mathematical properties, which are leveraged by PET when examining program equivalence. Instead of testing the equivalence of two MLTPs on all individual positions, PET only needs to verify their equivalence on $m+1$ specific positions in each box, where m is the number of dimensions of the output tensor.

Theorem 1 *For two MLTPs $\mathcal{P}_1$ and $\mathcal{P}_2$ with an m-dimension output tensor, let $\vec{e}_1,\ldots,\vec{e}_m$ be a set of m-dimension base vectors. That is, $\vec{e}_i = (0,\ldots,0,1,0,\ldots,0)$ is an m-tuple with all coordinates equal to 0 except the i-th.*

Let $\mathcal{B}$ be a box for $\mathcal{P}_1$ and $\mathcal{P}_2$, and let $\vec{v}_0$ be an arbitrary position in $\mathcal{B}$. Define $\vec{v}_j = \vec{v}_0 + \vec{e}_j, 1 \le j \le m$. If $\forall I, 0 \le i \le m$, $\mathcal{P}_1(I)[\vec{v}_i] = \mathcal{P}_2(I)[\vec{v}_i]$, then $\forall I, \vec{v} \in \mathcal{B}$, $\mathcal{P}_1(I)[\vec{v}] = \mathcal{P}_2(I)[\vec{v}]$.

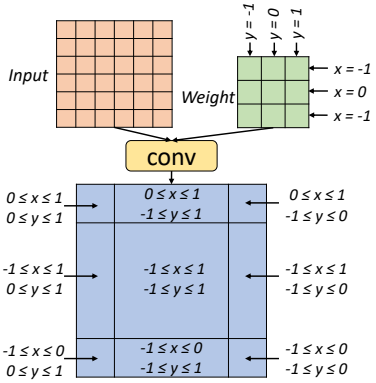


Figure 4: The nine boxes of a convolution with a 3×3 kernel and zero padding, as well as their summation intervals. A convolution has three summation dimensions (i.e., d, x, and y in Equation (1)). The channel dimension (i.e., d) has the same internal in all boxes and is thus omitted.

Proof sketch. The proof uses a lemma whereby if $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for positions $\vec{v}_0$ and $\vec{v}_0 + \vec{e}_i$, then the equivalence holds for $\vec{v}_0 + k\cdot\vec{e}_i$, where k is an integer. We prove this lemma by comparing the coefficient matrices of $\mathcal{P}_1$ and $\mathcal{P}_2$ with respect to the input variables. Using this lemma, we show that $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for the entire box $\mathcal{B}$, since any $\vec{v} \in \mathcal{B}$ can be decomposed to a linear combination of $\vec{v}_0$ and $\vec{e}_0,\ldots,\vec{e}_m$. □

Theorem 1 shows that, if $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent for $m+1$ specific positions in a box, identified by $\vec{v}_0,\ldots,\vec{v}_m$, then the equivalence holds for all other positions in the same box. This theorem significantly reduces the verification workload:

Table 2: Reducing verification workload in PET.

Methods	Output positions	Input combinations
Original	all	all
+ Theorem 1	a few positions	all
+ Theorem 2	a few positions	a few random inputs

instead of examining all positions of an output tensor, PET only needs to verify $m+1$ specific positions in each box.

The verification of a single position remains challenging, nevertheless, as each MLTP generally involves a large number of input variables. Proving the equivalence of two MLTPs requires examining all possible combinations of value assignments to these input variables. We further address this challenge using the following theorem.

Theorem 2 *For two MLTPs $\mathcal{P}_1$ and $\mathcal{P}_2$ with n input tensors, let $\vec{v}$ be a position where $\mathcal{P}_1$ and $\mathcal{P}_2$ are not equivalent, i.e., $\exists I,\ \mathcal{P}_1(I)[\vec{v}] \neq \mathcal{P}_2(I)[\vec{v}]$. Let I' be a randomly generated input uniformly sampled from a finite field $\mathbb{F}$. The probability that $\mathcal{P}_1(I')[\vec{v}] = \mathcal{P}_2(I')[\vec{v}]$ is at most $\frac{n}{p}$, where p is the number of possible values in $\mathbb{F}$.*

Proof sketch. This is a corollary of the Schwartz–Zippel Lemma [28, 35]. □

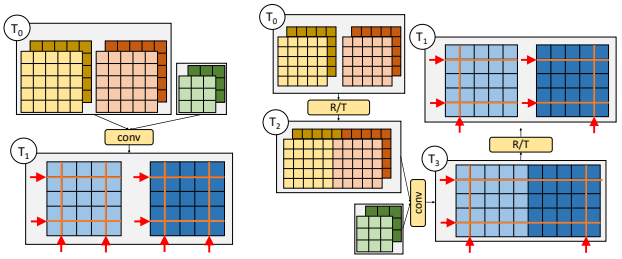
Theorem 2 shows that if two MLTPs with n inputs are not equivalent on a specific position $\vec{v}$, then the probability that they produce an identical result on this position with a random input sampled from a finite field $\mathbb{F}$ is low (i.e., at most $\frac{n}{p}$, where p is the number of possible values in $\mathbb{F}$). This theorem shows the sufficiency and effectiveness of random testing for examining the equivalence of two MLTPs.

Theorem 2 relies on the fact that $\mathbb{F}$ is a finite field, from which the random inputs are sampled, but MLTPs operate on the infinite field of real numbers. To apply Theorem 2, we choose $\mathbb{F}$ to be a field of integers modulo p, where p is a large prime number ($p = 2^{31}-1$ in our evaluation). The arithmetic operations in random testing are performed on integers and calculated modulo the prime number p. Working with a finite field provides another desirable property that applying arithmetic operators does not involve integer overflow.

By combining Theorems 1 and 2, PET reduces the original verification task of examining all output positions with respect to all input value combinations to a much more lightweight task that only requires testing a few representative positions using several randomly generated inputs, as shown in Table 2.

5.2 Mutation Correction Algorithm

The PET mutation correction algorithm exploits the theorems in §5.1 to calculate which regions of the output tensors in a mutant are not equivalent to the input MLTP and, therefore, need additional correction. In particular, it suffices to examine the equivalence for each pair of overlapped boxes from the two



(a) Input program $\mathcal{P}_0$. (b) A potential mutant $\mathcal{P}_1$.

Figure 5: Box propagation for the example in Figure 1. The red arrows indicate the split points of each tensor dimension.

MLTPs, using a small number of random tests. The overall algorithm works in the following three steps.

Step 1: Box propagation. First, we calculate the boxes of a given MLTP through *box propagation*. The idea of box propagation is similar to forward and backward propagation in deep learning: we compute the boxes of an operator's output tensors based on the boxes of its inputs, and the computation is conducted following the operator dependencies in a program. We maintain a set of *split points* for each dimension of a tensor to identify the boundaries of its boxes. For a multi-linear operator, we infer the split points of its output tensors based on the split points of its input tensors and the operator type and hyper-parameters. Figure 5 shows the box propagation procedure for the mutation example in Figure 1.

Step 2: Random testing for each box pair. After obtaining all boxes of an input MLTP $\mathcal{P}_1$ and its mutant $\mathcal{P}_2$, PET leverages the theorems in §5.1 to examine the intersected regions of each pair of boxes from $\mathcal{P}_1$ and $\mathcal{P}_2$. If two boxes do not have any overlapped region, they can be skipped. For each box intersection, PET examines the equivalence of the two programs on $m+1$ positions identified by Theorem 1, where m is the number of output tensor dimensions (e.g., $m=4$ in Figure 5, since the output of a convolution has four dimensions).

For each of these $m+1$ positions, PET runs a set of random tests by assigning input tensors with values uniformly sampled from a finite field $\mathbb{F}$ containing all integers between 0 and $p-1$, where $p = 2^{31}-1$ is a prime number. As a result, the probability that two non-equivalent MLTPs produce identical outputs on a random input is at most $\frac{n}{p}$, where n is the number of inputs to the MLTPs. Finally, two non-equivalent MLTPs pass all tests with a probability lower than $(\frac{n}{p})^t$, where t is the number of test cases and a hyper-parameter in PET that serves as a tradeoff between the speed of the corrector and the error probability that non-equivalent MLTPs pass all random tests.

Our approach introduces an extremely small and controllable probability of error that we have to tolerate. That is, non-equivalent programs may pass random testing with probability $(\frac{n}{p})^t$. We argue that this is an example of how random testing can enable a tradeoff between the cost of program verification and a small probability of unsoundness for verifying

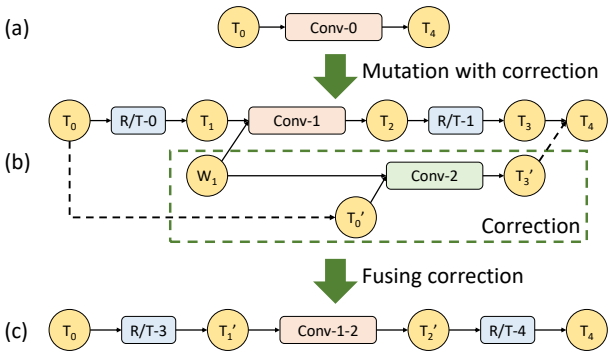


Figure 6: Fusing correction kernels with DNN kernels.

tensor program transformations.

To further reduce the verification workload, PET includes a *caching optimization*: the tests for all boxes share the same set of random inputs, and PET caches and reuses all intermediate results to avoid redundant computations.

Step 3: Correction kernel generation. For each box failing the random tests, PET generates *correction kernels* to fix its outputs and restore the mathematical equivalence between the original MLTP and its mutant. To fix the outputs, the correction kernel performs the same set of operations as the original MLTP but only on those boxes where the two input programs are not equivalent (shown as the red shaded boxes in Figure 1). These boxes are regular cubes in the multi-dimensional space and can be viewed as sub-tensors of the original ones but with much smaller sizes. Therefore, PET directly leverages existing DNN libraries [8, 10] or kernel generation techniques [6, 34] to generate correction kernels. To reduce the correction overhead, PET opportunistically fuses the correction kernels with existing tensor operators (§5.3).

5.3 Fusing Correction Kernels

Correction kernels may introduce non-trivial overheads due to the cost of launching the correction kernels and their limited degrees of parallelism. For example, some correction kernels may have similar execution time compared to the corresponding full-size tensor operators. This may eliminate the performance gains from applying partially equivalent transformations. To reduce the correction overhead, PET opportunistically fuses correction kernels with other tensor operators.

For example, Figure 6(b) shows the tensor program after applying the partially equivalent transformation in Figure 1. `Conv-2` is the correction kernel for fixing the output of `Conv-1`. Since the two convolution operators share the same weights (i.e., W_1), PET fuses them into a single convolution, shown as `Conv-1-2` in Figure 6(c). This fusion requires concatenating T_1 and T_0' into a single tensor and splitting the output of `Conv-1-2` into T_2 and T_3'. The concatenation and split only involve direct memory copies and can be fused with the `reshape` and `transpose` operators.

6 Program Optimizer

In this section, we describe the program optimizer in PET, which explores a large search space of program optimizations, combining fully and partially equivalent transformations, and quickly discovers highly optimized programs. The program optimizer first splits an input program into multiple subprograms with smaller sizes to allow efficient mutation generation (§6.1). Second, to optimize each individual subprogram, PET searches for the best mutants in a rich candidate space by varying both the subsets of operators to mutate together and the number of iterative rounds of mutation (§6.2). Finally, when stitching the optimized subprograms back together, PET applies additional post-optimizations across the boundaries of the subprograms, including redundancy elimination and operator fusion (§6.3). The overall program optimization algorithm is summarized in Algorithm 2.

Algorithm 2 Program optimization algorithm.

Input: An input tensor program $\mathcal{P}_0$
Output: An optimized tensor program $\mathcal{P}_{\text{opt}}$

Split $\mathcal{P}_0$ into a list of subprograms
Initialize a heap $\mathcal{H}$ to record the top-K programs
$\mathcal{H}$.`insert`($\mathcal{P}_0$)
// Greedily mutate each subprogram
for each subprogram $\mathcal{S} \in \mathcal{P}_0$ **do**
    mutants = GETMUTANTS($\mathcal{S}$)
    Initialize a new heap $\mathcal{H}_{\text{new}}$
    for $\mathcal{P} \in \mathcal{H}$ **do**
        for $\mathcal{M} \in$ *mutants* **do**
            $\mathcal{P}_{\text{new}}$ = replace $\mathcal{S}$ with $\mathcal{M}$ in $\mathcal{P}$
            Apply post-optimizations on $\mathcal{P}_{\text{new}}$
            $\mathcal{H}_{\text{new}}$.`insert`($\mathcal{P}_{\text{new}}$)
    $\mathcal{H} = \mathcal{H}_{\text{new}}$
$\mathcal{P}_{\text{opt}}$ = the program with the best performance in $\mathcal{H}$
return $\mathcal{P}_{\text{opt}}$

function GETMUTANTS($\mathcal{S}_0$)
    O = the set of mutant operators for $\mathcal{S}_0$
    $Q = \{\mathcal{S}_0\}$, *mutants* = $\{\mathcal{S}_0\}$
    for r rounds **do**
        $Q_{\text{new}} = \{\}$
        for $\mathcal{S} \in Q$ **do**
            for each subset of operators $\mathcal{S}' \in \mathcal{S}$ **do**
                for $\mathcal{M}' \in$ MUTATIONGENERATOR(O, $\mathcal{S}'$) **do**
                    $\mathcal{M}$ = replace $\mathcal{S}'$ with $\mathcal{M}'$ in $\mathcal{S}$
                    Add $\mathcal{M}$ to Q_{new} and *mutants*
        $Q = Q_{\text{new}}$
    return *mutants*

6.1 Program Splitting

The complexity of the mutation generation grows rapidly with the input program size, as explained in §4. It is nearly im-

possible to directly mutate a large tensor program with many hundreds of operators. Instead, PET splits an input program into multiple disjoint subprograms with smaller sizes.

It is crucial to properly select the split points for an input program, to effectively reduce the mutation complexity while still preserving most program optimization opportunities. More split points lead to smaller subprograms with fewer mutant candidates to be explored. As an extreme case, by constraining each subprogram to have only a small constant number of operators, the overall complexity scales linearly with the program size, rather than the naive exponential trend. However, an overly aggressive split may result in locally optimized mutants that are limited within subprograms, missing optimization opportunities across subprogram boundaries.

We use *non-linear operators* in tensor programs as the split points. First, non-linear operators such as the activation layers in DNNs are widely used in tensor programs. Typically, each one or a few linear operators are followed by a non-linear activation (e.g., ReLU or sigmoid). This effectively limits the split subprograms to the small sizes we expect. Second, as §5 explains, PET's mutation only applies to MLTPs; any non-linear operators must be excluded from the mutation. This makes splitting at the points of non-linear operators a natural choice for the partially equivalent mutation in PET. Third, our design is also motivated by an observation that most existing tensor program transformations [2, 6, 15] also do not include non-linear operators in their substitution patterns (except for fusion, which we handle in §6.3).

PET further adjusts the subprogram sizes after splitting an input program at the non-linear operators. For multiple individual subprograms without any data dependency, PET considers the possibility of combining them into a single subprogram using grouped or batched operators. Examples include fusing the standard convolutions on different branches of an Inception network [31] into a grouped convolution, as shown in Figure 10. On the other hand, if a subprogram is still too large, PET will only query the mutation generator with a subset of operators each time (see §6.2).

6.2 Subprogram Optimization

After splitting an input program into multiple individual subprograms, PET mutates each subprogram by querying the mutation generator in §4.1 and keeps the top-K candidates with the best estimated performance in a heap structure $\mathcal{H}$, as shown from Lines 7 to 16 in Algorithm 2. A larger K allows PET to tolerate intermediate performance decreases during the search but requires more memory to save all K candidates and involves higher computation cost. At each step, each of the obtained mutants replaces its corresponding subprogram in each of the current candidates (i.e., $\mathcal{P}$ in Algorithm 2) to generate a new candidate (i.e., $\mathcal{P}_{\text{new}}$), which is then applied a series of post-optimizations (see §6.3).

PET estimates the performance of each new candidate $\mathcal{P}_{\text{new}}$ using a cost model adapted from TASO [15]. The cost model measures the execution time of each tensor operator once for each configuration (e.g., different strides and padding of a convolution), and estimates the performance of a new program candidate $\mathcal{P}_{\text{new}}$ by summing up the measured execution time of its operators. The top-K program candidates with the best performance thus far are kept in $\mathcal{H}$.

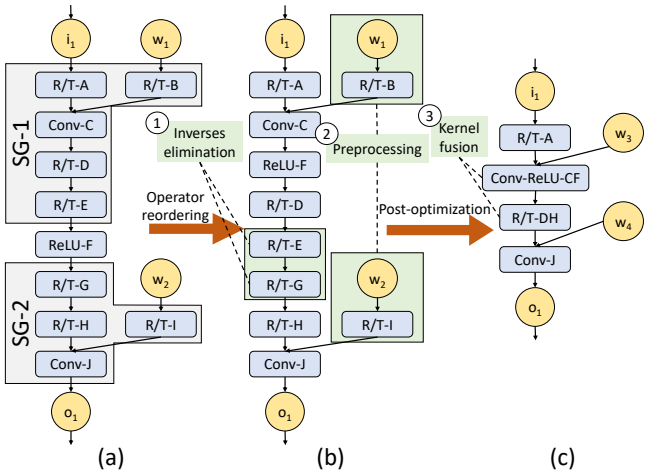


Figure 7: Post-optimizations applied when stitching two subprograms `SG-1` and `SG-2`. `R/T` refers to a `reshape` followed by a `transpose`. `Conv` and `ReLU` denote a convolution and a ReLU operator, respectively.

To explore a sufficiently large space of possible mutants for each subprogram within reasonable time and space cost, we manage the mutation process with several key features. First, when the number of operators in a subprogram exceeds a threshold d (our evaluation uses $d = 4$), PET breaks the subprogram into smaller subsets of operators by enumerating all possible combinations with up to d operators, and only queries the mutation generator on the subset, while keeping the remaining operators unchanged (Algorithm 2 Line 26). Second, we allow iterative mutation on a subprogram for up to r rounds (Algorithm 2 Line 23), which significantly enlarges the search space of possible mutants and allows PET to discover more optimized mutants. All generated mutants in all rounds are returned to the optimizer as potential candidates.

It is worth noting that PET's optimizer is compatible with and can incorporate existing fully equivalent transformations [2, 15] besides PET's mutations. Doing so merely requires enhancing the mutation generator to explore and return fully equivalent transformations as well, which are directly applicable to the input subprograms in the same way as the mutations. By combining fully and partially equivalent transformations, PET explores a significantly larger search space of program optimizations and discovers highly optimized programs that existing optimizers miss.

6.3 Post-Optimizations

Finally, the optimized mutants for all subprograms need to be stitched together. In addition to connecting their input and output tensors, we also perform several post-optimizations across the subprogram boundaries to further improve the overall performance. We observe that the mutation generator in PET introduces a large number of `reshape` and `transpose` (`R/T`) operators, especially at the beginning and the end of each subprogram. There are opportunities to fuse these `R/T` operators across subprograms and further fuse the non-linear operators that are excluded from the above subprogram optimizations.

Figure 7 shows an example with two optimized subprograms. To optimize the boundaries between subprograms, PET first groups together all `R/T` operators between subprograms by reordering the `R/T` operators with element-wise non-linear activations (e.g., ReLU and sigmoid), as shown in Figure 7(b). This reordering is functionally correct, since both `reshape` and `transpose` are commutative with element-wise operators. The reordering also allows PET to fuse the non-linear activations with other linear operators, such as fusing a `Conv` and a subsequent `ReLU` into a `Conv-ReLU`, as shown in Figure 7(c). We then apply the following three post-optimizations.

Inverses elimination. We eliminate any pairs of `R/T` operators that can cancel out each other and therefore are equivalent to a no-op. We call each such pair an inverse group and directly remove them as part of the post-optimization. An example of an inverse group is `R/T-E` and `R/T-G` in Figure 7(b).

Operator fusion. As shown in Figure 7(c), PET fuses the remaining consecutive `R/T` operators into a single operator (e.g., `R/T-DH`) to reduce the kernel launch cost. The non-linear activations in a tensor program are also fused with an `R/T` or with other linear operators. Note that operator fusion is the most commonly used, if not the only, program optimization for non-linear operators. PET is able to recover most of the efficiency that was lost when splitting the tensor program.

Preprocessing. We preprocess any operator if all its input tensors are statically known. For example, in Figure 7(b), both `R/T-B` and `R/T-I` can be preprocessed on the convolution weight tensors w_1 and w_2.

7 Implementation

PET is implemented as an end-to-end tensor program optimization framework, with about 13,000 lines of C++ code and 1,000 lines of Python code. This section describes our implementation of the PET mutation generator and corrector.

Mutation operators. Table 1 lists the tensor operators included in the current implementation of PET. We use cuDNN [8] and cuBLAS [10] as our backend operator libraries. PET can also be extended to include other libraries, such as TVM [6] and Ansor [34]. In our evaluation, we demonstrate this extensibility on TVM and Ansor, and show that they can directly benefit from PET's partially equivalent optimizations and automated corrections.

`Reshape` and `transpose` are two frequently used operators in partially equivalent transformations. Our implementation includes a series of optimizations on them, including eliminating inverse groups of `R/T` operators and fusing consecutive `R/T` operators, as described in §6.3. Since both `reshape` and `transpose` are multi-linear operators, PET directly uses the random testing method introduced in §5 to examine whether a sequence of `R/T` operators forms an inverse group and therefore can be eliminated.

Correction kernels. §5.2 describes a generic approach to generate correction kernels by directly running the original program on the positions with incorrect results. To reduce the correction overhead, PET fuses the correction kernels with other tensor operators, as described in §5.3. The correction kernel fusion introduces additional memory copies, which are also fused with the `R/T` operators during post-optimizations.

8 Evaluation

8.1 Experimental Setup

Platforms. We use a server equipped with two-socket, 28-core Intel Xeon E5-2680 v4 processors (hyper-thread enabled), 256 GB of DRAM, and one NVIDIA Tesla V100 GPU. All experiments use CUDA 10.2 and cuDNN 7.6.5 except for those with TVM and Ansor, which directly use the best kernels generated by these backends.

PET preserves an end-to-end equivalence between the original and optimized programs, same as all the baselines. PET takes ONNX models as input. TensorRT and TASO directly support the ONNX format. For TensorFlow and TensorFlow-XLA, we use the onnx-tensorflow tool [25] for format conversion.

Workloads. We use five DNN architectures. Resnet-18 [14] is a widely used convolutional network for image classification. CSRNet [20] is a dilated convolutional network used for semantic segmentation. Its sampling rate can be arbitrarily adjusted to enlarge the receptive field for higher accuracy. Inception-v3 [31] is an improved version of GoogLeNet [30] with carefully designed Inception modules to improve accuracy and computational efficiency. BERT [12] is a language representation architecture that obtains state-of-the-art accuracy on a wide range of natural language tasks. Resnet3D-18 [13] is a 3D convolutional network for video processing.

Unless otherwise stated, in all experiments, we use CUDA events to measure the elapsed time from launching the first CUDA kernel in a tensor program to receiving the completion notification of the last kernel. We set the default mutation generation depth to 4 (i.e., $depth = 4$ in Algorithm 1) and the search rounds to 4 (i.e., $r = 4$ in Algorithm 2). We further evaluate the scalability of the mutation generator and the program optimizer in §8.5.

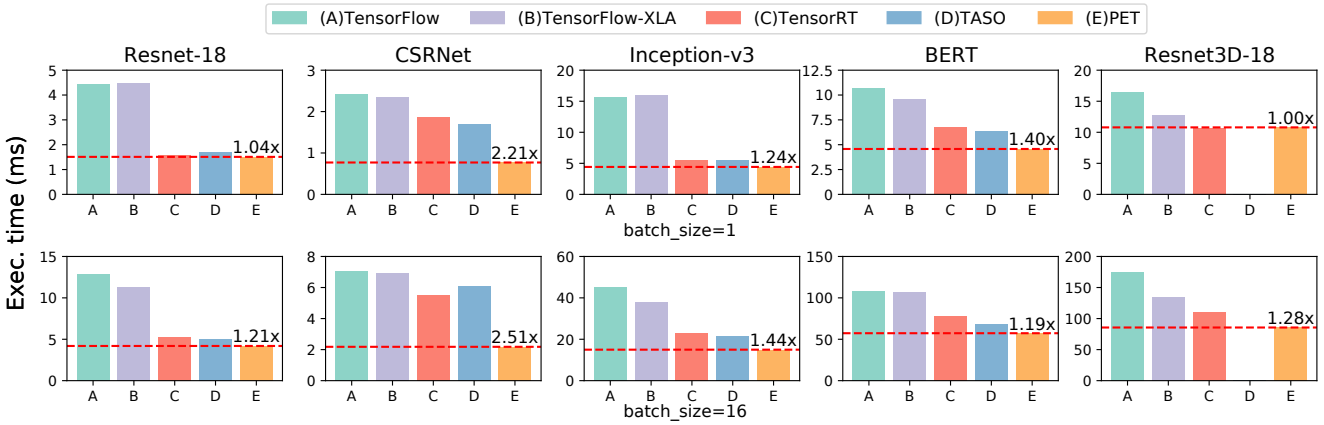


Figure 8: End-to-end performance comparison between PET and existing frameworks. For each DNN, the numbers above the PET bars show the speedups over the best baseline. TASO does not support the 3D convolution operators in Resnet3D-18.

8.2 End-to-End Evaluation

We first compare the end-to-end inference performance between PET and existing tensor program optimizers, including TensorFlow [3], TensorFlow XLA [1], TensorRT [32], and TASO [15]. Figure 8 shows the results under batch sizes of 1 and 16. To eliminate the impact of using different operator libraries, all optimizers use the same cuDNN [8] and cuBLAS [10] libraries as the backend. Therefore, the performance differences only come from different optimized tensor programs produced by PET and the baselines. §8.4 further evaluates PET with existing kernel generation techniques, such as TVM [6] and Ansor [34].

Among the five DNN architectures, Resnet-18 and Resnet3D-18 are commonly used and heavily optimized in existing DNN frameworks. However, PET is still able to improve their performance by up to 1.21× and 1.28×, respectively, by discovering new partially equivalent transformations not considered by existing optimizers. For Resnet-18, CSRNet, and Inception-v3, PET achieves higher speedups with a batch size of 16. This is because a larger batch size offers more mutation opportunities across different tensor dimensions for PET to exploit. Overall, PET outperforms existing DNN frameworks by up to 2.5×.

To further evaluate the partially equivalent transformations discovered by PET, we manually add them and corresponding correction kernels as additional graph substitutions into TASO, and measure by how much these new transformations improve TASO's performance. As shown in Figure 9, the enhanced version of TASO further improves the inference performance of Inception-v3 and BERT by 1.12× and 1.31×, respectively. This demonstrates that partially equivalent transformations indeed enlarge the design space of graph transformations, and PET unleashes these benefits automatically. Some non-trivial partially equivalent transformations are not leveraged by TASO, due to substantial correction overhead, while PET is able to avoid this overhead through correction kernel fusion (§5.3) and post-optimization (§6.3).

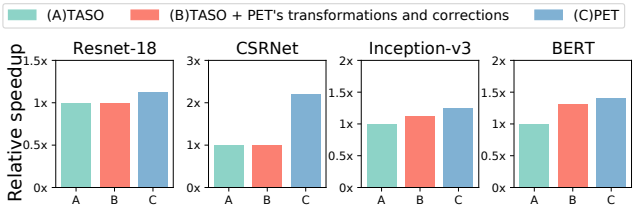


Figure 9: Performance benefits after adding PET's partially equivalent transformations into TASO.

Table 3: Operator benchmark list.

Operator	Input	Weight	#Op
conv	[1, 48, 38, 38]	[64, 48, 5, 5]	1
dilatedconv	[1, 512, 14, 14]	[256, 512, 3, 3]	1
groupconv	[1, 768, 18, 18] [1, 768, 18, 18]	[192,768, 1, 1] [160,768, 1, 1]	2 2
batchmatmul	[512, 768]	[768, 768]	3

8.3 Case Studies

To understand how partially equivalent transformations discovered by PET optimize DNN computation, we study four optimization categories in detail.

8.3.1 Tensor-Level Optimization

PET discovers many partially equivalent transformations that improve DNN computation by optimizing the shapes or linearization of tensors. We evaluate a convolution operator in Inception-v3, whose configuration is depicted in Table 3 conv. PET transforms the input tensor shape from [1, 48, 38, 38] to [16, 48, 10, 10] by splitting both the height and width dimensions each into four partitions. IGEMM and FFT are the most efficient convolution algorithms before and after the optimization, respectively. Using the transformed input tensor

Table 4: Case studies on the performance of the `conv` and `dilatedconv` operators in Table 3. `IGEMM`, `FFT`, and `WINO` refer to implicit GEMM, Fast Fourier Transform, and Winograd convolution algorithms, respectively. For `conv`, the optimized program transforms the input tensor shape from [1, 48, 38, 38] to [16, 48, 10, 10]. For `dilatedconv`, the optimized program replaces the `dilatedconv` with a regular convolution with the same input and kernel sizes.

		Algo	Time (us)	# GPU DRAM	# GPU L2	FLOP
conv	Original	IGEMM	90	1.51×10^4	2.80×10^6	2.26×10^8
		FFT	352	1.06×10^8	1.15×10^8	8.75×10^7
	Optimized	IGEMM	90	1.52×10^4	1.46×10^6	2.46×10^8
		FFT	51	1.09×10^6	7.44×10^6	1.26×10^8
dilated conv	Original	IGEMM	153	1.06×10^5	2.46×10^6	1.32×10^8
		WINO	N/A	N/A	N/A	N/A
	Optimized	IGEMM	153	8.54×10^4	1.80×10^6	1.32×10^8
		WINO	79	2.23×10^6	6.36×10^6	7.20×10^7

reduces the GPU DRAM and L2 accesses by 100× and 15×, respectively, and thus reduces the run time by 7× (Table 4).

As another example of tensor-level optimization, for `conv` with a stride size larger than 1 (i.e., the output tensor is a down-sample of the input tensor), PET can reorganize the linearization of the tensors and reduce the stride size to 1, which improves the computation locality.

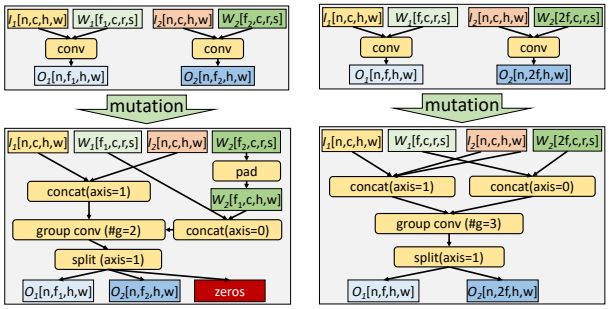
8.3.2 Operator-Level Optimization

For operators with less efficient implementations on specific hardware backends, PET can opportunistically replace them with semantically similar ones with more optimized implementations. We study the performance of a dilated convolution in CSRNet [20], whose configuration is shown in Table 3 `dilatedconv`. PET replaces it with a regular convolution operator (as shown in Figure 3) to enable more efficient algorithms on GPUs such as Winograd [17]. This reduces the execution time by 1.94× (Table 4).

Other examples of operator-level optimizations include replacing a batch matrix multiplication with a standard matrix multiplication, a group convolution with a convolution, and an average pooling with a group convolution or a convolution if the replacement leads to improved performance, even when including the correction cost.

8.3.3 Graph-Level Optimization

PET also discovers graph-level optimizations. Figure 10 shows two graph transformations discovered by PET to optimize Inception-v3 [31]. For two parallel `conv` operators with different numbers of output channels, Figure 10(a) shows a non-equivalent transformation that fuses the two `conv` operators into a `groupconv` by padding W_2 with zeros, so that the output of `pad` has the same shape as W_1. The correction splits and discards the *zeros* tensor at the end (shown in red).



(a) (b)

Figure 10: Mutants discovered by PET for Inception-v3. `axis` denotes the dimension on which to perform `concat` and `split`.

PET also discovers fully equivalent transformations that are missed by existing frameworks. The mutation corrector can successfully verify the equivalence for all output elements, in which case no correction is needed. Figure 10(b) shows a new equivalent transformation discovered by PET that optimizes two `conv` operators by duplicating the input tensors (i.e., I_1 and I_2) and fusing the two `conv` operators into a `groupconv`. Note that Figure 10 shows two different mutants of the same input program. PET's program optimizer can automatically select a more efficient one based on the performance of these mutants on specific devices.

8.3.4 Kernel Fusion

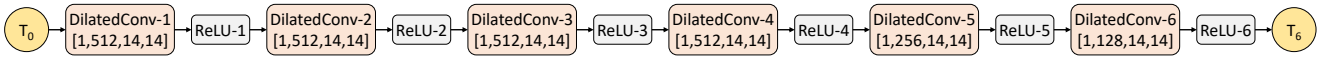
We use CSRNet [20] as an example to study the effectiveness of PET's kernel fusion optimization. Figure 11(a) and Figure 11(b) show the original and optimized model architectures of CSRNet. The numbers in each operator denote the input tensor shape. To demonstrate the correction kernel fusion and post-optimization in PET, Figure 11(c) shows the subprogram of a single dilated convolution before post-optimization, which contains three correction kernels and six `R/T` (i.e., `reshape` and `transpose`) operators. These correction kernels are fused with `Conv-4`, as described in §5.3. In addition, the multiple `R/T` operators between convolutions are fused into a single one during post-optimization (§6.3).

Fusing correction kernels and `R/T` operators is critical to PET's performance. In an ablation study, disabling kernel fusion in PET decreases the performance of the final program by 2.9×, making it even slower than the original one.

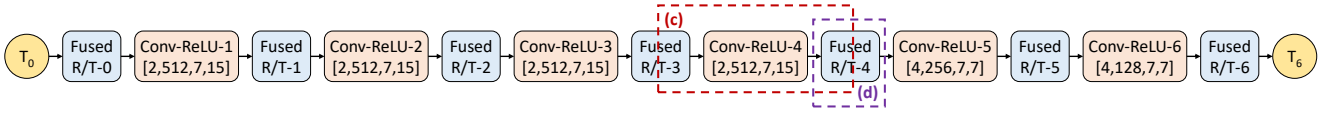
8.4 TVM and Ansor

PET improves tensor computations by generating and correcting partially equivalent transformations and is therefore orthogonal to and can potentially be combined with recent kernel generation techniques, such as TVM [6] and Ansor [34].

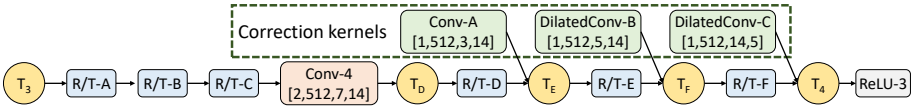
We evaluate PET on TVM and Ansor with a set of commonly used DNN operators, including `conv`, `dilatedconv`,



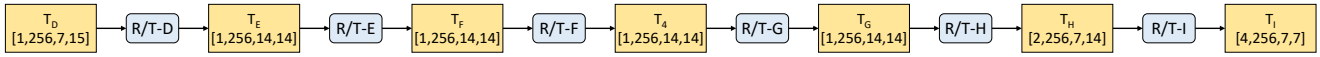
(a) CSRNet before optimization.



(b) CSRNet after optimization.



(c) `DilatedConv-4`'s subprogram after subprogram optimization but before post-optimization.



(d) Unfused R/T operators and corresponding tensors' shape of F R/T-4.

Figure 11: Optimization details in PET for CSRNet.

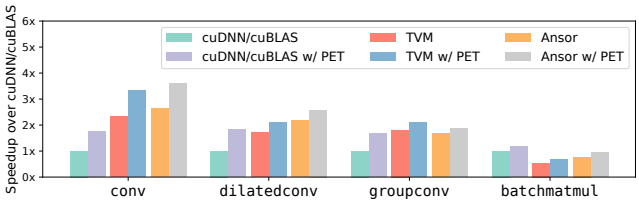


Figure 12: Performance comparison of PET on the cuDNN/cuBLAS, TVM, and Ansor backends. The performance is normalized to cuDNN/cuBLAS without PET.

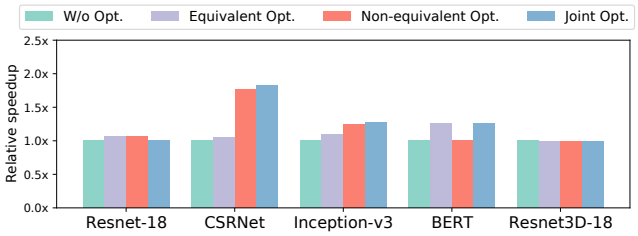


Figure 13: Performance comparison of tensor program optimizations using only (fully) equivalent transformations, only partially equivalent transformations, and both (as in PET).

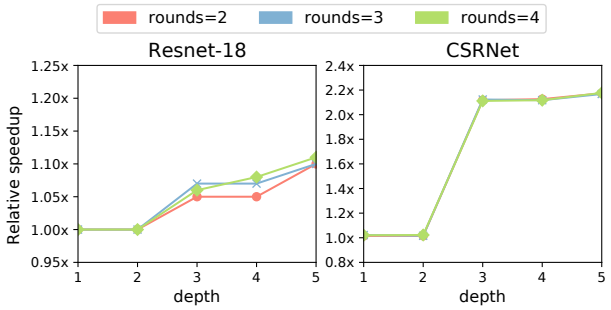


Figure 14: Performance comparison by using PET with different mutation depths (§4.1) and rounds (§6.2).

`groupconv`, and `batchmatmul`, which are obtained from Resnet-18, CSRNet, Inception-v3, and BERT, respectively. Their shape configurations are listed in Table 3. To generate kernels for potential mutants during the search, we allow TVM and Ansor to run 1024 trials and use the best discovered kernels to measure the cost of the mutants.

As Figure 12 shows, when combining PET with TVM and Ansor, PET can improve the performance of the evaluated operators by up to 1.23× and 1.21×, respectively, compared to directly generating kernels for these operators. Beyond such simple combinations, joint optimization of PET and existing kernel generation techniques would uncover more benefits, which we leave as future work.

8.5 Ablation and Sensitivity Studies

The key insight of PET is to explore partially equivalent program mutants, while state-of-the-art frameworks only capture fully equivalent transformations [15, 34]. We run several variants of PET to evaluate the benefits of considering either fully or partially equivalent program transformations, or both of them, as PET does. Figure 13 shows the results. When restricting PET to consider only equivalent transformations, it achieves similar performance gains as previous work such as TASO. Partially equivalent transformations, by themselves, enable noticeable benefits but also miss significant potential. Finally, PET achieves the highest performance by jointly considering both fully and partially equivalent transformations.

Finally, PET relies on several heuristic parameters to balance the search time and the resultant program performance. The mutation depth in Algorithm 1 limits the maximum number of operators in a program mutant; the mutation round in Algorithm 2 specifies the maximum number of iterations to apply mutations. Larger values of these thresholds allow larger design spaces of potential mutants but also require

more time to search. Figure 14 compares the performance of the optimized programs under different searching depths and rounds for Resnet-18 and CSRNet. The performance gains keep increasing with larger rounds values for Resnet-18, due to the generation of more optimized mutants, while for CSRNet, the performance improvement mainly comes from larger mutation depth. On the other hand, increasing the mutation depth from two to three improves the performance for both models significantly, since many mutations PET finds are subprograms with three operators. In summary, the key takeaway is that PET has only moderately high search complexity yet achieves significant performance gains.

8.6 Searching Time

PET uses a program optimizer to explore the search space of possible mutants and discover highly optimized candidates. Typically, it takes under 3 minutes (89 seconds, 88 seconds, 91 seconds, and 165 seconds on Resnet-18, CSRNet, BERT, and Resnet3D-18, respectively) for PET to find highly optimized program mutants with a batch size of 1. However, PET spends about 25 minutes optimizing Inception-v3, due to the multiple branches in the Inception modules [31]. Although their search spaces are not directly comparable, PET's search time is on par with state-of-the-art DNN optimization frameworks such as TASO [15] and Ansor [34], and is acceptable because it is a one-time cost before stable deployment. We leave any further search optimizations, such as aggressive pruning and parallelization, to future work.

9 Related work

Graph-level optimizations. TensorFlow [3], TensorRT [32], TVM [6], and MetaFlow [16] optimize tensor programs by applying substitutions that are manually designed by domain experts. TASO [15] generates graph substitutions automatically from basic operator properties, which significantly enlarges search space and reduces human effects. The key difference between PET and these frameworks is that PET can generate and correct partially equivalent transformations, enabling a significantly larger space of program optimizations.

Program mutation is a program testing technique designed to evaluate the quality of existing test cases [11]. By randomly mutating the input program and running the generated mutants on existing test cases, the technique can quickly estimate the coverage of these test cases. PET generates mutants for a different purpose. Instead of testing an input tensor program, the mutants generated by PET are used for performance optimizations on the program.

Code generation. Halide [27] is a programming language designed for high-performance tensor computing, and several works are proposed based on its scheduling model [4, 19, 22]. TVM [6, 7] uses a similar scheduling language and a learning-based approach to generate highly optimized code for different hardware backends. Ansor [34] explores larger search spaces than TVM and finds better optimized kernels. TensorComprehensions [33] and Tiramisu [5] use polyhedral compilation models to solve code generation problems in deep learning. As shown in §8.4, PET's program-level optimizations are orthogonal and can be combined with these code generation techniques.

Data layout optimization. NeoCPU [21] optimizes CNN models by changing the data layout and eliminating unnecessary layout transformations on CPUs, while Li et al. [18] explore the memory efficiency for CNNs on GPUs. Chou et al. [9] introduce a language to describe the different sparse tensor formats and automatically generate code for converting data layouts. Many transformations discovered by PET also involve layout conversions. However, the key differences between PET and prior work are that PET considers more complicated layouts and combines tensor layout optimizations with operator- and graph-level optimizations.

AutoML. Recent work has proposed approaches to search for accurate neural architectures by iteratively proposing modifications to the models' architectures and accepting proposals with the highest accuracy gain. Examples include automatic statistician [29] and TPOT [24]. These approaches apply non-equivalent transformations to a model architecture and rely on expensive retraining steps to evaluate how each transformation affects model accuracy. On the contrary, PET leverages performance optimizations in non-equivalent transformations and applies automated corrections to preserve an end-to-end equivalence. As such, PET does not require retraining.

10 Conclusion

We present PET, the first DNN framework that optimizes tensor programs with partially equivalent transformations and automated corrections. PET discovers program transformations that improve DNN computations with only partial functional equivalence. Automated corrections are subsequently applied to restore full equivalence with the help of rigorous theoretical guarantees. The results of our evaluation show that PET outperforms existing frameworks by up to 2.5× by unlocking partially equivalent transformations that existing frameworks miss. PET is publicly available at https://github.com/thu-pacman/PET.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Behnaz Arzani, for their valuable comments and suggestions. This work is partially supported by National Natural Science Foundation of China (U20A20226, 62072262) and Beijing Natural Science Foundation (4202031). Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017.

[2] TensorFlow Graph Transform Tool. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/graph_transforms, 2018.

[3] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[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] 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.

[6] 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. *CoRR*, abs/1802.04799, 2018.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018.

[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. *arXiv preprint arXiv:2001.02609*, 2020.

[10] Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

[11] Richard A DeMillo, Edward W Krauser, and Aditya P Mathur. Compiler-integrated program mutation. In *1991 The Fifteenth Annual International Computer Software & Applications Conference*, pages 351–352. IEEE Computer Society, 1991.

[12] 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.

[13] 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.

[14] 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*, CVPR, 2016.

[15] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[16] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[17] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[18] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016.

[19] 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.

[20] Yuhong Li, Xiaofan Zhang, and Deming Chen. Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1091–1100, 2018.

[21] 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.

[22] 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.

[23] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[24] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[25] TensorFlow Backend for ONNX. https://github.com/onnx/onnx-tensorflow.

[26] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

[27] 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, 2013.

[28] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[29] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. The automatic statistician. In *Automated Machine Learning*, pages 161–173. Springer, Cham, 2019.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[31] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[32] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

[33] 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. *CoRR*, abs/1802.04730, 2018.

[34] 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.

[35] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.

A Artifact Appendix

A.1 Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the OSDI' 21 paper: PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections.

A.2 Pet Usage

PET provides C++ API to build the input tensor program, and also supports importing input tensor program from ONNX[1] model. For each input tensor program, PET generates a mathematically equivalent executable that includes the performance optimizations described in this paper. PET uses cuDNN/cuBLAS as backend by default, but users can also export the mutation subprograms with their corresponding input/output tensor shapes to use different backends like TVM and Ansor.

A.3 Scope

The artifact can be used for evaluating and reproducing the main results of the paper, including the end-to-end evaluation, the operator-level evaluation, the performance comparison across different optimization policies and heuristics parameters, and the searching time.

A.4 Contents

The artifact evaluation includes the following experiments:
E1: An end-to-end performance comparison between PET and other frameworks. (Figure 8)
E2: An operator-level performance comparison on different backends, including cuDNN/cuBLAS, TVM, and Ansor. (Figure 12)
E3: A performance comparison across different optimization policies, including fully-equivalent transformations, partially-equivalent transformations, and joint optimization using both. (Figure 13)
E4: A performance comparison using different heuristics. (Figure 14)
E5: Searching time. (Section 8.6)

[1] `https://onnx.ai/`

A.5 Hosting

The source code of this artifact can be found on GitHub: `https://github.com/whjthu/pet-osdi21-ae`, master branch, with commit ID: 9e07cb1.

A.6 Requirements

Hardware dependencies

This artifact depends on an NVIDIA V100 GPU.

Software dependencies

This artifact depends on the following software libraries:

- PET uses cuDNN and cuBLAS libraries as backend. Our evaluation uses CUDA 10.2 and cuDNN 7.6.5.
- TensorFlow, TensorRT, TASO, TVM and Ansor are used as baseline DNN frameworks in E1 and E2. Our evaluation on these baseline uses TensorFlow 1.15, TensorRT 7.0.0.11, TASO with commit ID f11782c (we add some minor fixes for TASO to support the tested models), and TVM with commit ID 3950639.

A.7 Installation

A.7.1 Install Pet from source

- Clone code from git
- Install PET
  - `mkdir build; cd build; cmake ..`
  - `make -j`
- Set the environment for evaluations
  - `export PET_HOME=path_to_pet_home`

A.7.2 Install other frameworks

Please refer to the artifact evaluation instruction (README.pdf in the git repo `https://github.com/whjthu/pet-osdi21-ae`) or the installation instructions provided by the frameworks.

A.8 Experiments workflow

The following experiments are included in this artifact. All DNN benchmarks use synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU. The detailed running instruction can be found in the artifact evaluation instruction (README.pdf in the git repo `https://github.com/whjthu/pet-osdi21-ae`).

A.8.1 End-to-end performance (E1)

This experiment reproduces Figure 8 in the paper. Prerequisite: generate ONNX models

- `cd $PET_HOME/models-ae`
- `./generate_onnx.sh`

TensorFlow & TensorFlow XLA. The TensorFlow & TensorFlow XLA results of the 4 models are available in the `tensorflow_ae` folder. The following command lines measure the inference latency of TensorFlow and TensorFlow XLA, respectively:

- `cd $PET_HOME/tf-ae`
- `./run.sh`

TensorRT. The TensoRT results of the 4 models are available in the `tensorrt_ae` folder. The following command lines measure the inference latency of TensorRT:

- Load TensorRT environment (add library path to `LD_LIBRARY_PATH`)
- `cd $PET_HOME/trt-ae`
- `./run.sh`

TASO. The TASO results of the 4 models are available in the `taso_ae` folder. The following command lines measure the inference latency of TASO:

- Load TASO environment
- `cd $PET_HOME/taso-ae`
- `./run_e2e.sh`

PET. The PET results of the 4 models are available in the `pet_ae` folder. The following command lines measure the inference latency of PET:

- `cd $PET_HOME/pet-ae`
- `./run_e2e.sh`

A.8.2 Operator-level performance (E2)

This experiment reproduces Figure 12 in the paper. The scripts are available in the `operator_ae` folder. The experiments of TVM and Ansor will take a very long time to search different mutation kernels.
cuDNN/cuBLAS. The following command lines measure cuDNN/cuBLAS results for the 4 operator-level benchmarks:

- `cd operator_ae/cudnn`
- `./run.sh`

TVM & Ansor. The scripts in `operator_ae/autotvm` and `operator_ae/ansor` search the kernels for the 4 operator-level benchmarks using TVM and Ansor, respectively.

A.8.3 Different optimization policy (E3)

This experiment reproduces Figure 13 in the paper. The scripts are available in the `pet-ae` folder. The following command lines measure the results:

- `cd $PET_HOME/pet-ae`
- `./run_policy.sh`

A.8.4 Different heuristic parameters (E4)

This experiment reproduces Figure 14 in the paper. The scripts are available in the `pet-ae` folder. The following command lines measure the results:

- `cd $PET_HOME/pet-ae`
- `./run_param.sh`

A.8.5 Searching time (E5)

This experiment reproduces Section 8.6 in the paper. The scripts are available in the `pet-ae` folder. The same commands for PET in E1 print the searching time at the same time.

- `cd $PET_HOME/pet-ae`
- `./run_e2e.sh`

Note that our evaluation platform for AE has different CPUs from the platform we used for the paper so that the searching time could be different. Nevertheless, they should be within the same scale.

Privacy Budget Scheduling

Tao Luo*
Columbia University

Mingen Pan*
Columbia University

Pierre Tholoniat*
Columbia University

Asaf Cidon
Columbia University

Roxana Geambasu
Columbia University

Mathias Lécuyer
Microsoft Research

Abstract

Machine learning (ML) models trained on personal data have been shown to leak information about users. Differential privacy (DP) enables model training with a guaranteed bound on this leakage. Each new model trained with DP increases the bound on data leakage and can be seen as consuming part of a *global privacy budget* that should not be exceeded. This budget is a scarce resource that must be carefully managed to maximize the number of successfully trained models.

We describe *PrivateKube*, an extension to the popular Kubernetes datacenter orchestrator that adds privacy as a new type of resource to be managed alongside other traditional compute resources, such as CPU, GPU, and memory. The abstractions we design for the privacy resource mirror those defined by Kubernetes for traditional resources, but there are also major differences. For example, traditional compute resources are replenishable while privacy is not: a CPU can be regained after a model finishes execution while privacy budget cannot. This distinction forces a re-design of the scheduler. We present *DPF* (*Dominant Private Block Fairness*) – a variant of the popular Dominant Resource Fairness (DRF) algorithm – that is geared toward the non-replenishable privacy resource but enjoys similar theoretical properties as DRF.

We evaluate PrivateKube and DPF on microbenchmarks and an ML workload on Amazon Reviews data. Compared to existing baselines, DPF allows training more models under the same global privacy guarantee. This is especially true for DPF over Rényi DP, a highly composable form of DP.

1 Introduction

Increasing evidence suggests that machine learning (ML) models trained on sensitive, personal information – such as auto-complete models trained on users' emails – expose individual entries from their training sets [8, 57]. Despite the evidence, there is an increasing trend to push models to end-user devices for faster predictions [6,27,54], share them across teams in a company [36, 56] and even externally [2, 43].

Differential privacy (DP) [15] promises to enable safe sharing of models by providing solid guarantees regarding the exposure of individuals' data through these models. DP randomizes a computation over a dataset (e.g. training one model) to bound the leakage of individual entries in the dataset through the output of the computation (the model). Each new DP computation increases this bound over data leakage, and can be seen as consuming part of a *global privacy budget* that should not be exceeded. DP is *mature algorithmically*: most popular ML algorithms have been adapted to *individually* enforce the DP guarantee. There are also libraries that implement these algorithms, including TensorFlow Privacy [21], Opacus for PyTorch [18], and multiple libraries for statistics [20, 29, 47].

Comparatively, DP research is *primitive on systems* that enforce a global DP guarantee across *multiple* DP algorithms. Indeed, enforcing a global DP guarantee creates scheduling challenges that have never been addressed in the literature. For example, given a dynamic ML workload of multiple models trained on the same user data stream, how should the global privacy budget be allocated to maximize the number of models that are successfully trained with DP? Recently, we presented Sage, an incipient design of an ML training platform that maintains a global DP guarantee for a dynamic workload of ML pipelines operating on a continuous data stream [35]. Our key contribution was to show that by splitting the data stream into *blocks* (for example by time), enforcing a global DP guarantee over the entire stream reduces to enforcing the guarantee on each block. This showed at a basic level how to operationalize a global DP guarantee for a dynamic ML workload. but left the challenging questions related to scheduling unresolved. Moreover, our block notion was rudimentary, supporting only limited DP semantics (Event DP, which offers non-ideal protection [33, 41]) and basic DP composition methods (which scale poorly with the number of models).

In this paper, we present *PrivateKube*, a plug-in extension to the popular Kubernetes workload orchestrator that can be used to schedule global privacy budgets for a dynamic workload of DP ML pipelines akin to Sage's. The key insight is to (1) generalize the notion of private blocks to support a

*First co-authors of the paper with equal, complementary contributions.

wider range of DP semantics and composition methods, and (2) incorporate private blocks as *a new, native resource* into Kubernetes, alongside traditional compute resources (such as CPU, GPU, and RAM), so they can be scheduled uniformly. Despite intuitive correspondence of our privacy abstraction to Kubernetes abstractions for traditional resources, there are also significant semantic differences that force us to redesign the scheduling at a fundamental, algorithmic level.

Specifically, private blocks differ from traditional computing resources in two key dimensions. First, once a portion of a private block is allocated to a task, it can never be recuperated. Second, in many use cases, the utility of using private blocks is a step function: if a task has enough privacy budget it can make progress, but if it does not have sufficient budget, its accuracy can be affected in complex ways and it is often preferable to wait to accumulate enough budget before proceeding. These two properties invalidate assumptions typically made by scheduling algorithms for traditional computing resources, such as the popular DRF [19], which we show loses the max-min fairness property if applied directly to private blocks. In fact, we find that the very definitions of standard game-theoretical scheduler properties require change to apply to the characteristics of the privacy resource.

We develop a new algorithm for scheduling private blocks, called DPF (Dominant Private block Fairness). DPF treats each private block as a *separate resource* that can be demanded (or not) by tasks. Different tasks can demand different private blocks, creating heterogeneous resource demands and pointing to multi-resource scheduling algorithms, such as DRF [19], as a basis for DPF. Similar to DRF, DPF allocates private blocks to the user that has the minimal *dominant private block share* – the maximum privacy budget requested by a user across the private blocks. Different from DRF and other related scheduling algorithms [32, 49], DPF releases privacy budgets progressively into the blocks, to ensure that future pipelines have access to the privacy resource in accordance to a fairness policy. Moreover, DPF allocates requested budgets all-or-nothing to ensure that pipelines can achieve their accuracy goals. We prove that DPF satisfies several important game-theoretic properties: sharing incentive, strategy-proofness, dynamic envy-freedom (a variant of traditional envy-freedom), and Pareto efficiency.

We evaluate PrivateKube on microbenchmarks and a workload on Amazon Reviews data. We find that: (1) DPF grants more pipelines than baseline policies at a small cost in delay; (2) stronger DP semantics (such as User DP) require more budget and data, increasing the need for judicious budget allocation as with DPF; (3) adapting DPF to Rényi DP [42], the state-of-the-art composition method, enables allocation of either many more or much larger pipelines, and (4) our native integration of the privacy resource into Kubernetes lets us easily adapt the Grafana compute resource monitor to track privacy usage on par with compute usage.

Overall, this paper is the first to pose these questions: (1) what are the characteristics of the "privacy resource" in ML workloads, (2) how should scheduling algorithms support this resource, and (3) what kinds of game-theoretical properties can be guaranteed for this resource? The answers, which form our primary contributions, are: (1) the abstraction of the privacy resource as dynamically-arriving, non-replenishable private blocks, (2) the DPF algorithm, and (3) the theoretical properties of DPF. All these are integrated into real systems, Kubernetes and Kubeflow, in a prototype that we have open-sourced: https://github.com/columbia/privatekube.

2 Threat Model and Background

2.1 Threat Model

We are concerned with the sensitive data exposure that may occur when pushing models trained over user data to untrusted locations, such as mobile devices [6, 27, 54], model stores that are widely shared among teams in a company [36, 56], or even opened to the world via prediction APIs [2, 43]. Our focus is not on singular models, pushed once, but rather on workloads of many models, trained periodically over increasing data from user streams. For example, a company may train an auto-complete model daily or weekly to incorporate new data from an email stream, distributing the updated models to mobile devices for fast predictions. Moreover, the company may use the same email stream to periodically train and disseminate multiple types of models, for example for recommendations, spam detection, and ad targeting. This creates ample opportunity for an adversary to collect models and perform *privacy attacks* to siphon personal data.

Two classes of privacy attacks are particularly relevant: (1) *membership inference*, in which the adversary infers whether a particular entry (e.g., user) is in the training set based on either white-box or black-box access to the model and/or predictions [4, 17, 28, 57]; and (2) *reconstruction attacks*, in which the adversary infers unknown sensitive attributes about entries in the training set based on similar white-box or black-box access [8, 14, 16]. We aim to ensure that an entry's *participation* in a company's model *does not increase the risk* of an adversary learning something about that entry.

Of particular concern are attacks that can access *multiple* models or statistics trained on the same or overlapping portions of a data stream. While individually these may leak limited information about specific entries, together they may leak significant information, especially when combined with side information about an entry. Consider two statistics: (1) average value of a sensitive column s (say representing user salary); and (2) average value of column s across entries whose ID differs from "1234." Individually, they reveal nothing specific about any entry in a dataset. Together, they reveal the value of sensitive column s for entry "1234." This is a trivialized example in which the queries are ideally chosen and the adversary has access to ideal side-information about their target: the ID. However, research in more practi-

cal settings has shown that releasing multiple (versions of) ML models trained over overlapping datasets increases the attacker's membership inference power compared to releasing just one [61]. Moreover, many pieces of information, such as demographic traits and locations, can be pieced together to uniquely identify individuals and used as side information in such attacks [4, 12, 45]. Thus, a significant data exposure threat stems from the repeated release of models/statistics from overlapping portions of a stream.

2.2 Differential Privacy

DP is known to address the preceding attacks [8, 16, 31, 57]. At a high level, membership and reconstruction attacks work by finding data points (which can range from individual events to entire users) that make the observed model more likely: if those points were in the training set, the likelihood of the observed output increases. DP prevents these attacks by ensuring that no specific data point can drastically increase the likelihood of the model outputted by the training procedure.

To prevent such information leakage, DP introduces *randomness* into the computation to hide details of individual entries. A randomized algorithm $\mathcal{Q}: \mathcal{D} \to \mathcal{V}$ is (ε, δ)-DP if for any neighboring datasets $\mathcal{D}, \mathcal{D}'$ that differ in one row and for any $\mathcal{S} \subseteq \mathcal{V}$, we have: $P(\mathcal{Q}(\mathcal{D}) \in \mathcal{S}) \leq e^{\varepsilon} P(\mathcal{Q}(\mathcal{D}') \in \mathcal{S}) + \delta$. Parameters $\varepsilon > 0$ and $\delta \in [0, 1]$ quantify the strength of the privacy guarantee: small values imply that one draw from such an algorithm's output gives little information about whether it ran on D or D'. The *privacy budget* ε upper bounds an (ε, δ)-DP computation's privacy loss with probability (1-δ).

A key strength of DP is its *composition* property, which in its basic form, states that the process of running an $(\varepsilon_1, \delta_1)$-DP and an $(\varepsilon_2, \delta_2)$-DP computation on the same dataset is $(\varepsilon_1 + \varepsilon_2, \delta_1 + \delta_2)$-DP. Therefore, privacy loss accumulates linearly with the privacy loss of each computation. Composition lets one account for the privacy loss resulting from a sequence of DP-computed outputs, such as the release of multiple models. It is thus critical for enforcing a global DP guarantee. There are more advanced forms of composition, such as Rényi DP [42], which permit much tighter analysis of cumulative privacy loss (sublinear). We discuss those in the latter parts of the paper, because they are vital to a well-performing globally DP system, but for the next two sections we assume basic composition for simplicity.

Multiple DP mechanisms exist, such as the Laplace and Gaussian mechanisms. They add noise to the computation from a Laplace/Gaussian distribution scaled by a function of ε, δ, and the sensitivity of the computation. The noise scale depends linearly in $1/\varepsilon$ and at most logarithmically in $1/\delta$. When enforcing a global DP guarantee, which we denote in this paper as $(\varepsilon^G, \delta^G)$, both parameters become "resources" that must be allocated among the individual computations to ensure that cumulatively the computations do not exceed either. However, because individual computations are much more sensitive to the allocated ε than to δ, throughout this paper we will focus on ε^G as the sole global resource to schedule. In evaluation, we set the individual δ requested by each pipeline small enough in comparison to δ^G (10^{-9} and 10^{-7}, respectively) such that ε^G is always the bottleneck.

The DP semantic can be instantiated at multiple granularities, the difference being what a "row" corresponds to. *Event DP* enforces DP on individual data points (e.g., individual clicks). *User DP* enforces DP on all data points contributed by a user. It is stronger but challenging to sustain when new models must keep training on new data from the same users. *User-Time DP* is a middle-ground that enforces DP on all data points contributed by a user in a given period (e.g., one day).

2.3 Assumptions

Our overarching goal is to *develop infrastructural support for organizations to enforce a global DP guarantee – at Event, User, or User-Time level – across the entire ML workload they operate on sensitive data streams*. This would let organizations control the leakage of personal information through the models. The focus of this paper is on how to orchestrate the global privacy budget across *competing* but *trusted* ML training processes, each of which is assumed to be coded by their programmers to enforce DP. We assume that the programmers are trusted to correctly implement DP training processes and to adhere to the protocols we establish for them. Moreover, we assume that the training processes themselves, plus the compute infrastructure, are trusted. For example, if our scheduler refuses to allocate a requested privacy budget to a training task, the task will not access the data. If the scheduler allocates the task's requested budget, ε, then the training process will not attempt to use more than ε. On the other hand, programmers may be incentivised to achieve higher accuracy for their models by requesting more ε. Therefore, we must provide users with strong incentives to fairly share ε^G.

3 PrivateKube Architecture

PrivateKube is a plug-in extension to the popular Kubernetes workload orchestrator. It can be used to allocate privacy budgets for a dynamic workload of ML pipelines to enforce a global $(\varepsilon^G, \delta^G)$ DP semantic. Our key insight is to incorporate the privacy budget as *a new, native resource* alongside traditional compute resources so developers can manage compute and privacy uniformly. Despite one-to-one correspondence of our privacy resource abstractions to traditional Kubernetes abstractions, there are also significant semantic differences that cause us to re-think scheduling for the privacy resource. This section gives an architectural view of our privacy resource abstraction, with the similarities and differences from Kubernetes' abstractions. §4 then describes *DPF*, the first scheduling algorithm suitable for the privacy resource. §5 presents extensions of DPF to support both Rényi composition and all three DP semantics: Event, User, User-Time. These, too, constitute firsts for the DP systems literature.

3.1 Overview

Fig. 1 shows the PrivateKube architecture alongside the main components of a standard Kubernetes deployment. It

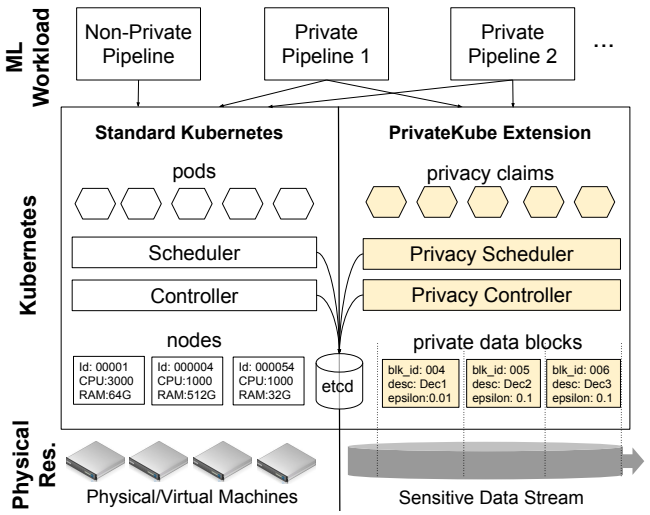


Fig. 1: **PrivateKube architecture.** Clear components are standard Kubernetes. Highlighted components (yellow) are added by PrivateKube.

underscores the correspondence between traditional and privacy abstractions. Kubernetes orchestrates the execution of a *workload* – in our case an *ML workload* consisting of multiple training pipelines – onto the *physical resources* available to the Kubernetes deployment. In standard Kubernetes, the physical resources are physical or virtual machines. The main abstractions that standard Kubernetes provides are: (1) *node*, an abstract representation for a physical or virtual machine; and (2) *pod*, a containerized unit of execution. A pod specifies the container image to execute, plus the type and quantity of compute resources it demands, such as CPU, GPU, RAM, SSD. A node specifies the type and quantity of compute resources it has available. The primary functions of Kubernetes are to: (i) monitor for pods with unsatisfied resource demands (component *Controller* in Fig. 1) and (ii) *bind* each pod to one node that has the demanded resources (component *Scheduler*). Once a pod is bound to a node, the pod's image is executed.

PrivateKube extends Kubernetes to add a new type of physical resource: sensitive data streams. We correspondingly add two new abstractions to Kubernetes: (1) *private data block* and (2) *privacy claim*. Private data blocks (or *private blocks* for short) constitute non-overlapping portions of a sensitive data stream, such as daily windows of data from that stream. Private blocks are the finest granularity at which data can be requested by a training pipeline, and the level at which PrivateKube keeps track of the total privacy loss incurred by an ML workload of multiple pipelines. Private blocks specify the portion of the data they represent (e.g., the start and end times of the corresponding window), plus the privacy budget still available for use in that window. *Privacy claims* are used by training pipelines to demand privacy budget for the private blocks they are interested in. A pipeline specifies in its privacy claims a selector for the private blocks it is requesting (such as the window of time from which they want data), plus the privacy budget it demands for these blocks. The primary functions of PrivateKube are to: (i) monitor for privacy claims

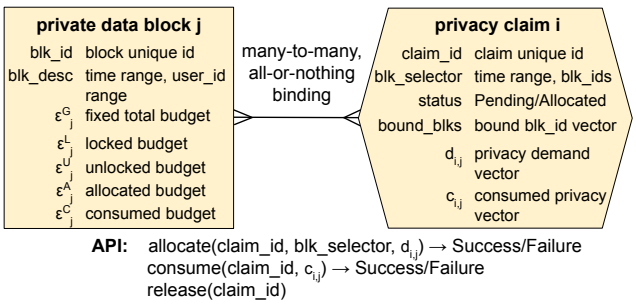


Fig. 2: **PrivateKube abstractions and API.** Some variables are indexed by block (j) or claim (i) for consistency with notation needed in §4.

with unsatisfied private block demands (component *Privacy Controller* in Fig. 1) and (ii) *bind* each privacy claim to the private blocks it demands (component *Privacy Scheduler*).

In a Kubernetes deployment with PrivateKube enabled, the workload may consist of a mix of non-private pipelines (which interact with insensitive data) and private pipelines (which interact with sensitive data). Each pipeline has multiple steps organized in a directed acyclic graph, including steps that read the data, transform it, train models, etc. The non-private pipeline interacts with standard Kubernetes to schedule its steps for execution by registering a pod for each step as soon as the step's inputs are available. The private pipeline interacts not only with standard Kubernetes (to allocate compute resources for each step) but also with PrivateKube (to allocate and consume privacy budget needed to execute the steps on the sensitive data in a privacy preserving way).

3.2 PrivateKube Abstractions

PrivateKube's abstractions are implemented *natively* in Kubernetes using its Custom Resource Definition extension API. Fig. 2 shows the state maintained for each abstraction. As with standard abstractions, state for custom resources is stored in the fault-tolerant, strongly consistent etcd store.

Private Block (Fig. 2, left): This abstraction has three constant fields: a globally unique block id (`blk_id`), a descriptor specifying the portion of the sensitive data stream it represents (`blk_desc`), and the global privacy guarantee PrivateKube is configured to enforce against the entire stream ($\varepsilon^G_j = \varepsilon^G$). PrivateKube supports multiple ways of splitting the stream into private blocks, and splitting determines the type of DP guarantee PrivateKube enforces: Event, User, or User-Time DP. §5 shows how splitting works for each.

Each block j also maintains four variable fields. (1) ε^C_j denotes the budget that has been consumed for the block. We leverage the theory we developed for Sage [35] to justify that enforcing a global ε^G privacy guarantee over the entire stream reduces to ensuring that $\varepsilon^C_j \leq \varepsilon^G_j = \varepsilon^G$ for all blocks j at all times. Thus, when ε^C_j reaches ε^G, we remove private block j from Kubernetes and it no longer represents a resource. (2) ε^A_j denotes the part of block j's budget that has been allocated to some claims but not yet consumed. (3) ε^U_j, called *unlocked budget*, is the unallocated and unconsumed budget made presently available for allocation to privacy claims.

(4) ε_j^L, called *locked budget*, is the unconsumed and unallocated budget not yet made available for allocation. Our DPF algorithm (§4) leverages the last two fields to unlock budget from ε_j^G progressively to ensure that future pipelines have access to the privacy resource in accordance to a fairness policy. Among all fields, the invariant is: $\varepsilon_j^G = \varepsilon_j^L + \varepsilon_j^U + \varepsilon_j^A + \varepsilon_j^C$.

Privacy Claim (Fig. 2, right): This abstraction is used by pipelines to allocate and consume privacy budget from one or more private blocks. When creating a privacy claim, the programmer specifies a selector for the data blocks relevant for their pipeline (`blk_selector`). Typically, this means specifying a time range from which the programmer wishes to obtain data samples (e.g., the past year). PrivateKube then maps this descriptor onto the private blocks that contain data samples from that time range. In addition to the block selector, the programmer also specifies the demanded privacy budget for each of the blocks that match the selector. While often the demanded privacy budget will be uniform across all selected blocks, we allow the programmer to specify a *demand vector*, $d_{i,j}$, with one separate entry for each selected block.

API (Fig. 2, bottom): We implement three functions on privacy claims: `allocate`, `consume`, and `release`. A pipeline can invoke them multiple times on the same claim, and they will be executed sequentially. `allocate` invokes the Privacy Scheduler to allocate privacy demand, $d_{i,j}$, to blocks that match the `blk_selector`. The scheduler will perform the selection, verify that every matching block has sufficient unconsumed and unallocated budget to potentially honor $d_{i,j}$, and if so, binds the matching blocks to the claim. It then adds the claim to its internal list of claims to schedule with the DPF algorithm. The scheduler will ultimately decide to allocate the request, or not. If it does, `allocate` succeeds and the caller is guaranteed that the entire demand vector $d_{i,j}$ has been allocated to the bound blocks. If it does not, the blocks are unbound, and the caller can assume that none of the requested budgets in its demand vector were allocated. `consume` invokes the Privacy Controller to deduct a part of previously allocated budget, $c_{i,j}$, from blocks already bound to the claim. The function is similarly not guaranteed to succeed, for example if the caller is asking to consume more than the budget it has left for a block. `release` invokes the Privacy Controller to reclaim a previous unconsumed allocation to a claim. For example, a pipeline invokes `release` if it decides to stop early and not execute some steps. The Privacy Controller can also invoke `release` if the pipeline that owns the claim fails.

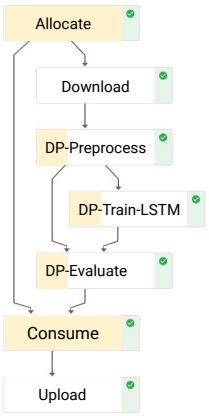
3.3 Example Pipeline

To exemplify usage of PrivateKube's abstractions and API, we describe a pipeline from our evaluation (Product/LSTM in §6.2). It is built in Kubeflow, an ML pipeline orchestrator for Kubernetes, and trains an NLP model on Amazon Reviews to predict a product category. Fig. 3 shows (a) our code in Kubeflow DSL and (b) the pipeline's execution graph. Highlighted are the distinctions between private and non-private versions.

```
@kubeflow.dsl.pipeline
def pipeline(..., blk_selector, eps):
  allocate = privatekube.allocate_op(
     blk_selector, eps)
  download = kubeflow.download_op(
     allocate.output)
  preprocess = amzn_ex.dp_preprocess(
     download.output,
     0.25*eps)
  train = amzn_ex.dp_train_lstm(
     preprocess.output,
     0.5*eps)
  evaluate = amzn_ex.dp_evaluate(
     preprocess.output,
     train.output,
     0.25*eps)
  consume = privatekube.consume_op(
     evaluate.output,
     allocate.output,
     eps)
  kubeflow.upload_op(consume.output)

kubeflow.compile(pipeline)
```

(a) **Pseudocode**



(b) **Execution Graph**

Fig. 3: **Example private Kubeflow pipeline.** Distinctions from the non-private version are highlighted in yellow background.

The pipeline has three processing steps: `Preprocess` tokenizes the reviews; `Train-LSTM` trains an LSTM model with stochastic gradient descent (SGD); `Evaluate` validates that the model passes a baseline accuracy. The Kubeflow runtime executes each step in a separate pod and passes artifacts along the computation graph [34]. If a step fails, its children in the graph will not be launched. An important note for PrivateKube is that in Kubeflow, most steps of a pipeline are pure functions and do not communicate with the outside. Only a few well-defined Kubeflow components do, including: `Download` (loads data from an external source) and `Upload` (pushes an artifact to the serving infrastructure).

Focusing on the *private version* (highlighted parts of Fig. 3), the distinctions from a non-private pipeline are two-fold. First, each step is coded by the programmer to enforce DP. For example, the training step uses DP SGD instead of SGD. The DP steps take an additional parameter: privacy budget (`eps`). The programmer splits `eps` among the steps to enforce `eps` DP at pipeline level. In the example, `dp_preprocess` gets 25% of `eps`, `dp_train` 50%, `dp_evaluate` 25% (Fig. 3a).

Second, the private pipeline interacts with PrivateKube to demand and consume `eps`. This interaction is through drop-in Kubeflow components that we created to wrap PrivateKube's API. This example highlights two such components: (1) `Allocate` and (2) `Consume`, wrappers around `allocate` and `consume`, respectively (Fig. 3b). The protocol is simple: place `Allocate` before any component accessing sensitive data (e.g., `Download`); place `Consume` before any component with externally visible side-effects (e.g., `Upload`). (1) `Allocate` creates a privacy claim and invokes `allocate` on it with a block selector and `eps` privacy budget. If `allocate` succeeds, then `Download` reads the data of the blocks bound to the claim (`bound_blks`) and the training process begins. If `allocate` fails, then `Download` is never launched and the sensitive data never accessed. (2) `Consume` receives the privacy claim from `Allocate` and invokes `consume` on it with a privacy budget equal to the one that was consumed. If `consume` succeeds,

then `Upload` runs and outputs the model artifact. If `consume` fails, then `Upload` is never launched and the model never externalized. Assuming programmers adhere to this protocol (§2.3), the above ensures that PrivateKube controls the privacy loss resulting from externalizing ML artifacts.

3.4 Kubernetes – PrivateKube Distinctions

Despite one-to-one mapping of our abstractions with Kubernetes' – node::private block, pod::privacy claim – there are also semantic differences. First is the level at which we make scheduling decisions. Consider the pipeline from §3.3. The Kubernetes Scheduler performs a scheduling decision for each step. It schedules our `Allocate` and `Consume` pods, as well as the functional pods. In PrivateKube, we decided to ***allocate privacy at the level of entire pipelines***. Indeed, after being allocated compute resources, the `Allocate` pod creates a privacy claim and invokes `allocate` on it. This is when the Private Scheduler makes a scheduling decision for the privacy resource. The privacy claim is then kept for the entirety of the pipeline and passed among its components as needed.

Second, in Kubernetes, the binding of pod to node is many-to-one: one pod can be bound only to one node, but the same node can be bound to multiple pods. In PrivateKube, the binding is many-to-many: a privacy claim can be bound to many private blocks, and the same block can be bound to multiple claims. This leads to a question of atomicity for the binding across multiple blocks. A critical design decision we have made is an ***all-or-nothing semantic*** for scheduling: a pipeline can expect `allocate` on its privacy claim to either fail or guarantee that (1) all the blocks matching the claim's selector were bound to the privacy claim, and (2) for each block, the demanded privacy budget was allocated in full. This decision, which has significant impact on the scheduling algorithm (§4), should be thought of as a plausible assumption, though not the only reasonable one. Multiple use cases justify all-or-nothing. Many DP algorithms have complex interactions with hyper-parameters, such as learning rate and batch size; programmers may want to run on the budget for which those were tuned. Other use cases include the need for comparable models and DP budget searches on a fixed schedule (as proposed in Sage [35]). Furthermore, the ***non-replenishable*** nature of the privacy budget suggests that the scheduler should grant no more budget than a pipeline demanded, to keep as much budget available for future pipelines.

4 DPF Algorithm

Given the preceding integration of private blocks as a new resource in Kubernetes, we now explore how scheduling should work for this resource. Can we achieve for privacy the same types of theoretical guarantees that compute schedulers often achieve? How should scheduling algorithms change given the semantic differences between privacy and compute resources? To obtain initial answers, we focus on max-min fairness guarantees and algorithms that support them.

Our idea is to model each private block as a *separate resource* that must be allocated to different pipelines based on their demands. Demands will differ across pipelines, both in the blocks they select and in the privacy budgets they request for selected blocks. Consider four blocks $(B0, B1, B2, B3)$ and three pipelines requesting: $d_1 = (0.5, 0.5, 0.5, 0.0)$; $d_2 = (0.0, 0.1, 0.1, 0.1)$; and $d_3 = (0.0, 0.0, 0.0, 0.01)$. The pipelines could be: a large model (user embedding) registered before block B3 appeared; a smaller model that needs recent data (news recommendation) registered after B3 appeared; and a daily statistic invoked on B3. Privacy demands being heterogeneous, the four blocks will have heterogeneous capacities left after the pipelines complete.

The preceding formulation points to DRF (Dominant Resource Fairness) [19] – an algorithm that achieves max-min fairness for multiple, heterogeneous compute resources (e.g., CPU, memory) – as a basis for scheduling privacy. However, as we will show, DRF's max-min fairness guarantees do not hold for scheduling privacy. We next describe the limitations of DRF and several variations for privacy scheduling, after which we present the design and analysis of our new algorithm, *DPF* (*Dominant Private block Fairness*).

4.1 Limitations of DRF and Variations

We identify three limitations of DRF with respect to the privacy resource. First, DRF assumes static resources and sometimes even static workloads. In PrivateKube, we focus on a ***dynamic setting***: both pipelines and private blocks arrive to the system dynamically. If we applied DRF on private blocks, at every point in time, DRF would try to consume the entire available budget to satisfy the demands of all present tasks. This would make it violate the sharing incentive guarantee of max-min fairness. A new task arriving to the system that asks for its fair share of privacy budget might not be able to get it, since DRF had already allocated the budget to previous tasks.

Second, DRF, like most scheduling algorithms for compute resources [9, 23–25, 58], assumes these resources are *replenishable*: a resource can grant utility (i.e. via CPU cycles, network bandwidth) indefinitely. For instance, if multiple pipelines need to time-share a CPU core, prior work assumes that if a pipeline was assigned to the core in time interval T_1, the core will naturally be available for other pipeline in time intervals T_2, T_3, etc. and provide them with the same amount of CPU cycles per time slot. In contrast, an individual private block is a ***non-replenishable resource***. If a pipeline is assigned a budget for a particular private block, that budget is consumed forever, and there may not be sufficient budget remaining for another pipeline in that particular block. *Dynamic DRF* [32], a more recent extension of DRF, considers both dynamic settings and non-replenishable resources. Unfortunately, Dynamic DRF has its own limitation, as follows.

Third, as discussed in §3.4, PrivateKube adopts an ***all-or-nothing semantic***: a pipeline is either allocated all of its demanded budget, or none at all. Therefore, pipelines have an all-or-nothing utility function, where they can only be sched-

uled (with a utility of 1) if *their entire demand vector is allocated*, otherwise their utility is 0. Once a pipeline is allocated its entire demand vector, it leaves the system. Having an all-or-nothing utility function departs from both Dynamic DRF and DRF, which assume compute resources with continuous utility. In fact, an all-or-nothing utility function would break the Pareto efficiency of Dynamic DRF and DRF alike, which allocate resources proportionally based on demand (see §7).

4.2 DPF

Due to the ***dynamic*** arrival of pipelines and the ***non-replenishable*** nature of private blocks, we need to *gradually unlock* privacy budget as pipelines arrive to the system, in order to award those pipelines their fair share. Therefore, we need to define a more constrained notion of a *fair share* that divides the budget of private blocks over some particular number of pipelines, or a particular time period. This section presents a version of DPF that defines a fair share over the *first N pipelines that select particular private blocks*, and provides formal fairness guarantees *for those first N pipelines*. For any subsequent pipelines (after the first N) that request a budget for those particular blocks, PrivateKube will *not* guarantee them a fair share, but will make a best-effort to schedule them with leftover budget. §5 discusses a version of DPF that instead of dividing resources by pipelines, divides resources by time intervals, and has weaker fairness guarantees. In both cases we ensure that DPF schedules budget ***all-or-nothing***, so that no budget is wasted on tasks that will not end up being scheduled, thus violating Pareto efficiency.

Algorithm 1 gives pseudocode for DPF. When a new block j is created (ONDATABLOCKCREATION), its per-block budget, ε_j^G, is determined by the fixed global privacy budget ε^G. To ensure that the first N tasks that request j get their fair share, j's budget is initially completely *locked* ($\varepsilon_j^U = 0$).

Recall that each pipeline in PrivateKube has in its privacy claim a privacy demand vector, d, whose entries represent the epsilon demand for the private blocks matching the claim's selector. We define the *privacy budget fair share* of each private block j as: $\varepsilon_j^{FS} = \varepsilon_j^G/N$. DPF guarantees the fair share of a given private block j to the first N pipelines that arrive to the system that have a non-zero demand for j.

We unlock the budget as pipelines arrive (function ONPIPELINEARRIVAL): a new pipeline i that requests budget from a particular block j unlocks ε_j^{FS} of that block's budget, up until all the block's budget is unlocked. The scheduler's responsibility is to allocate the total unlocked budget (ε^U) among the different pipelines.

To determine which pipeline gets scheduled first, the scheduler maintains a sorted list of the waiting pipelines, based on their *dominant private block share*. This is defined as the maximum demand within each pipeline's demand vector:

$$\text{DominantShare}_i = \max_j \frac{d_{i,j}}{\varepsilon_j^G}, \quad (1)$$

where $d_{i,j}$ is the demand for block j of pipeline i and ε_j^G is the total budget of private block j. The scheduler sorts pipelines by their dominant private block share, with the smallest share ranked first (function ONSCHEDULERTIMER). If there are one or more pipelines that have the same dominant private block share, DPF will sort them by taking the smallest of the second-most dominant private block share of each pipeline, followed by the smallest third-most dominant share, etc.

DPF tries to allocate pipelines based on their order in the list. It tries to allocate *all of the demanded privacy budget vector of the pipeline at once*. If it cannot allocate the pipeline fully (function CANRUN returns false), then it moves to the next one in the list, until it reaches the end of the list.

Example. Fig. 4 shows an example run of DPF with three pipelines and two private blocks. Suppose the fair share (ε^{FS}) of each block is equal to 1. Pipeline 1 (P_1) arrives at $t = 1$, then P_2 and P_3 at each time unit. The demand vector of P_1 is $d_1 = (0.5, 1.5)$, while the vector of P_2 is $d_2 = (1.0, 1.0)$ and P_3's demand is $d_3 = (1.5, 1.0)$. The bottom of the figure depicts the state of of DPF's sorted list at each time unit, where

Algorithm 1 DPF (max-min fairness for first N pipelines).

```
# Config.: (ε^G, δ^G) global DP guarantee to enforce.
function ONDATABLOCKCREATION(block index j)
    ε_j^G ← ε^G, ε_j^U ← 0, ε_j^A ← 0, ε_j^C ← 0
end function
function ONPIPELINEARRIVAL(demand vector d_i)
    for ∀j : d_{i,j} > 0 do
        ε_j^U ← min(ε_j^G, ε_j^U + ε_j^G / N)
    end for
end function
function ONSCHEDULERTIMER(waiting pipelines wp)
    sorted_pipelines ← sortBy(DOMINANTSHARE, wp)
    for i in sorted_pipelines do
        if CANRUN(d_i) then
            ALLOCATE(d_i)
            Run task i, which either consumes d_{i,j} (moving
            it to ε_j^C) or releases it (moving it back to ε_j^U).
        end if
    end for
end function
function DOMINANTSHARE(demand vector d_i)
    return max_{j: d_{i,j}>0} d_{i,j} / ε_j^G
end function
function CANRUN(demand vector d_i)
    return ∀j : d_{i,j} ≤ ε_j^U
end function
function ALLOCATE(demand vector d_i)
    for ∀j do
        ε_j^U ← ε_j^U − d_{i,j}
        ε_j^A ← ε_j^A + d_{i,j}
    end for
end function
```

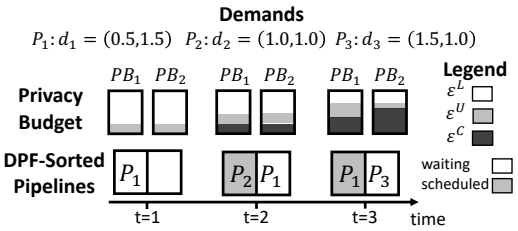


Fig. 4: **DPF example.** DPF is scheduling three pipelines (P_1, P_2, P_3) over two private blocks (PB_1, PB_2), over time. Shows the state of DPF's sorted list, and what portion of each private block is locked (ε^L), unlocked (ε^U), and consumed (ε^C). Assumes budget is consumed instantaneously ($\varepsilon^A = 0$).

the shaded pipeline in the list is the one that is scheduled at that time unit, while the unshaded one remains waiting.

When P_1 arrives it unlocks a privacy budget of 1 in each block. Since it is the only pipeline in the system (and therefore has the minimum dominant resource), the scheduler tries to allocate it a budget. However it is unable to do so, since P_1 requires a budget of 1.5 from PB_2 but only 1 is unlocked.

When P_2 arrives, more budget is unlocked. The dominant resource of P_1 is then the second block (with a demand of 1.5) and the dominant resource of P_2 is either block 1 or 2, each of which has a share of 1. Therefore, the scheduler tries to allocate budget to P_2, and does so successfully. It then tries to allocate budget to P_1, but is unable to (since there is only a budget of 1 left in PB_2). P_1 will have to keep waiting. When P_3 arrives, its dominant share is for block 1 (1.5), while the dominant share for P_1 is block 2 (1.5). Since their dominant share is the same, DPF orders them based on their second highest share, which is 0.5 for P_1 and 1.5 for P_2. Therefore, the scheduler allocates the budget for P_1. P_3 must wait, since the remaining unlocked budget for block 2 is only 0.5.

4.3 DPF Analysis

We prove four properties of DPF: *sharing incentive*, *strategy-proofness*, *dynamic envy-freeness*, and *Pareto efficiency*. We use the same definitions for these properties defined for dynamic environments based on Kash, et.al. [32].

Definition 1 (*fair demand pipeline*)**.** A fair demand pipeline has two properties: (a) the pipeline is within the first N pipelines that requested some budget for all its requested blocks, and (b) its demand for each one of the blocks is smaller or equal to the fair share (i.e. for pipeline i, $\forall j : d_{i,j} \leq \varepsilon_j^{FS}$).

Theorem 1 (*sharing incentive*)**.** A fair demand pipeline is granted immediately.

Proof. Consider a fair demand pipeline i with demand d_i. We proceed by induction over the number of waiting pipelines. *Base case:* no waiting pipelines. $d_{i,j} > 0 \Rightarrow \varepsilon_j^{FS} \leq \varepsilon_j^U$, since ε_j^{FS} is unlocked by d_i. d_i is fair so $d_{i,j} \leq \varepsilon_j^{FS} \leq \varepsilon_j^U$. The pipeline is granted, and no fair pipeline is waiting. *Induction step:* Consider any waiting pipeline k with demand d_k and dominant share DominantShare_k. By the induction assumption no fair pipeline is waiting, so $\text{DominantShare}_k > \varepsilon_j^{FS} \geq \text{DominantShare}_i$. As before, $d_{i,j} > 0 \Rightarrow d_{i,j} \leq \varepsilon_j^{FS} \leq \varepsilon_j^U$, and d_i can be granted. d_i is ordered first so it is granted. □

Theorem 2 (*strategy-proofness*)**.** A pipeline has no incentive to misreport its demand.

Proof. A pipeline has no incentive to ask for more budget than its real demand, because: (a) its utility would not increase if it obtains more budget than it needs, (b) its dominant share will be greater or equal so it can only become less likely to get scheduled. A pipeline also has no incentive to ask for less budget than its real demand, because its utility will drop to zero if it is not allocated its demanded budget. □

Theorem 3 (*dynamic envy-freeness*)**.** A pipeline present at time t cannot envy the allocation of another pipeline present at time t, except if their DominantShares are identical.

Proof. Consider pipeline i. There are two cases. Case 1: i was granted. Its utility cannot improve due to all-or-nothing utility, there is no envy. Case 2: i is waiting. Consider any pipeline j that i envies and is non identical (i and j are strictly ordered by DPF). We show by contradiction that j was granted before i entered the system. Suppose that was not the case. When j was granted: either $\text{DominantShare}_j < \text{DominantShare}_i$ and j could be granted; or $\text{DominantShare}_j > \text{DominantShare}_i$ but i could not be granted while j could. In both bases i cannot be granted from j's allocation, which would give i a utility of zero. i cannot envy j, which is a contradiction. □

Theorem 4 (*Pareto efficiency*)**.** No allocation from unlocked budget can increase a pipeline's utility without decreasing another pipeline's utility.

Proof. Consider pipeline i. If d_i was already allocated, its utility cannot improve due to all-or-nothing utility. If i is waiting, it cannot be allocated from unlocked budget as DPF grants pipelines until no pipeline can be allocated. Allocating d_i would require extra budget, which can only come from another allocated pipeline. Since each allocated pipeline has exactly its requested budget this would decrease its utility from one to zero, which is not Pareto-improving. □

4.4 Best-effort Scheduling for Higher Demands

While DPF only guarantees immediate allocation for fair demand pipelines, the algorithm has a best-effort approach to schedule pipelines that do not have a fair demand. There are two scenarios where pipelines do not have a fair demand. First, a pipeline's demand may be higher than its fair share for at least one block. From Theorem 1, fair demand pipelines always get immediately scheduled. Therefore, if there is any leftover unallocated budget after a fair demand pipeline gets scheduled, that budget can be used to schedule pipelines with higher demands. This budget will not be needed by any future fair demand pipeline, since they unlock a budget equal to the fair share. In Fig. 4, even though pipeline 1 has a higher demand than its fair share for block 1, it still gets scheduled. Second, for the same reason, DPF can safely schedule pipelines that are not among the first N to request budget from some blocks, if there is leftover unallocated budget in those blocks.

Algorithm 2 DPF-T (shows what changes in Alg. 1).

```
# Replace ONPIPELINEARRIVAL with:
function ONPRIVACYUNLOCKTIMER(data lifetime L)
    for ∀ j do
        ε_j^U ← min(ε_j^G, ε_j^U + ε_j^G / L)
    end for
end function
```

4.5 Scheduling Compute Alongside Privacy

DPF only schedules private blocks. However, a pipeline will also need computing resource. Currently, our PrivateKube prototype implements two schedulers: the privacy scheduler (based on DPF) schedules private blocks to private pipelines. The default Kubernetes scheduler schedules traditional computing resources for non-private pipelines, and for private pipelines that have been allocated their privacy budget. DPF's game theoretic properties hold *if* the system is bottlenecked by privacy budget, rather than computing resources. We leave open the problem of scheduling privacy together with computing resources while guaranteeing game theoretic properties.

5 DPF Extensions

We have focused so far on the core version of DPF that unlocks budget based on pipeline arrival, and uses basic DP composition and Event DP. We consider three extensions of DPF to address limitations of this core version: unlocking budget by time, using a stronger DP composition (Rényi) and stronger DP semantics (User and User-Time DP).

5.1 Time-based DPF

Gradually unlocking privacy budget is key to dealing with a non-replenishable resource and a dynamic workload. The preceding DPF algorithm unlocks ε_j^{FS} for each requested block j, whenever a new pipeline arrives. We also define a version of DPF that unlocks budget *over time*, regardless of workload. Many organizations already enforce an expiration period, L, for collected data. In time-based DPF (Algorithm 2), each block gradually unlocks its budget over its lifetime L, and the fair share is defined as $\varepsilon_j^{FS} = \frac{t}{L}\varepsilon_j^G$, where t is the interval of time at which private block budgets are unlocked. The advantage of this version is the budget unlocking is predictable and independent of the pipeline arrival patterns. Moreover, by pacing budget unlocking over the data's lifetime, we ensure that the data will have DP budget remaining while still accessible.

Unfortunately, time-based DPF does not guarantee the sharing incentive. A fair-share pipeline may overlap with many other, smaller pipelines that are ordered first and consume budget when it becomes available, forcing it to wait longer than t or even never be granted.

However, the other three properties are guaranteed by this policy. We briefly sketch out the proofs for each. Strategy-proofness is guaranteed because there is no advantage in demanding more than the real demand, since the pipeline will need to wait longer for the budget to be unlocked. Envy-freeness is guaranteed for the same reason as in the base version of DPF. At any given time DPF will prioritize the pipeline with minimum dominant private block, so a pipeline with a higher dominant resource can only be scheduled earlier than another pipeline by being granted before the other pipeline arrives. Finally, Pareto efficiency is guaranteed by the combination of all-or-nothing utility and allocation.

5.2 DPF with Rényi DP

Rényi DP [42] is an alternative DP definition that is stronger than (ε,δ)-DP for $\delta \in (0,1]$ (in the sense that Rényi DP always implies (ε,δ)-DP but the converse is not true) and is weaker than $(\varepsilon,0)$-DP ($(\varepsilon,0)$-DP always implies Rényi DP). The great benefit of Rényi DP is that it permits convenient composition of multiple mechanisms that scales much better than the basic composition we have been assuming so far. We thus believe it is important for any globally DP system to support Rényi DP, and for this reason we describe our integration of it in PrivateKube. However, the definition and formulas of Rényi DP are more complex than those of (ε,δ)-DP, so we will not attempt to detail them here. Instead, we include a Rényi DP primer in our extended paper [38] and only state here a few facts needed to understand this paper.

Rényi DP Facts. As described in §2.2, DP in general upper bounds the change in the output distribution of a randomized algorithm that can be triggered by a small change in its input. Making $\delta = 0$ in the DP definition in §2.2, we see that $(\varepsilon,0)$-DP puts a *multiplicative bound* on the change in the output distribution: $\forall \mathcal{S}. \frac{P(Q(\mathcal{D})\in\mathcal{S})}{P(Q(\mathcal{D}')\in\mathcal{S})} \leq e^{\varepsilon}$. (ε,δ)-DP loosens this multiplicative bound with an *additive factor*, δ. In contrast to these definitions, Rényi DP puts an upper bound on the *Rényi divergence*, a particular measure of distance between the output distributions: $\text{RényiDivergence}_{\alpha}(Q(\mathcal{D}), Q(\mathcal{D}')) \leq \varepsilon$. We state three facts about this distance and Rényi DP.

First, Rényi divergence is parameterized by a parameter, $\alpha > 1$, hence Rényi DP is expressed in terms of two parameters: (α,ε). Second, for every value of α, there is a direct translation from Rényi DP to (ε,δ)-DP. The formula is: $(\alpha, \varepsilon - \frac{log(1/\delta)}{\alpha-1})$-Rényi DP implies (ε,δ)-DP for any value of $\varepsilon > 0$, $\delta \in (0,1]$, and $\alpha > 1$. Also, (∞,ε)-Rényi DP is equivalent to $(\varepsilon,0)$-DP for any value of $\varepsilon > 0$. Thus, the α parameter can be seen as adding a spectrum between pure $(\varepsilon,0)$-DP and (ε,δ)-DP; from any point on that spectrum, one can reconstruct back the traditional (ε,δ)-DP guarantee. For our work, this means that PrivateKube can use Rényi DP internally while exposing the same (ε^G,δ^G)-DP guarantee externally.

Third, Rényi DP allows tighter analysis of the privacy loss from multiple mechanisms. For example, the scale of the Gaussian distribution required to achieve (ε,δ)-DP depends linearly on $1/\varepsilon$. The scale of the Gaussian required to achieve (α,ε)-Rényi DP depends on $1/\sqrt{\varepsilon}$ (and on α). In traditional DP, when composing (summing the ε's of) k Gaussian mechanisms with the same scale, σ, the composite mechanism is equivalent to a Gaussian mechanism with σ/k scale, so it's

"k times less private." But in Rényi DP, when composing the same k Gaussian mechanisms, the composite mechanism is equivalent to a Gaussian mechanism with $\sigma/\sqrt{k}$ scale, so it's just "$\sqrt{k}$ less private." Thus, Rényi DP scales much better in the number of computations and should enable more pipelines to share the global budget.

DPF with Rényi DP. Our goal is to take advantage of Rényi composition without sacrificing DPF's game-theoretical properties. One option is to pick one point in the Rényi DP spectrum (one value of $\alpha > 1$) and apply DPF as is, internally using Rényi to analyze and compose privacy loss, and ultimately translating the Rényi guarantee back into traditional DP. Unfortunately, when composing multiple, heterogeneous mechanisms (think different σ for Gaussian) in Rényi DP, it is unclear *a priori* which parameter α will ultimately give the best traditional-DP guarantee; this is because both the Rényi analysis of privacy loss and the translation to traditional DP depend on α, in inverse directions (see [38] for details). In PrivateKube, we thus choose to track a set A of $\alpha > 1$ values, and to use one that ultimately gives the best traditional-DP guarantee. As the Rényi DP author shows [42], and as we observed experimentally, fine-grained choice of values is not important, so we select several values based on recommendations from [42]: $A = \{2,3,4,8,...,32,64\}$.

Algorithm 3 summarizes the changes DPF requires to support Rényi. For each private block, j, PrivateKube initializes a *vector of Rényi budgets*, with one entry for each value of $\alpha \in A$, based on the preceding translation formula (function OnDataBlockCreation). Other privacy variables maintained in the block similarly become vectors in α ($\varepsilon^U, \varepsilon^A$, etc.). Moreover, a pipeline's privacy demand also becomes a vector *for each block*: $d_{i,j}(\alpha)$. In practice, a developer will decide on the mechanism and noise scale to use (e.g. Gaussian mechanism with scale σ), based on which a library can compute the Rényi privacy demand vector for the tracked α's. When a pipeline is allocated (function Allocate), the requested budget is deducted from each block, *and for each* α.

With these changes, the question becomes how to schedule over the α vectors. One approach is to treat each $(block, \alpha)$ tuple as a separate resource. Since DPF already supports multiple resources, its game-theoretical guarantees should hold. Indeed, this is how we compute the DominantShare under Rényi: return the maximum demand over all requested blocks and α orders. However, treating each $(block, \alpha)$ tuple as a separate resource does not work when deciding if a pipeline CanRun. Indeed, doing so would allocate pipelines only when enough budget is unlocked for *all* α values. However, recall that in Rényi DP, *any* α with sufficient privacy budget can be translated to an ε, δ-DP guarantee. Requiring *all* to have that would just block progress until the largest α acquires sufficient budget, which removes the benefits of Rényi composition. Instead, we allow allocation of any pipeline in which each requested block has enough unlocked budget $\varepsilon_j^U(\alpha)$ for *any* α (potentially at different α across blocks).

Algorithm 3 DPF-Rényi (shows what changes in Alg. 1).

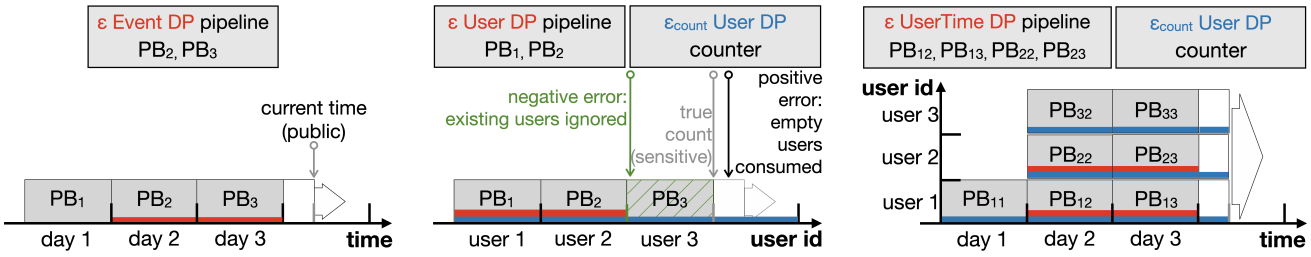
```
# Config.: (ε^G, δ^G): global DP guarantee to enforce;
#   A: Rényi parameters (default: {2,3,4,8,...,64}).
function OnDataBlockCreation(block index j)
    ∀α ∈ A : ε_j^G(α) ← ε^G − log(1/δ^G)/(α−1)
end function
# Either OnPipelineArrival or OnPrivacyUnlock-
# Timer, modified to unlock budget for each alpha.
function DominantShare(demand vector d_i(α))
    return max_{j:d_{i,j}>0} max_{α∈A} d_{i,j}(α)/ε_j^G(α)
end function
function CanRun(demand vector d_i(α))
    return ∀j : ∃α s.t. d_{i,j}(α) ≤ ε_j^U(α)
end function
function Allocate(demand vector d_i(α))
    for ∀j and ∀α ∈ A do
        ε_j^U(α) ← ε_j^U(α) − d_{i,j}(α)
        ε_j^A(α) ← ε_j^A(α) + d_{i,j}(α)
    end for
end function
```

Analysis. Under this behavior, the consumed budget at some α values may be higher than the unlocked budget, and even the global one. However, for each block j there will always remain one α such that $0 \le \varepsilon_j^U(\alpha) \le \varepsilon_j^G(\alpha)$. The global $(\varepsilon^G, \delta^G)$-DP guarantee is thus preserved (proof in [38]). Moreover, DPF's four properties (§4.3) can be proven to hold under the following definition of a fair pipeline: $\forall j, d_{i,j}(\alpha) \le \varepsilon_j^{FS}(\alpha)$, where $\varepsilon_j^{FS}(\alpha) = \frac{\varepsilon_j^G(\alpha)}{N}$ (proofs in [38]).

5.3 Supporting Varied DP Semantics

Finally, we detail how we incorporate support for all three DP semantics – Event, User, and User-Time DP – in our private block abstraction. To our knowledge, no one has shown how to support all three with one abstraction, and since we believe that a DP system should support diverse semantics, suitable for different cases, we describe here how we do so.

DP conceals a change between neighboring D and D' that are identical with a row added or removed. This neighboring definition, or what we treat as a row that is added or removed, defines the protection semantic. In *Event DP*, the most common but weakest semantic, D and D' differ in one event (e.g., one click). DP thus conceals the impact of adding or removing one such event (e.g. yesterday's click on a health related post about a specific condition), but since one user can contribute a large number of such events, important aspects of a user's behavior can still leak though DP computations (e.g. repeated clicks related to said medical condition). In *User DP*, the strongest semantic, neighboring datasets differ by all the data of one user. User DP conceals the entire contribution of a user regardless of the amount of data (e.g., many clicks in a health app). This semantic can be challenging to enforce on streams, since users with an exhausted privacy



(a) **Event DP.** Same as Sage [35]. (b) **User DP.** New in PrivateKube. (c) **User-Time DP.** New in PrivateKube.

Fig. 5: **PrivateKube's support for diverse DP semantics.** Shows how the data is split into blocks and how pipelines request them. Light-gray private blocks can be requested by pipelines, white blocks are in-progress. A block's area represents its ε_j^G budget. Red portions are consumed by pipelines, blue by counters.

budget cannot contribute to new computations, even if they generate new data. *User-Time DP* is a middle-ground [33], in which neighboring datasets D and D' differ by the addition or removal of all data from one user in a given time period (e.g., one day). Repeated actions of a user in that time period are protected (e.g., a browsing session with repeated clicks), and newly generated data in the next period can still be used.

Fig. 5 illustrates how we support all three DP semantics in our private block abstraction. It requires instantiating two aspects: (1) how data is split into private blocks and (2) how blocks are requested by the pipelines.

Event DP (Fig. 5a). *(1) Splitting data:* At pre-set time intervals (e.g., a day), the data collected in this interval forms a new private block with a total of ε^G privacy budget. *(2) Requesting blocks:* Because time is public, we always know which past blocks have been created and filled with data. Pipelines registered on PrivateKube can thus request blocks from a time range of interest without risking consuming budget from an empty block. In Fig. 5a, blocks for the first three days are available. The pipeline requests data from the last two days, thereby consuming budget only for those. This design is identical to Sage [35], which supports *only* Event DP.

User DP (Fig. 5b). *(1) Splitting data:* Computing on any user's event must consume DP budget for the entire user; time-based splitting is therefore insufficient because a user's clicks can span large time intervals. Instead, PrivateKube maintains a private block for each (group of) user id(s) that will ever exist in the system, lazily instantiated. New data is added to the block responsible for the corresponding user without changing its remaining DP budget, or to a newly created block if this user is new. For instance, in Fig. 5b, only the first three users contributed data so far.

(2) Requesting blocks: This raises a challenge. Unlike in Event DP, where we know which past blocks have been created and filled with data, in User DP we do not know which users exist in the system at a given time. Knowing that would leak information about which users join when, violating User DP. Instead, PrivateKube maintains a DP counter that estimates, in a user-DP way, the number of users in the system at any time. The counter is updated periodically (e.g., daily) and consumes a bit of DP budget from every block (in blue on Fig. 5b). Since the count is noisy, pipelines requesting user blocks may sometimes overshoot and consume budget from users that do not yet exist (and therefore cannot possibly supply any data). To avoid consuming budget from empty user blocks, our design has pipelines request user blocks based on a *high probability lower-bound of the true count*. This ensures the true count is under-estimated with high probability, so no empty user is wastefully requested. Our extended paper [38] gives the specific formulas to obtain this lower bound.

The counter does consume some ε_{count}-DP budget, which is a configuration parameter of PrivateKube, fixed when PrivateKube is deployed. The budget is deducted once for each data block, upon the block's creation. For example, for Rényi-DP, OnPrivateBlockCreation(j) initializes j's global Rényi budget vector to: $\varepsilon_j^G(\alpha) = \varepsilon^G - \frac{\log(1/\delta^G)}{\alpha-1} - 2\varepsilon_{count}^2\alpha$, where the last term corresponds to the Renyi consumption of the ε_{count}-DP counter. Since DPF always works from this $\varepsilon_j^G(\alpha)$, all DPF properties are preserved.

User-Time DP (Fig. 5c). A middle-ground between Event and User DP, User-Time DP combines both mechanisms. *(1) Splitting data:* Data is split over both user and time; newly collected data is assigned to the block managing the corresponding user and the time range that includes the data creation. Some of the blocks may be empty (e.g. user 1, day 2), but since no new data can ever be added to them once their timeframe passes, there is no cost to the future of using their DP budget now. *(2) Requesting blocks:* Blocks are requested on both time and a continuous DP counter of the number of users. The counter works similarly to User DP, except that the first (smallest time) block for a user id is created when the upper-bound of the user counter reaches this user id. This corresponds to the first time a user may have contributed data.

6 Evaluation

We implemented PrivateKube on Kubernetes 1.17. Our experiments run on Google Cloud with managed GKE on two pools of CPU (n1-standard8 machines) and GPU (n1-standard8 machines with one Tesla K80 GPU) servers. Each pool is autoscaled by Kubernetes up to a cap of 10 servers per pool.

Our evaluation seeks to answer six questions:

Q1: How does DPF compare to baseline scheduling policies?

Q2: How do workload characteristics impact DPF?

Q3: How does Rényi DP impact DPF?

Q4: How does the DP semantic impact model accuracy?

Q5: How does the DP semantic impact DPF?

Q6: Does native integration facilitate tool reuse?

We develop two methodologies. First, we create a simple, controlled *microbenchmark* that helps us explore DPF under varied workload characteristics (Q1, Q2, Q3). Second, we create a *macrobenchmark* consisting of multiple ML pipelines trained on Amazon Reviews [46] to investigate Q1, and Q4-6.

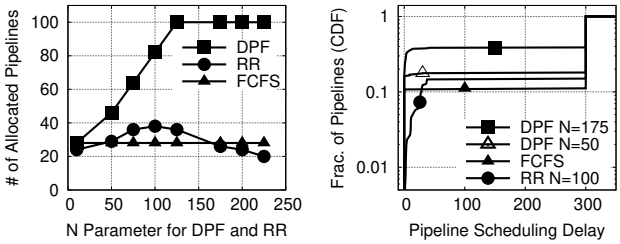
Metrics and Baselines. Across our experiments, we use the following metrics. *Number of allocated pipelines* is the number of pipelines that were successfully allocated their privacy budget throughout the experiment. *Scheduling delay* is the time measured from when a pipeline arrives to the point where it is allocated its privacy budget. *Accuracy* is the percentage of correct classification of a model.

We compare DPF to two baseline scheduling algorithms. *First-come-first-serve (FCFS)* tries to allocate pipelines by their order of arrival on available privacy budget. All the budget is immediately available to pipelines (i.e. unlocked) from the outset. *Round robin (RR)* allocates budget evenly among pipelines that are currently in the system. We implement two versions of RR that correspond to the two versions of DPF. The first one unlocks ε_j^{FS} of budget for each pipeline that arrives that demands a block j, and the second one unlocks budget in the block over time in proportion to its lifetime. For example, if the data lifetime is a year, a third of the budget of a block will be released after four months. This latter policy is similar to the one used by the Sage system [35].

Evaluation Highlights. DPF is able to grant more pipelines than the baselines at the cost of a small delay (Q1), especially over heterogeneous workloads (Q2). Rényi DP enables allocation of either many more or much larger pipelines (Q3). Stronger DP semantics require more DP budget and data (Q4), which increases the need for judicious budget allocation as with DPF (Q5). Our native integration enables reuse of existing tooling for privacy resource management, such as using Grafana to monitor privacy consumption (Q6).

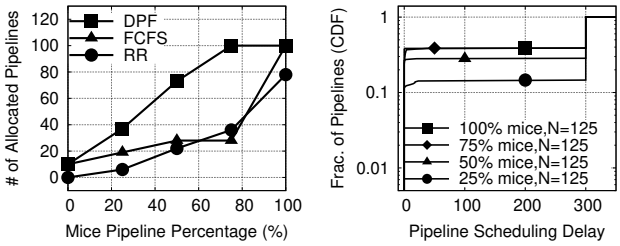
6.1 Microbenchmark (Q1, Q2, Q3)

Our microbenchmarks evaluate the performance of DPF compared to the two baselines. We assume pipeline arrival follows the Poisson process. In the single-block experiment, the pipeline arrival rate is 1 per second. We generate two types of pipelines, mice and elephants, split 75% to 25% by default, with respective demands of $\varepsilon = 0.01\varepsilon^G$, and $\varepsilon = 0.1\varepsilon^G$. In the multi-block experiment, blocks are created every 10 seconds. By default, pipeline's demand ε follows the same distribution as single-block. However, it can either request the last block with probability 0.75, or the last 10 blocks with probability 0.25, independently of the requested ε. We used a load that emphasizes the differences between the policies, where newly arrived pipelines' average demand is 13.5× of the newly generated blocks. This results in the basic composition experiments using an arrival rate of 12.8 per second, and the Rényi experiments using 234.4 per second. If not allocated, pipelines timeout after 300 seconds.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 6: **DPF behavior on a single block.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 7: **DPF with varied workload mix, single block.** (b) DPF N=125.

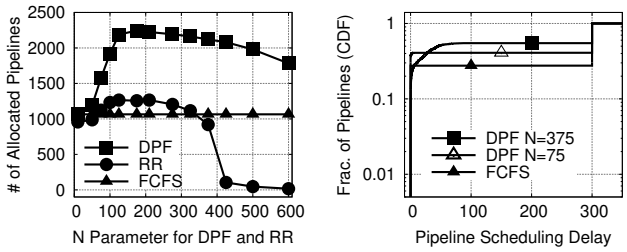
6.1.1 DPF Behavior on a Single Block

We first evaluate the performance of DPF in the simplest possible setup: with a single private block. In this case, the demand vector of each pipeline will only contain one item, and DPF will prioritize the pipeline with the lowest demand.

Fig. 6 shows DPF and RR under different N values, and FCFS. Fig. 6a shows allocated pipelines. With FCFS early elephants take away the budget of many mice, only 28 pipelines are granted. With RR, a low value of N directly unlocks all DP budget, behaving like FCFS. When N is high enough to maintain a large number of mice, but low enough to eventually grant them, RR is able to grant up to 38 pipelines (more than FCFS). At large N RR's proportional allocation creates multiple partially granted pipelines and only 20 are granted. Neither outperforms DPF. When N is equal to 1, the first pipeline unlocks all the budget and DPF behaves like FCFS. At higher values of N, DPF prefers mice over elephants and a higher number of pipelines get allocated, up to the maximum possible of 100. Since DPF never wastes budget on unallocated pipelines it outperforms RR when N is large.

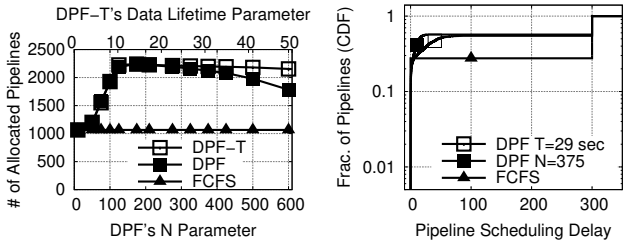
As expected, granting more jobs comes at the cost of increased delay (Fig. 6b shows scheduling delay at notable operating points for each policy). With DPF at $N = 50$ some elephants experience scheduling delays before being granted from unlocked budget. At $N = 175$ some mice wait since ε^{FS} is higher than the mice requests, but only mice are granted.

To summarize, DPF is always able to allocate budget to more pipelines than FCFS or RR. N presents a trade-off between the number of pipelines that are successfully allocated and the scheduling delay the pipelines experience.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 8: **DPF behavior on multiple blocks.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 9: **DPF and DPF-T behavior on multiple blocks.**

6.1.2 DPF Behavior with Mice Percentage

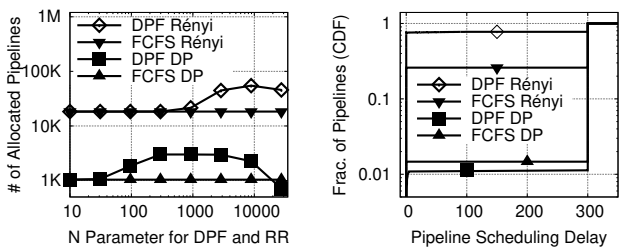
Fig. 7 compares the three scheduling policies under a variable percentage of mice and elephants. At either extreme, all pipelines are identical so DPF and FCFS allocate the same number of pipelines. In this case, the scheduling delay of FCFS is slightly better, since it always immediately schedules these pipelines. However, when there is a mix of pipelines, DPF always allocates more pipelines. RR performance is mixed: for some workloads it is able to allocate slightly more pipelines than FCFS, since it assigns a higher percentage of budget to mice; for others it underperforms FCFS, since it wastes budget on pipelines that are never scheduled.

6.1.3 DPF Behavior on Multiple Blocks

Fig. 8 shows the multi-block experiment results are similar to the single-block experiment. The main difference is that DPF performance with very large N drops, because some blocks do not see enough requests to unlock all their budget. For RR, proportional allocation helps cross-blocks pipelines to be granted (small N), yielding a small improvement over FCFS and $N = 1$ DPF. When $N > 400$, the multiple blocks create more DP budget spread over ungrantable pipelines, and there is no high allocation peak: RR grants collapse while DPF shows a $2\times$ increase over FCFS.

6.1.4 DPF-N vs. DPF-T

Fig. 9 compares DPF-N, the version used throughout the paper, which unlocks budget based on arriving pipelines, and DPF-T, which releases budget based on time (§5). We observe that on low N and T they behave almost identically. This is because DPF-T will release budget on less queried blocks, sometimes allowing multi-block pipelines to be prematurely granted. On large N and T values DPF-T does much better, as all budget is eventually unlocked and some waiting pipelines can be granted, even when no new request is made to the blocks they demanded. Fig. 9b shows the delay for equivalent N and T values.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 10: **Traditional vs. Rényi DP, multiple blocks.** (a) Note log axes. Workload is highly amplified to saturate Rényi. (b) DPF N=8875.

6.1.5 Traditional DP vs. Rényi DP

Fig. 10 compares the DPF algorithm with traditional DP (the default DP composition used in the paper), against Rényi DP, including FCFS with both compositions as a baseline. The results show that switching to Rényi DP results in much better pipeline allocation: Rényi DP allows DPF to allocate more than $17\times$ more pipelines than traditional DP, at their respective peaks. Even FCFS using Rényi DP significantly outperforms DPF with traditional DP. Note that DPF provides a benefit at different values of N for the two compositions, since Rényi DP requires a higher N value to reach the point where DPF starts prioritizing small pipelines. We conclude that switching to Rényi DP leads to much more efficient privacy budget utilization, regardless of the scheduling policy.

6.2 Macrobenchmark (Q1, Q4, Q5)

We use a subset of Amazon Reviews [46] in which users and products have 5 reviews or more, and keep product categories with 1M+ reviews. Each event has a review, timestamp, user, 1-5 rating, and product in one of eleven categories (e.g., books, clothing). We keep the reviews from 01-01-13 to 01-01-18, in total 43.4M reviews from 3.7M users. Tab. 1 specifies our workload: eight ML pipelines and six summary statistics pipelines. For ML, we define four types of models for each of two tasks: product classification (assigns a review to its product category) and sentiment analysis (predicts whether a review is positive). Reviews are embedded using a Wikipedia-trained GloVe [50] except for the fine-tuned BERT model. We run non-DP architecture searches for non-DP and DP pipelines on a 1% hold-out.

We set an accuracy goal for each pipeline: for summary statistics, 5% relative error; for ML models, an accuracy reachable by User DP (e.g., 60% for LSTM/Product). Each pipeline demands the minimum amount of private blocks necessary to reach its goal with $\varepsilon \in \{0.01, 0.05, 0.1\}$ ("mice," i.e. statistics) and $\varepsilon \in \{0.5, 1, 5\}$ ("elephants," i.e. ML models). The demands range from 1 to 500 private blocks. Models use $\delta = 10^{-9}$. The workload draws 75% mice and 25% elephants. Each private block holds one day of data and has $\varepsilon^G = 10$. The experiments replay 50 days of the dataset. Pipelines register

Task	Model	Architecture*	Training
Product classification	Linear	75; 100; [] 1,111 parameters	Optimizer: Adam (for DP, non-DP). DP algo: DP-SGD (Opacus). Epochs: non-DP, event/event-time DP: 15; user DP: 60. Batch: non-DP: 256; DP: $\sqrt{N}$ for N train samples (per [1]). DP clipping: flat, max norm = 1.
	FF††	60; 100; [185, 150] 48,246 parameters	
	LSTM	30; 100; [40]† 23,171 parameters	
	BERT	L 4; H 256; A 4§ 858,379 parameters	
Sentiment analysis	Linear	50; 100; [] 101 parameters	
	FF††	30; 100; [150, 110] 31,871 parameters	
	LSTM	50; 100; [40]† 22,761 parameters	
	BERT	L 4; H 256; A 4§ 855,809 parameters	
Statistics	Reviews: total #, per category #		Laplace. Bounded user contribution: 20/day, 100 in total
	Tokens: total #, avg, stdev		
	Rating: avg		

Tab. 1: **Macrobenchmark pipelines.** ∗: Architecture column: the first line, $x; y; z$, shows the input sequence length (x), embedding size (y), and the list of hidden layers' size (z). The second line shows the number of trainable parameters. ††: Fully-connected feed-forward neural network. †: The LSTM is single directional and has no dropout. §: We use a pretrained BERT model and fine-tune the last transformer layer with over 850K trainable parameters.

with PrivateKube at exponentially distributed time intervals, at a rate of 300 pipelines per day.

6.2.1 Accuracy of Individual Models with DP Semantic

Fig. 11 shows the LSTM's product classification accuracy with increasing data, with no DP and for $\varepsilon \in \{0.5, 1, 5\}$ for each DP semantic. Other pipelines show similar trends. We make two observations. First, DP semantic has a large impact on accuracy for a given DP budget and data size. As expected, Event DP, the weakest semantic, provides the highest accuracy: 73%, 72%, and 72%, for DP budgets of 5, 1, and 0.5 respectively, on 20M datapoints. The larger budgets get close to the non-DP baseline, at 77%. User DP requires larger budgets: the largest reaches 72% while the smallest yields 68%. User-time DP's behavior is closer to, but lower than, Event DP, with accuracies of 72%, 71%, and 70%.

Second, increasing data or budget improves accuracy: the DP models approach the baseline slowly, but can reach it given enough data and DP budget. The relationship between accuracy, data, and budget however is non linear. For event DP with 20M datapoints, increasing the budget from 0.5 to 5 increases accuracy from 72% to 73%, while at 2.5M datapoints the same increase goes from 68% to 71%. This relationship also depends on DP semantics, with low budget models being disproportionately impacted by smaller amounts of data and budget. For user DP for instance, the accuracies go from 68% to 72% for 20M datapoints, and from 57% to 68% for 2.5M.

6.2.2 DPF Behavior with Macrobenchmark

Fig. 12 shows the performance of *DPF with Rényi DP* under our end-to-end workload. Fig. 12a shows the number of granted pipelines under the different DP semantics. We make two observations. First, as expected stronger DP semantics require more private block and DP budget, so fewer pipelines are granted in total: event, user-time, and user DP can grant 13.8k, 10.4k, and 6.7k pipelines, respectively. Second, as before, increasing N helps DPF prioritize later mice over current elephants, increasing the total number of pipelines granted by 67% (event), 75% (user-time) and 17% (user) compared to low N and FCFS. Fig. 12b shows the scheduling delay of user DP for N values of 200 and 400. We see that increase in pipelines granted comes at a reasonable cost in delay.

Fig. 13 shows the cumulative number of incoming pipelines below a given DP size in our workload, as well as those granted under DP and Rényi DP. The DP size of a pipeline is the sum of ε-DP budget over all requested blocks, and is a measure of the total amount of budget requested by the pipeline. The Rényi DP allocates about 29% more pipelines than DP. This difference is *quantitatively* smaller than we obtained in our microbenchmark. However, there is a big *qualitative* difference that this graph also illustrates: while DP only grants mice (cumulative budget below 0.1), Rényi DP is able to also run some elephants: it grants all pipelines with a cumulative budget below 2 and some pipelines up to 10. This confirms that Rényi DP is very valuable in realistic workload settings.

6.3 Kubernetes Tool Reuse (Q6)

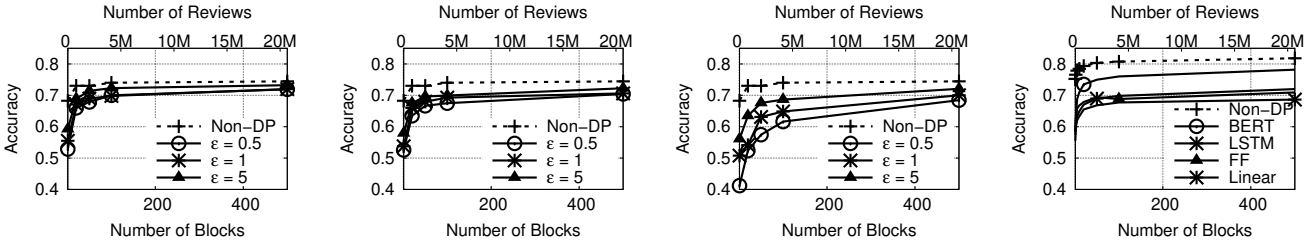
To illustrate the value of integrating with Kubernetes, we extended the Grafana-Kubernetes resource utilization monitor to track privacy usage (screenshot depicted in Fig. 14) with only 150 lines of code. We envision a suite of tools for monitoring privacy, on par with compute resources.

7 Related Work

To our knowledge, there is no work on scheduling DP, but our work builds upon a vast literature in each of these two topics.

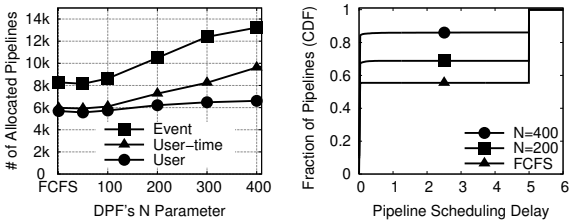
Scheduling. Decades of work exist on scheduling compute resources, such as CPU, network, memory and storage [3,7,9,10,13,19,22–25,30,37,48,49,51,53,58]. Typically, schedulers aim for max-min fairness, achieving both high system-wide utilization and high utility for each tenant. However, compute resources are replenishable, while privacy budget is not: the particular budget consumed by task i will never be available for another task in the future, whereas a CPU core granted to task i can be granted to another task after i finishes.

The two closets to our work are Dynamic DRF [32] and SEQUENTIALMINMAX [49]. Dynamic DRF provides fairness guarantees for agents arriving over time, consuming a fixed set of non-replenishable resources. Unfortunately, the all-or-nothing utility function of private blocks violates Dynamic DRF's Pareto efficiency, since Dynamic DRF would waste budget on tasks that may never get fully allocated. SEQUENTIALMINMAX is an algorithm focused on "indivisible" jobs, or jobs that have an all-or-nothing utility, and thus, similar to DPF, it only assigns resources in a sequential fashion and all-or-nothing fashion ordered by the dominant resource share. However, unlike DPF, SEQUENTIALMINMAX has static jobs, it assumes all resources are replenishable, and it does not



(a) **Product/LSTM: Event DP** (b) **Product/LSTM: User-Time DP** (c) **Product/LSTM: User DP** (d) **All Product: Event DP**

Fig. 11: **Performance of macrobenchmark Product models.** (a)-(c) Accuracy of the product classification LSTM with various DP semantics. (d) Accuracy of all four product classification models with $\varepsilon = 1$ and Event DP. The dotted baseline is non-DP BERT, whose accuracy is highest. The y axes start at 0.4, the accuracy of the naive classifier for this task (i.e. the classifier that returns the most common class).



(a) **Allocated for 3 DP semantics.** (b) **Event DP Scheduling delay.**

Fig. 12: **DPF on macrobenchmark.** $\varepsilon^G = 10$, $\delta^G = 10^{-7}$.

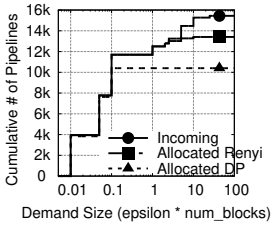


Fig. 13: **Distribution of allocated pipeline sizes.** Event DP, DPF N=400.

consider dynamically arriving resources (private blocks in our case). Therefore, it provides no mechanism for gradually releasing or unlocking these resources, and would not provide a sharing incentive in our setting.

Even under a static setting, standard DRF [19] violates Pareto efficiency with all-or-nothing utility. CARBYNE schedules analytics jobs, which depend on the parallel execution of multiple tasks and have an all-or-nothing utility [24]. However, it assumes replenishable resources.

Differential privacy. There is vast literature on *DP algorithms*, which includes versions of most popular ML algorithms (e.g., SGD [1, 60], Federated Learning [39]) and statistics (e.g., contingency tables [5], histograms [59]). There are also open source implementations available [18, 20, 21, 29, 47]. This literature is at a lower level than PrivateKube, and we leverage it extensively in our pipelines. Some algorithms focus on workloads [26], including on a data stream [11], but they remain very limited, supporting only linear queries.

A few *DP systems* exist, providing DP SQL-like [40, 52] or MapReduce interfaces [55] to static datasets, as well as support for summary statistics [44]. None focuses on workloads of ML pipelines or supports continuous streams of data. The only such system is Sage [35], which introduces block composition for event DP, and proposes a procedure to iteratively increase a model's privacy budget until reaching a good accuracy. However, Sage does not support user and user-time DP, for which we extend block composition, and leaves the question of scheduling unexplored.

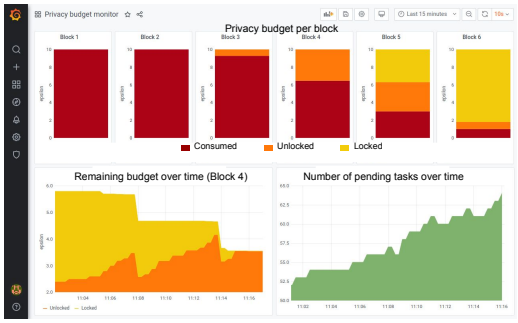


Fig. 14: **Screenshot of Grafana-Kubernetes Privacy Dashboard.**

8 Conclusion

For workloads operating on sensitive user data privacy loss should be carefully orchestrated to enforce a global bound on personal data leakage. This paper presented *PrivateKube*, an extension to the Kubernetes workload orchestrator that adds differential privacy budget as a new native resource to be managed alongside traditional compute resources. PrivateKube incorporates a novel scheduling algorithm, *DPF*, the first one suitable for the unique characteristics of the privacy resource, including its all-or-nothing utility and non-replenishable nature. We show that DPF has desirable theoretical properties, outperforms baseline scheduling algorithms, and that native integration of privacy into Kubernetes can facilitate reuse of existing tools to better manage this scarce resource.

Acknowledgments

We thank Su Ji Park for developing and tuning baseline models for Amazon Reviews. We thank our shepherd, Malte Schwarzkopf, and the anonymous reviewers for the valuable comments. This work was funded by the U.S. Department of Energy (DOE) under award DE-SC-0001234; by the U.S. Army Research Office (ARO) under award W911NF-21-1-0078; by Google Research and Cloud awards; and by Sloan, Microsoft, Google, and Facebook awards.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[2] AWS. Buy and Sell Amazon SageMaker Algorithms and Models in AWS Marketplace. https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-marketplace.html. Accessed: 2020-12-7.

[3] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[4] Michael Backes, Pascal Berrang, Mathias Humbert, and Praveen Manoharan. Membership privacy in microRNA-based studies. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[5] Boaz Barak, Kamalika Chaudhuri, Cynthia Dwork, Satyen Kale, Frank McSherry, and Kunal Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2007.

[6] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A Tensorflow-based production-scale machine learning platform. In *Proc. of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2017.

[7] 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, General Track*, pages 337–352, 2005.

[8] Nicholas Carlini, Chang Liu, Ulfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. arXiv:1802.08232, 2018.

[9] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, Santa Clara, CA, March 2016. USENIX Association.

[10] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[11] Rachel Cummings, Sara Krehbiel, Kevin A Lai, and Uthaipon Tantipongpipat. Differential privacy for growing databases. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[12] Yves-Alexandre de Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific Reports*, 2013.

[13] 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.

[14] Irit Dinur and Kobi Nissim. Revealing information while preserving privacy. In *Proc. of the International Conference on Principles of Database Systems (PODS)*, 2003.

[15] Cynthia Dwork. Differential privacy. In *Automata, languages and programming*. 2006.

[16] Cynthia Dwork, Adam Smith, Thomas Steinke, and Jonathan Ullman. Exposed! A survey of attacks on private data. *Annual Review of Statistics and Its Application*, 2017.

[17] Cynthia Dwork, Adam Smith, Thomas Steinke, Jonathan Ullman, and Salil Vadhan. Robust traceability from trace amounts. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2015.

[18] Facebook. Opacus. https://opacus.ai/. Accessed: 2020-11-10.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[20] Google. Differential Privacy. https://github.com/google/differential-privacy/. Accessed: 2020-11-10.

[21] Google. TensorFlow Privacy. https://github.com/tensorflow/privacy. Accessed: 2020-11-10.

[22] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 157–168, 1996.

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[24] 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.

[25] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 81–97, USA, 2016. USENIX Association.

[26] Moritz Hardt and Guy N Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *Symposium on Foundations of Computer Science*, 2010.

[27] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. of International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[28] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V Pearson, Dietrich A Stephan, Stanley F Nelson, and David W Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*, 2008.

[29] IBM. Diffprivlib. https://github.com/IBM/differential-privacy-library. Accessed: 2020-12-7.

[30] 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.

[31] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *Proc. of USENIX Security*, 2019.

[32] Ian Kash, Ariel D Procaccia, and Nisarg Shah. No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research*, 51:579–603, 2014.

[33] Daniel Kifer, Solomon Messing, Aaron Roth, Abhradeep Thakurta, and Danfeng Zhang. Guidelines for implementing and auditing differentially private systems. *ArXiv*, 2020.

[34] Kubeflow. Overview of Kubeflow Pipelines. https://www.kubeflow.org/docs/components/pipelines/overview/pipelines-overview/. Accessed: 2021-05-11.

[35] Mathias Lécuyer, Riley Spahn, Kiran Vodrahalli, Roxana Geambasu, and Daniel Hsu. Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[36] Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In *Proc. of The International Conference on Predictive Applications and APIs*, 2017.

[37] Yonghe Liu and Edward Knightly. Opportunistic fair scheduling over multiple wireless channels. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 2, pages 1106–1115. IEEE, 2003.

[38] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy Resource Scheduling (extended version). https://github.com/columbia/privatekube, 2021.

[39] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2018.

[40] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2009.

[41] Darakhshan Mir, S Muthukrishnan, Aleksandar Nikolov, and Rebecca N Wright. Pan-private algorithms via statistics on sketches. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2011.

[42] I. Mironov. Rényi Differential Privacy. In *Computer Security Foundations Symposium (CSF)*, 2017.

[43] Model Zoo. https://modelzoo.co/. Accessed: 2020-12-7.

[44] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. GUPT: Privacy preserving data analysis made easy. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[45] Arvind Narayanan and Vitaly Shmatikov. Robust deanonymization of large sparse datasets. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2008.

[46] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*., pages 188–197, Hong Kong, China, November 2019. Association for Computational Linguistics. https://nijianmo.github.io/amazon/index.html.

[47] OpenDP. https://smartnoise.org/. Accessed: 2020-11-10.

[48] Abhay K Parekh and Robert 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.

[49] David C Parkes, Ariel D Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation (TEAC)*, 3(1):1–22, 2015.

[50] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[51] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198, 2012.

[52] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 2014.

[53] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, March 2016. USENIX Association.

[54] Sujith Ravi. On-device machine intelligence. https://ai.googleblog.com/2017/02/on-device-machine-intelligence.html, 2017.

[55] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[56] D. Shiebler and A. Tayal. Making machine learning easy with embeddings. In *Proceedings of the Fourth Conference on Machine Learning and Systems (SysML)s*, 2018.

[57] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[58] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[59] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, Ge Yu, and Marianne Winslett. Differentially private histogram publication. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[60] Lei Yu, Ling Liu, Calton Pu, Mehmet Emre Gursoy, and Stacey Truex. Differentially private model publishing for deep learning. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Santiago Zanella-Béguelin, Lukas Wutschitz, Shruti Tople, Victor Rühle, Andrew Paverd, Olga Ohrimenko, Boris Köpf, and Marc Brockschmidt. Analyzing information leakage of updates to natural language models. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 363–375, New York, NY, USA, 2020. Association for Computing Machinery.

A Artifact Appendix

A.1 Abstract

Our open-source artifact contains the main parts of the PrivateKube system, a scheduling simulator as well as experimental setups to reproduce our evaluation results.

A.2 Scope

The artifact allows to validate the microbenchmark (Fig. 6, Fig. 7, Fig. 8, Fig. 9 and Fig. 10) and the macrobenchmark (Fig. 11 and Fig. 12).

The privacy resource implementation and the DPF scheduler can be reused on any Kubernetes cluster, as well as modified to study other aspects, such as different scheduling algorithms, or the co-scheduling of privacy budgets with computational resources.

A.3 Contents

We release the following parts of the PrivateKube system: the privacy resource implementation (for both DP and RDP); the DPF scheduler (DPF-T and DPF-N); and an example of Kubeflow pipeline using PrivateKube.

We also release the discrete-event simulator, which we leverage to study and prototype scheduling algorithms of privacy and computational resources.

We also provide command line interfaces to reproduce: the microbenchmark; the DP workloads (dataset, models and parameters) used for the macrobenchmark; and the evaluation of the DPF scheduler on the macrobenchmark workloads.

The artifact does not contain: the Grafana dashboard; data ingestion pipelines and other data management infrastructure; nor a cloud-agnostic deployment for Kubeflow pipelines. We can make these components available upon request, but at the time of this publication they are fairly specific to our Kubernetes cluster.

A.4 Hosting

The artifact is available at https://github.com/columbia/privatekube/releases/tag/v1.0.

A.5 Requirements

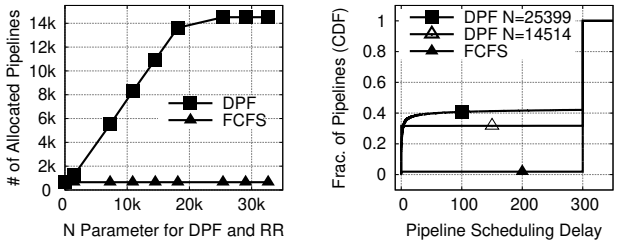
This artifact requires a Kubernetes cluster. The documentation explains how to set up a small cluster on a laptop and details the other requirements. Optionally, an NVIDIA GPU can speed up the evaluation.

The privacy resource implementation, the scheduler and the macrobenchmark do not require anything else. The Kubeflow components and the Kubeflow pipeline example require a Google Cloud Platform Kubernetes cluster with Kubeflow enabled.

It is highly recommended to reproduce the microbenchmark with a beefy machine. It normally takes us several hours to finish it with two 32-core CPUs.

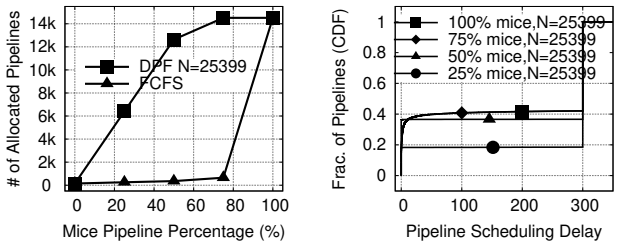
A.6 Additional Evaluation Results

The released artifact supports evaluation of PrivateKube and DPF beyond the results included in the paper. We include here a few of the results that we omitted in the paper.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 16: **Rényi DPF behavior on a single block.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 17: **Rényi DPF behavior with variable workload mix, single block.** DPF N=25,399.

Additional Microbenchmark Results. §6.1 explores in detail the behavior of DPF with basic composition on one or multiple blocks, and under varied mice::elephant ratios. Our artifact allows exploration of these behaviors for DPF with Rényi composition, as well. For thoroughness, we include the corresponding graphs here:

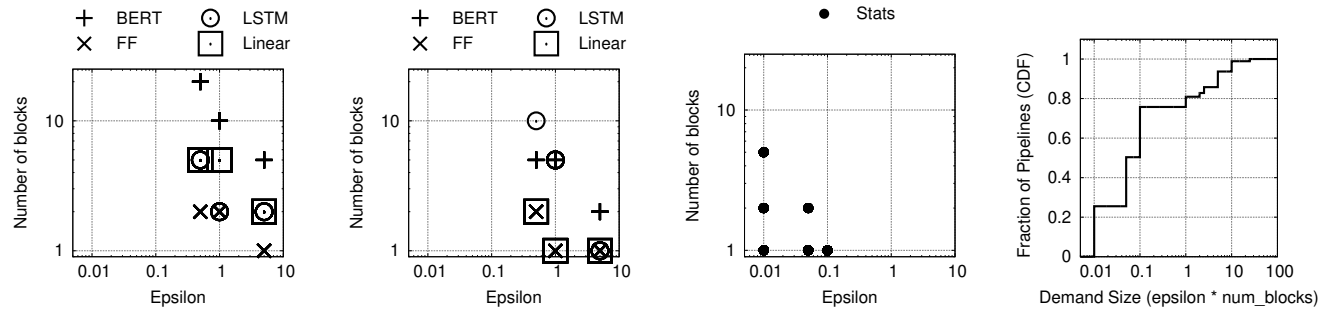
Fig. 16 (Rényi version of Fig. 6) shows that, when the load is amplified appropriately (as described in §6.1.5), Rényi DP can allocate more than 14× more pipelines than traditional DP for the optimal values of N, in the single block setting.

Fig. 17 (Rényi version of Fig. 7) shows that increasing the mice percentage has a similar impact on the number of allocated pipelines for DPF under Rényi DP and traditional DP. Similar to the basic composition results, FCFS also behaves the same as DPF when the percentage of Mice is either 0% or 100%.

Fig. 18 (Rényi version of Fig. 9) shows that, similarly to the traditional DP case, DPF performs better for large N and T. In addition, T outperforms N for large N values, since all budget is eventually locked.

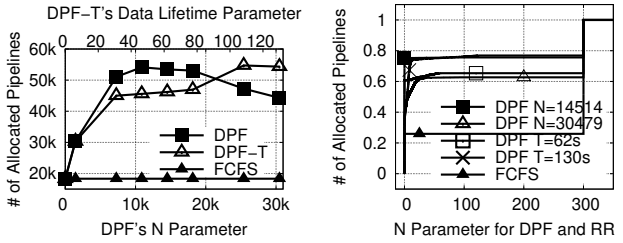
Additional Macrobenchmark Results. §6.2 shows the results from our macrobenchmark evaluation of the Rényi DP instantiation of our system. Our artifact allows evaluation of the macrobenchmark against the traditional DP instantiation as well. For completeness, we include here some of the omitted macrobenchmark results:

First, in the body of the paper, we provided an analytical description of how we chose privacy demands for our macrobenchmark workload. Fig. 15 plots the distribution of these demands for the pipelines in the Event-DP workload. The



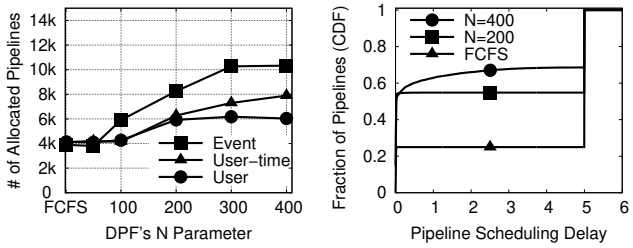
(a) **Product classification demands** (b) **Sentiment analysis demands** (c) **Statistics demands** (d) **Distribution of the demands**

Fig. 15: **Pipeline demands for the Event-DP workload.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 18: **Rényi DPF and DPF-T behaviors on multiple blocks.**



(a) **Allocated for 3 DP semantics** (b) **Scheduling delay for Event DP**

Fig. 19: **DPF behavior on the macrobenchmark workload with basic composition.** The global privacy guarantee is $\varepsilon^G = 10$, $\delta^G = 10^{-7}$.

x-axis of Fig. 15a, 15b, 15c represents the ε demand in terms of traditional DP for product classification, sentiment analysis and statistics pipelines. Each ε also corresponds to the best possible DP-ε for the Rényi DP version of a given pipeline. We can see that the demands are scattered across a wide range of sizes, both in terms of blocks and epsilon, and with finer granularity than the microbenchmark's clear-cut mice and elephants. Finally, Fig. 15d shows how these varied demands are combined to form a workload. This workload gives the incoming load in Fig. 12 and Fig. 13, which evaluate PrivateKube's performance with Rényi DP.

Second, under the same workload, we add here the results from our evaluation of PrivateKube on *traditional DP* with basic composition. Fig. 19 (basic composition version of Fig. 12) shows the performance of DPF for the three DP semantics. We observe the same overall behavior as with Rényi DP: stronger semantics can allocate less pipelines, and larger values of *N* increase the number of granted pipelines. As expected, Rényi DP allocates more pipelines than traditional DP. However, as illustrated in Fig. 13, the pipelines allocated by Rényi DP are qualitatively different from the pipelines allocated by traditional DP. This effect explains why the gap in the number of allocated pipelines is smaller than in the microbenchmark, in particular when the workload contains larger pipelines (such as under User-DP).

Modernizing File System through In-Storage Indexing

Jinhyung Koo
DGIST

Junsu Im
DGIST

Jooyoung Song
DGIST

Juhyung Park
DGIST

Eunji Lee
Soongsil University

Bryan S. Kim
Syracuse University

Sungjin Lee
DGIST

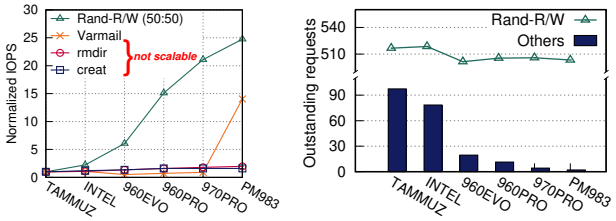
Abstract

We argue that a key-value interface between a file system and an SSD is superior to the legacy block interface by presenting KEVIN. KEVIN combines a fast, lightweight, and POSIX-compliant file system with a key-value storage device that performs in-storage indexing. We implement a variant of a log-structured merge tree in the storage device that not only indexes file objects, but also supports transactions and manages physical storage space. As a result, the design of a file system with respect to space management and crash consistency is simplified, requiring only 10.8K LOC for full functionality. We demonstrate that KEVIN reduces the amount of I/O traffic between the host and the device, and remains particularly robust as the system ages and the data become fragmented. Our approach outperforms existing file systems on a block SSD by a wide margin – 6.2× on average – for metadata-intensive benchmarks. For realistic workloads, KEVIN improves throughput by 68% on average.

1 Introduction

Files and directories are the most common way of abstracting persistent data. Traditionally, storage devices like hard disk drives simply export an array of fixed-sized logical blocks, and file systems abstract these blocks into files and directories containing user data by managing the storage space (*e.g.,* bitmaps) and the locations for the data (*e.g.,* inodes). Whenever files and directories are created or deleted, the file-system metadata, such as bitmaps and inodes, must be retrieved and updated to reflect the newly updated state of the system [3]. Since these persistent data structures must remain consistent, file systems need to employ techniques like journaling to ensure that they are atomically updated [2, 35, 38, 47]. Considering all of these responsibilities, file systems are highly intricate and performance-critical software [27, 37, 45].

However, the architecture of complex and sophisticated file systems that sits on top of storage devices with a simple array-of-blocks interface is ill-suited for today's technology trends. Before processing the actual file operations, file systems have to perform extra operations on on-disk metadata. This not only involves many extra I/Os and data transfers over the host interface, but also causes serious delays owing to I/O ordering [6, 7, 52] and journaling [26, 32]. The end of Moore's Law [50] means that the performance of file systems can no longer scale with faster CPUs. Moreover, the rise of fast storage devices like solid-state drives (SSDs) further exacerbates this problem, shifting the system bottleneck from the device to the host-side software I/O stack.



(a) EXT4 performance (b) Number of outstanding requests

Figure 1: The performance of the EXT4 file system with respect to SSD performance. With the current block interface, the file system exhibits poor performance scalability under metadata and `fsync` intensive workloads.

Figure 1 illustrates this problem by measuring the performance of the EXT4 file system as the performance of the underlying SSD increases: TAMMUZ is the slowest one, while PM983 is the fastest. We run three benchmarks: `creat` and `rmdir` as the metadata-intensive workload and `Varmail` [48] as the fsync-intensive workload. As a performance indicator for the six SSDs, we also run `Rand-R/W` that issues random reads/writes to the SSD which is directly mounted to the host without the file system. The measured throughput in Figure 1(a) is normalized to that of the slowest SSD (TAMMUZ). Under `Rand-R/W` without any metadata operations, the I/O performance increases greatly by up to 24.8× as the SSD gets faster. However, under `creat` and `rmdir`, the file system's performance increases by only 1.6× and 2.0×, respectively. Similarly, for `Varmail`, the measured throughput scales poorly from TAMMUZ to 970PRO (the second fastest SSD); the 14.0× improvement for PM983 is only possible because

the SSD ignores `fsync`[1]. Figure 1(b) shows the number of outstanding requests (measured by `iostat`) averaged across the metadata- and fsync-intensive workloads, and compares it with that of `Rand-R/W`. For `Rand-R/W`, the host system can fully utilize the performance of the underlying SSD by sending a sufficient number of I/Os. Thus, the I/O performance is mostly decided by the SSD performance. However, under the metadata- and fsync-intensive workloads, the file system fails to submit large enough I/Os to fully drive the SSD, in particular when the underlying SSD is fast, which results in much lower throughputs. These results indicate that we cannot increase the overall I/O performance just by improving the performance of the underlying SSD.

To alleviate this problem, we believe it is necessary to rethink the storage interface between the file system and the storage device; an independent improvement at either the file system or the device cannot solve the issue imposed by the legacy block interface. We are not the first to put forward this argument: many prior works have investigated extending the block interface [6, 16] or exposing a file object interface [23]. However, these either have a limited scope (*e.g.,* OPTR [6] on ordering and Janus [16] on fragmentation) or require a significant amount of resources (*e.g.,* DevFS [23] with respect to memory and CPU) that limit their effectiveness.

In this work, we argue that a key-value interface between the file system and the SSD is a better choice over the legacy interface for three primary reasons. First, it is simple and well-understood: it is widely used not only in databases (*e.g.,* key-value stores and backend storage engines for databases [12]), but also as a common programming language construct (*e.g.,* dict in Python). Second, there is great interest in the industry with the development of KV-SSD prototypes [22] and the ratification of key-value storage APIs [46]. Third, the key-value interface is more expressive than the narrow block interface and makes exposing atomicity to support transactions considerably easier. This further enhances application programmability with respect to persistence, as well as, facilitates attaining the elusive goal of syscall atomicity.

To demonstrate the effectiveness of the key-value storage interface, we design KEVIN. KEVIN consists of KEVINFS (**ke**y-**v**alue **in**terfacing **f**ile **s**ystem), which translates the user's files and their inode-equivalent metadata into key-value objects, and KEVINSSD (**ke**y-**v**alue **in**dexed **s**olid-**s**tate **d**rive), which implements a novel in-storage indexing of key-value objects in the SSD's physical address space. We observe that KEVIN has the following quantitative advantages over the traditional file system on a block SSD. First, KEVIN significantly reduces the amount of I/O transfers between the host and the device. On the other hand, a file system on a block device must access its many on-disk data structures before the user's file, incurring high I/O amplification. Second, KEVIN simplifies crash consistency without needing to employ journaling. KEVINSSD supports transactions across key-value `SET`s and `DELETE`s that make it easy to maintain a consistent and persistent state. Lastly, KEVIN is resilient to performance degradation caused by file fragmentation. As a traditional file system ages, its performance drops significantly as its data is dispersed across a fragmented block address space. In KEVIN, however, all persistent data are partially sorted and indexed through a variant of a log-structured merge (LSM) tree that prevents file fragmentation.

We implement KEVINFS in the Linux kernel v4.15 and KEVINSSD on an FPGA-based development platform. We measure our system using both microbenchmark and real-world applications, and compare it to EXT4 [49], XFS [47], BTRFS [40], and F2FS [25]. Our experiments reveal that on average, KEVIN increases system throughput by 6.2× and reduces I/O traffic by 74% for metadata-intensive workloads. These results are further accentuated when the file systems are aged and files are fragmented, highlighting the long-term effectiveness of our approach. Across eight realistic workloads (five benchmarks and three applications), KEVIN achieves 68% higher throughput on average. In summary, this paper makes the following contributions:

- We propose a novel in-storage indexing technique that eliminates the metadata management overhead of file systems by making the storage capable of indexing data.
- We prototype an SSD controller that exposes KV objects through the KV interface and optimize the LSM-tree in-storage indexing engine to efficiently service file system requests with low overhead.
- We develop a full-fledged in-kernel file system in Linux that operates over the KV interface, supporting efficient crash recovery.
- We investigate the effectiveness of KEVIN using micro and realistic benchmarks. Evaluation results show that KEVIN significantly improves I/O performance, especially under metadata-intensive scenarios.

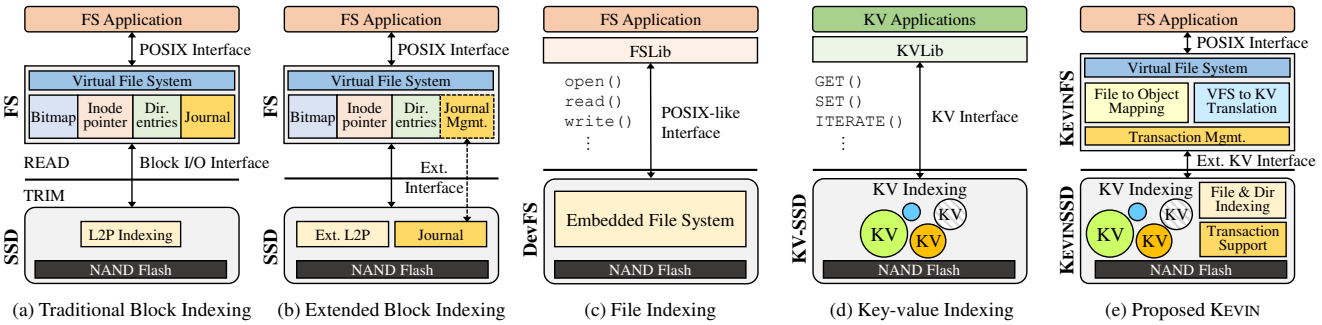
2 Background and Related Work

In this section, we review the traditional block I/O interface, and discuss how our work relates to prior studies [9, 26, 30, 55]. We then describe the basics of the LSM-tree that are fundamental to our indexing algorithm.

2.1 Traditional Block I/O Interface

Existing block storage devices expose the block I/O interface that abstracts underlying storage media as a linear array of fixed-size logical blocks (*e.g.,* 512 B or 4 KB) and provides block I/O operations. Internally, they employ a simple form of in-storage indexing to hide the unreliable and unique properties of the underlying media. HDDs maintain an indirection table to handle bad blocks [17]. Flash-based SSDs contain a flash translation layer (FTL) that maps logical blocks to physical flash pages through the logical-to-physical (L2P)

[1] A number of enterprise-grade SSDs ignore `fsync` by relying on supercapacitors to guarantee durability [52].



(a) Traditional Block Indexing (b) Extended Block Indexing (c) File Indexing (d) Key-value Indexing (e) Proposed KEVIN

Figure 2: Categories of in-storage indexing technologies

indexing table, so as to emulate over-writable media over out-of-place updatable NAND devices and to exclude bad blocks [1] (see Figure 2(a)). To virtualize files and directories over a block device, file systems maintain various on-disk data structures (*e.g.,* disk pointers, bitmaps, and directory entries). However, the management of on-disk data structures is costly, as it involves moderate extra I/O traffic, requires journaling to support consistency, and is vulnerable to fragmentation.

2.2 Review of In-Storage Indexing

Extended block I/O interface. There have been various approaches to enhancing the block I/O interface and the naive L2P-based indexing. Many have suggested custom interfaces with transactional SSDs to ensure consistency at a low cost. While specific designs differ, they commonly aim to offload a journaling mechanism to storage so that a storage controller can keep track of journaling records to avoid double-writing during journal checkpointing [9, 21, 26, 36] (see Figure 2(b)). Some have proposed an order-preserving interface and corresponding L2P indexing design to shorten I/O ordering delays for journaling [6]. Resolving fragmentation of disk pointers (*e.g.,* EXT4's extents) at the storage hardware level was presented by [16]. Those measures have alleviated specific problems (*e.g.,* journaling, ordering, and fragmentation) but have been unable to fundamentally eliminate I/O overhead associated with file-system metadata. And, since the individual strategies have specific designs, applying all of them collectively is also quite difficult.

File indexing & interface. DevFS is a local file system completely embedded within the storage hardware [23] (see Figure 2(c)), DevFS exposes the POSIX interface to a user-level application so that the application can access a file without trapping into and returning from the OS. Since all the metadata operations are performed inside the storage device, I/O stacks and communication overhead can be completely removed. However, moving the entire file system into the storage device has serious drawbacks, such as requiring costly hardware resources and providing limited file system functionalities. As discussed in [23], it is in fact difficult to run a full-fledged file system without adding large DRAM and additional CPU cores to the storage controller. This approach also limits the implementation of advanced file system features, such as snapshot and deduplication. Firmware upgrades to provide new features add maintenance costs.

Key-value indexing & interface. Kinetic HDD and KV-SSD implement parts of a key-value store engine in the storage hardware to accelerate KV clients [13] (see Figure 2(d)). KV-SSDs expose variable-size objects, each of which has a string key, and provide KV operations to manipulate objects. Samsung's KV-SSD indexes KV pairs using the hash because of its simplicity [22], but it suffers from tail latency and poor range query speed [41]. To address this, LSM-tree-based indexing is proposed [18]. Some go a step further by showing that the KV interface can be extended to support compound commands and transactions [24]. But, its FTL design was not explained in detail. Needless to say, KV-SSDs speed up KV clients by doing KV indexing on the storage side. However, since they target KV clients, existing KV interfaces and algorithms are insufficient to index files and directories. For example, Samsung's KV-SSD device based on the hash shows (*i*) slow iteration performance, (*ii*) slow sequential performance (= random), and (*iii*) slow performance on small-value KV pairs. The atomicity and durability support is limited to only a single KV object, which makes it difficult to remove file-system journaling. As a result, naively implementing file systems over KV-SSDs without fundamental design and interface changes may not promise performance improvement.

KEVIN. KEVIN is the natural extension of existing KV-SSDs. While maintaining a lean indexing architecture for a storage controller, our in-storage engine based on an LSM-tree is designed to efficiently index files and directories, together with transaction support to remove file-system journaling (see Figure 2(e)). Over such a KV storage device, we present a new POSIX-compatible file-system design that translates VFS calls and maps files and directories to KV objects. In other words, KEVIN splits the file system into the OS and the device, proposing an extended KV interface to glue the two components efficiently.

2.3 File System over Key-value Store

There have been attempts to run file systems over KV stores [19, 39]. The B^{ε}-tree and LSM-tree algorithms often used in KV stores are write-optimized, so the file system's metadata operations or small file writes are handled efficiently.

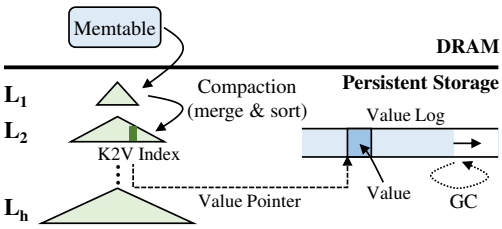


Figure 3: Overall architecture of LSM-tree

BεtrFS in particular employs full-path indexing to improve directory scanning performance. It also adopts zones [56], range deletion [56], and tree-surgery [57] techniques to improve rename and directory deletion operations. Those studies, however, still rely traditional file systems (*e.g.,* EXT4) as a data store and are based on in-kernel (*e.g.,* TokuDB [19]) or user-level (*e.g.,* LevelDB [39]) KV stores. This host-side indexing inevitably causes I/O traffic between the host and the device. The write-ahead logging (WAL) to ensure consistency with KV objects also incurs double-writes like the journaling of traditional file systems. To avoid this, BεtrFS employs late-binding journaling for sequential writes [56]. In addition, TableFS uses EXT4 as a file store to keep big files, so it suffers from fragmentation as the file system ages.

2.4 LSM-Tree Basics

We explain the basics of LSM-tree algorithms [31]. The LSM-tree, as shown in Figure 3, maintains multiple levels, L_1, L_2, ..., L_{h-1}, and L_h, where h is a tree height. Levels are organized such that L_{i+1} is T times larger than L_i. Each level contains unique KV objects sorted by the key. However, the key range of one level may overlap with those of other levels.

A KV object is first written to a DRAM-resident memtable. When the memtable becomes full, buffered KV pairs are flushed out to L_1 in persistent storage. The LSM-tree sequentially writes buffered KV objects to free space in L_1. Once L_1 becomes full, KV pairs of L_1 are flushed out to L_2 and similarly, L_i is flushed out to L_{i+1} when L_i is full. To satisfy the tree property, when flushing out L_i to L_{i+1}, the LSM-tree should perform compaction that merges and sorts the KV objects of L_i and L_{i+1}. The compaction requires many I/Os since it has to read all KV pairs from two levels, sort them by the key, and write sorted KV pairs back to the storage.

To reduce compaction costs, one suggests managing keys and values separately [28]. It appends a value of a KV object to a value log; only a key and a value pointer locating a corresponding value in the log are put into the tree. In this paper, a pair of <key, value pointer> is called a *K2V index*. Because object values do not need to be read during compaction, compaction costs can be greatly reduced, especially when a value is larger than a key. The value log contains obsolete values that must be reclaimed by garbage collection.

For retrieving a KV object, the LSM-tree may look up multiple levels, which involves extra reads due to the fact that the key ranges of the levels can overlap. If a candidate KV object fetched from L_i is not matched with a wanted one, we should move on to L_{i+1} and look up another candidate. To reduce reads for level lookups, bloom filters are used. According to [11], the number of extra reads can be reduced to around one. Once a desired KV object is found, the LSM-tree returns a value to the client because a key and its value are read together. If keys and values are separated, another read is required to retrieve its value stored in the value log.

3 Overall Architecture of KEVIN

This section explains the architecture of KEVIN, focusing particularly on its indexing schema to offload file-system metadata (inode/data bitmaps, disk pointers, and directory entries) to storage. We design two major components of KEVIN, KEVINFS and KEVINSSD, so that they have specific roles: (*i*) the mapping of files and directories to KV objects at the file-system level (§3.1); (*ii*) the indexing of KV objects in flash using the LSM-tree at the storage level (§3.2).

Before explaining the details of our system, we explain the KV interface in Table 1. KEVINSSD exports basic KV operations, `SET`, `GET`, and `ITERATE`, to read, write, and iterate over KV objects. `SET` and `GET` also support partial reads and writes that are useful for dealing with micro reads and writes on a large object. KEVINFS invokes `ITERATE` repeatedly to retrieve KV pairs whose keys are lexicographically equal to or greater than a given `pattern`. KEVINSSD supports transaction commands, `BeginTX`, `AbortTX`, and `EndTX`, exploited by KEVINFS to ensure file-system consistency. A (range) deletion command, `DELETE`, is included to support object deletion or truncation. The length of an object key is variable, but is limited to 256 B. Technically, there is no limit to a value size.

3.1 Mapping of File and Directory

KEVINFS uses only three types of KV objects: *superblock*, *meta*, and *data* objects. A superblock object keeps file system information. While a meta object stores attributes of a file or a

KV Command	Description
`GET(TID,key,off,len)`	Retrieve a value given `key`; if `off` and `len` are given, read `len` bytes of data at offset `off` from a value of `key`
`SET(TID,key,off,len,val)`	Set `key` to hold data `val`; if doesn't exist, create a new one; partially update a value given `off` and `len`
`DELETE(TID,key,off,len)`	Delete an object of `key`; truncate part of a value given `off` and `len`
`ITERATE(TID,pattern,cnt)`	Iterate over objects and return at most `cnt` objects that are lexicographically equal to or greater than `pattern`
`BeginTx(TID)`, `EndTx(TID)`, `AbortTx(TID)`	Start a new transaction with `TID`; commit the transaction; abort the transaction, discard changes (see §5.1)

Table 1: Key KV commands supported by KEVIN

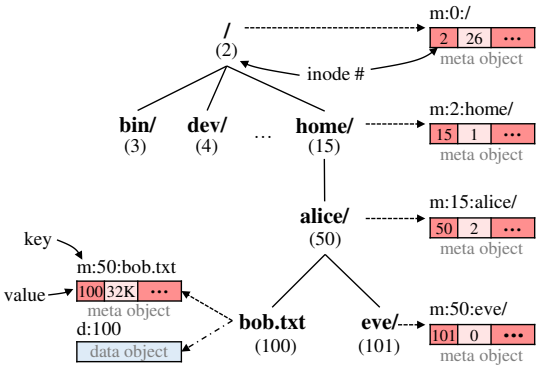


Figure 4: Meta and data objects

directory (*e.g.,* an inode number, size, and timestamps), a data object holds file data. The sizes of a superblock object and a meta object are 128 B and 256 B, respectively. Conversely, a data object can be as large as a file size.

Figure 4 illustrates how files and directories are stored as the form of KV objects. A regular file consists of a pair with one part meta object and the other data object. In contrast to a file, a directory only has a meta object to keep its attributes. For a regular file, the size field of a meta object represents a file size; for a directory, it is the total number of subdirectories and files. All objects are retrieved (`GET`), stored, or updated (`SET`) by KV commands with unique keys. Directory traversals are supported by `ITERATE` as well (explained in detail later).

For assigning a key of an object, KEVIN uses two inode-based key naming rules. **Rule #1:** a meta object key is a combination of (*i*) a prefix '`m:`', (*ii*) an inode number of a parent directory, (*iii*) a delimiter '`:`', and (*iv*) a file or directory name. **Rule #2:** a data object key is a combination of (*i*) a prefix '`d:`' and (*ii*) an inode number of a file. Our naming rules are based on [39], but is extended to deliver the semantics of KV objects so that the storage hardware can index them more efficiently (see §3.2).

Figure 4 shows an example directory tree and associated KV objects. Consider a file `bob.txt` in a directory `/home/alice/`. The inode numbers of `/home/alice/` and `bob.txt` are `50` and `100`, respectively. According to the rule #1, the meta object key is `m:50:bob.txt`. Similarly, following the rule #2, the data object key is `d:100`. As another example, consider a directory `eve/` in `/home/alice/`. A directory has a single meta object only, so a meta object whose key is `m:50:eve/` exists.

KEVINFS has no directory entries, but a list of files and directories belonging to a specific directory can be retrieved by using `ITERATE`. To list up files and directories in `/home/alice/` whose keys start with `m:50:`, KEVINFS creates a new iterator `ITERATE(m:50:,2)` and sends it to the storage, which then returns meta objects with the prefix `m:50:` (*e.g.,* `bob.txt` and `eve/` in Figure 4). To prevent too many objects from being fetched at once (which might take so long), we can specify the maximum object count `cnt` in the `ITERATE` command. In this example, `cnt` is 2, representing the number of subdirectories and files in `/home/alice/`. Traversing an entire file-system tree is easily implemented. The inode number of '`/`' is fixed to 2. KEVINFS retrieves all the files and directories in '`/`' with `ITERATE(m:2:,26)`. By repeating the above steps for directories , it builds up the entire tree.

To efficiently handle small files, KEVINFS packs attributes and data of a file in a meta object together if their size is smaller than 4 KB. This reduces I/Os since a small file can be read or written by one `GET` or `SET` to its meta object.

As an alternative to the inode-based indexing, one might suggest using the full-path indexing [19]. This improves scan performance when using a KV store based on sorted algorithms (*e.g.,* B^{ε}-trees), as it globally sorts the entire file-system hierarchy. While this is beneficial on devices with high seek time such as HDDs, on devices with fast random access like SSDs, its benefits are diminished. On the other hand, the inode-based indexing shows good performance on operations other than directory scans and offers fast directory renaming without techniques such as zones [56] or tree-surgery [57]. Especially as KEVINSSD performs more efficiently when key lengths are short (see §3.3), the inode-based indexing that has shorter key lengths is a more appropriate choice.

3.2 Indexing of KV Objects

KV objects exposed to the file system are managed by our in-storage indexing engine, KEVINSSD, which makes use of LSM-tree indexing. KEVINSSD maps KV objects to the flash, allocating and freeing flash space, and handles read and write requests on objects which are usually done by an FTL. In our system, the FTL only does simple tasks (*e.g.,* bad-block management and wear-leveling). The hardware resources (*e.g.,* CPU cycles and DRAM) saved by disabling such FTL features are used to run our indexing algorithm.

Figure 5 shows the architecture of KEVINSSD. For each level, it maintains a tiny in-memory table (48 MB DRAM for 1 TB SSD) to keep track of KV objects in the flash. Each entry of the table has <start key, end key, and pointer>, where a pointer points to the location of a flash page that holds KV objects; start and end keys are the range of keys in the page. Those key ranges can be overlapped on multiple levels. For fast search operations, all entries are sorted by start keys.

KEVINSSD manages the keys and values of meta and data objects separately. This is a reasonable choice because a key size is much smaller than its value size. This is even true for a small meta object whose value size is 256 B. According to our analysis, the average length of a meta-object key is 32 B, which is 8× smaller than its value. Since keys and values are separated, only K2V indices (*i.e.,* <key, value pointer>) for objects are stored in flash pages, called *key-index pages*, which are separated from their values in other flash pages. Meta and data objects begin with a different prefix ('`m:`' or '`d:`'), so their K2V indices are sorted in different pages.

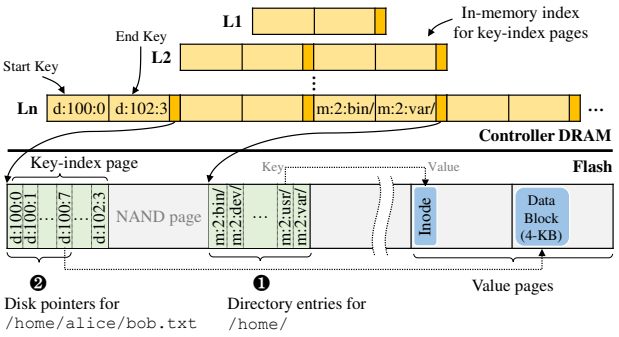


Figure 5: Layout of KV objects in KEVINSSD

Meta object indexing. A meta object key corresponds to a file or directory name in typical file systems, while meta object value is equivalent to an inode. Key-index pages for meta objects thus contain K2V indices, each of which is a pair of <meta object key, value pointer>. This is similar to a directory entry, <file or directory name, inode #>, in a directory file. According to the naming rule #1, K2V indices that belong to the same parent directory are sorted by the parent's inode number and thus are likely to be packed into the same key-index pages (see ❶ in Figure 5).

A list of files and directories belonging to a specific directory can be retrieved quickly if associated K2V indices are fully sorted. To get directory entries in `/`, for example, KEVINSSD requires one flash read (or a few if directory size is huge). However, as mentioned in §2, to reach a wanted key-index page, KEVINSSD should look up multiple levels of the tree. Moreover, if K2V indices are fragmented across multiple levels, more than one flash read is required to build up a complete directory list. We explain how we mitigate this problem in the storage (§3.3) and the file system levels (§4).

Directory entries are updated efficiently. Existing file systems read and write a 4 KB block(s) to modify a list of directory entries. In KEVINSSD, just by writing (`SET`) or removing (`DELETE`) a meta object, we can update directory entries in directories. This removes the necessity of maintaining directory files, thereby eliminating data movement costs.

Data object indexing. In contrast to a meta object, a data object can be very large. Indexing a large object (*e.g.,* 1 GB) as the form of a single KV pair incurs high I/O overhead when a small part of it is read or updated. For example, to update only 512 B of data, KEVINSSD has to read an entire object, modify it, and write it back to the flash, updating its index in the tree. To avoid this, KEVINSSD splits a data object into 4 KB subobjects with unique suffixes and manages them as if they are independent KV pairs. For `/home/alice/bob.txt` whose size is 32 KB, its data object is divided into eight 4 KB subobjects with different suffixes, '`d:100:0`', '`d:100:1`', ..., and '`d:100:7`', in storage. If a small part of a huge object is retrieved or updated, only the corresponding subobject needs to be read from or written to the flash. Please be advised that there is no additional indirection (or index) for subobjects because subobject keys are decided by file's offset.

Since subobject keys and their values are separated, key-index pages hold K2V indices, each of which is a pair of <subobject key, pointer>. As one might notice, a K2V index is like a disk pointer (or extents in EXT4) pointing to a data block in existing file systems. According to the rule #2, K2V indices are sorted by file's inode number and by suffix numbers. Therefore, K2V indices belonging to the same data object (*i.e.,* the same file) tend to be packed in the same key-index pages (see ❷ in Figure 5).

To retrieve 4 KB data from a data object, KEVINSSD should look up levels to find a desired key-index page. Once it is found, KEVINSSD can read a K2V index from the flash with one page read. Then, actual data are read by referring to its pointer. Other K2V indices read together are cached in the controller's DRAM (see §3.3). This reduces lookup costs for future requests. This indexing mechanism is similar to the management of disk pointers (*e.g.,* an extent tree in EXT4). Existing file systems maintain index blocks that contain pointers only, where each pointer points to a data block or another index block. Before reading file data, index blocks must be loaded from a disk.

In KEVINSSD, looking for a K2V index for reading data is done in storage. The update of K2V indices for a data object is done by writing or deleting a data object via `SET` and `DELETE`. Compared to typical file systems that read and write a 4 KB block(s) to retrieve and to update disk pointers, KEVINSSD does not involve any external I/Os to index file data.

3.3 Mitigating Indexing Overhead

As mentioned in §3.2, putting the LSM-tree indexing onto the storage hardware causes extra I/Os, which never happen in typical FTLs using a simple L2P indexing table (which is entirely loaded in DRAM). We introduce three main causes that create internal I/Os and explain how we solve them (see Figure 6). Note that garbage collection occurs both in KEVIN and existing SSD controllers, so it is not explained here.

Compaction cost. Compaction is an unavoidable process and may involve many reads and writes [28]. KEVINSSD manages meta and data objects in a manner that minimizes compaction I/Os by separating keys and values. Particularly, our inode-based naming policy that assigns short keys to data objects lowers the compaction cost because it enables us to pack many subobject keys into flash pages. We go one step further by compressing K2V indices for data objects. Subobject keys have regular patterns (*e.g.,* '`d:100:0`', '`d:100:1`', ...), so they are highly compressible even with naive delta-compression requiring negligible CPU cycles. This reduces the amount of data read and written during compaction. According to our analysis with write-heavy workloads, the write amplification factor (WAF) of the compaction was less than 1.19× under the steady-state condition (see §6.2).

Level lookup cost. The LSM-tree inevitably involves multiple lookups on levels until it finds a wanted KV object (see

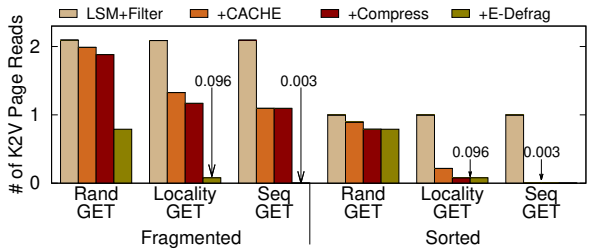


Figure 6: The number of reads per KV request to retrieve a key-index page. The LSM-tree with bloom filters is our default setting. We add each optimization technique one by one to understand their impact. The size of bloom filters is set to 6.5 MB for 40M objects. The cache size is 110 MB.

§2). To avoid useless lookups, KEVINSSD employs small bloom filters. It reduces the number of extra reads for level lookups to around one [11]. To further reduce lookup costs, it also caches popular K2V indices in DRAM. SSDs usually have a large DRAM (*e.g.,* 1 GB for 1 TB SSD) to keep an L2P table, but this large L2P mapping table is unnecessary for KEVINSSD. This enables us to use large DRAM for caching. To increase an effective DRAM size, KEVIN maintains cached K2V indices in the compressed form. To make it searchable, in between compressed indices, we add uncompressed keys sparsely which can then be used as a pivot index for binary searching. This optimization shows its strength with large files. For example, to index a 10 GB file without compression, 45 MB are required for indexing KV pairs, but with compression, only 10.8 MB memory is needed.

Fragmented tree cost. The LSM-tree allows each level to have overlapped key ranges with other levels. Therefore, K2V indices belonging to the same parent directory or file can be fragmented across multiple levels, even they have the same prefix. To retrieve a full list of directory entries or disk pointers, multiple flash pages on different levels must be read. This problem is implicitly resolved by compaction that merges and sorts K2V indices in adjacent levels. KEVIN also provides an offline user-level tool that explicitly triggers compaction in storage. Unlike traditional tools (*e.g.,* `e4defrag` [14]), this does not involve moving the entire file system's metadata and data and is thus much more efficient.

Figure 6 shows the impact of optimization techniques in reducing indexing overhead. For each KV request, we counted the number of page reads required (*i*) to find a key-index page in the LSM-tree and (*ii*) to read that page from flash. The I/O cost of reading a value was not included here. Over KV objects that were fragmented (*i.e.,* unsorted) or fully sorted, we ran three types of queries: random `GET` (= point-query), 90:10 localized `GET` (= point-query), and sequential `GET` (= range-query). Sequential `GET`s over fully-sorted KV objects required almost zero cost to read a key-index page. This is because after the first miss on a specific key-index page, following KV requests were hit by the cached indices. Caching KV indices were also useful when `GET` requests were localized. Regardless of the distribution of KV objects, random `GET`s suffered from extra reads, but even in the worst case, they required about two reads. This is because, with bloom filters, the number of page reads that happen while searching for a key-index page in the tree is theoretically limited to around one, on average [44].

Syscalls	KEVINFS	EXT4
`mkdir`	SET(MO)	W(BB + IB + I + DE)
`rmdir`	DELETE(MO)	W(BB + IB + DE)
`creat`	SET(MO)	W(IB + I + DE)
`unlink`	DELETE(MO + DO)	W(BB + IB + DE)
`setattr`	SET(MO)	W(I)
`write`	SET(DO)	W(BB + D)
`open`	GET(MO)	R(I)
`lookup`	GET(MO)	R(DE + I)
`read`	GET(DO)	R(D)
`readdir`	ITERATE(MO)	R(DE + I)

Table 2: I/O operations of KEVIN and EXT4 for basic syscalls. (MO: meta object, DO: data object, BB: block bitmap, IB: inode bitmap, I: inode, DE: directory entry, and D: data block)

Even with such optimizations, KEVINSSD exhibits slightly slower read performance than block storage devices that do not suffer from any extra I/Os for indexing. However, our entire system exhibits much higher performance than existing systems thanks to the reduction in metadata I/Os. Moreover, while metadata I/Os on existing file systems increase as it ages and gets fragmented, KEVINSSD's indexing cost is maintained constantly through regular compaction of the LSM-tree in storage and other optimizations.

4 Implementing VFS Operations

We describe how KEVINFS implements VFS operations using the KV interfaces. KEVINFS is a POSIX-compatible in-kernel file system and implements 86 out of 102 VFS operations. We summarize the types of I/O operations to handle major file syscalls in Table 2, comparing them with EXT4.

Handling write syscalls. All the write-related syscalls can be handled by two KV commands, `SET` and `DELETE`. It is clear that KEVINFS requires fewer I/O operations than EXT4. This benefit stems from the fact that KEVINFS does not need to modify on-disk metadata. Taking the example of `unlink`, KEVINFS issues two `DELETE` commands to remove a meta object and a data object (which are associated with the file to be deleted) from storage. On the other hand, EXT4 has to update data and inode bitmaps to return a data block as well as an inode. EXT4 needs to update directory entries to exclude the deleted file from the directory.

KEVINSSD does not involve many internal I/Os for `SET` and `DELETE`. `SET` first buffers a KV object in the memtable and then appends to the flash later, leaving the old version if it exists. `DELETE` internally involves a small write to leave a tombstone (4 B) in the tree. Outdated objects (overwritten by `SET`) and deleted objects are persistently removed during compaction, which is not expensive in our design as we manage keys and values separately.

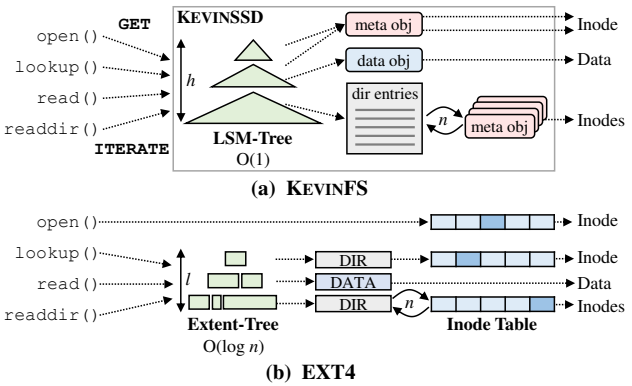


Figure 7: Handling of read syscalls of KEVINFS and EXT4

Handling read syscalls. Read-related syscalls can be implemented by two KV commands, `GET` and `ITERATE`. Regardless of the type of data being accessed, KEVINFS needs to send `GET` or `ITERATE` to a designated meta or data object as shown in Figure 7(a). `open`, which retrieves an inode of a file, can be implemented as `GET` to a meta object. `lookup` is the same as `open` in that, given a full path name (*e.g.,* `/home/alice/`), it retrieves inodes of directory components (*e.g.,* '`/`', '`home/`', and '`alice/`') by sending `GET`s to meta objects. Reading data from a file is also translated into `GET` to a data object. Finally, `readdir` corresponds to `ITERATE`, which retrieves a set of meta objects (*i.e.,* inodes) that belong to the same parent directory.

While the LSM-tree is used as a unified indexing data structure to service all the read-related syscalls in KEVIN, EXT4 relies on several on-disk data structures: an inode table, an extent tree that indexes disk pointers, and a directory file that holds directory entries and their inode numbers (see Figure 7(b)). KEVIN and EXT4 should be comparatively analyzed further because the two systems operate dissimilarly over different data structures. But, KEVINFS benefits from its in-storage indexing; all the I/Os associated with the LSM-tree are performed in storage without any external data transfers.

When opening a file, EXT4 fetches an inode from the inode table by using its inode number as an index. EXT4 requires a 4 KB block read and is faster than KEVINFS that has to look up the LSM-tree before reading an inode.

For `lookup` and `read`, KEVINFS needs to look up the LSM-tree to get locations of meta or data objects (*i.e.,* key-index pages). Similarly, EXT4 needs to search the extent tree to find disk pointers that locate disk blocks for a directory or regular file. Both cases may involve extra reads from the disk. To skip the tree search step for small files, EXT4 embeds a few disk pointers in an inode. KEVINFS cannot avoid the tree search. However, it does not require reading a directory file during `lookup`, and the data of a small file is preloaded when its meta object is read. Thus, for `lookup` and small `read`, the two systems exhibit similar performance.

If a file or a directory is huge, the tree search cost could be high. In KEVINFS, the worst-case I/O cost of looking up the LSM-tree is $O(h)$, where h is the tree height. However, as shown in Figure 6, even under random I/Os, the average I/O cost is not as high as two reads thanks to bloom filters [11]. In EXT4, the worst-case I/O cost of the extent tree is $O(l)$, where l is the height of the tree ($l = 5$ by default). The average I/O cost is $O(log\ n)$, where n is the number of extents for a file which actually decides the tree height. If a file is not fragmented, n is close to 1, and thus the tree search requires less than two reads. However, if it is severely fragmented, the I/O cost could be more than two reads.

Besides the tree lookup cost, KEVINFS has another benefit in that it is never logically fragmented. In EXT4, once a file is fragmented, many pieces of file data are scattered across non-continuous logical blocks. In this case, even when the file is sequentially read, EXT4 has to issue many read requests to the disk [10]. As reported by [16], it badly affects I/O throughput. In KEVINFS, no logical fragmentation happens because each file is represented as an object, not a set of logical blocks. Hence, KEVINFS can always perform sequential reads in big granularity. Also, as explained in §3.3, KEVINSSD shows high sequential I/O performance on subobjects with index caching and compression. As a result, EXT4 generally provides good performance when a file is continuously allocated, but KEVINFS is more resistant to fragmentation.

Finally, `readdir` requires retrieving a full list of directory entries to read the associated inodes. EXT4 offers different performance depending on the degree of inode table fragmentation. If the inodes are allocated together and thus are stored in the same blocks, only few block I/Os are needed to retrieve them. However, if they are highly fragmented, EXT4 suffers from high I/O overhead. In KEVINFS, the inodes (*i.e.*, *values* of meta objects) pointed to by the directory entries are scattered across multiple pages (see ❶ in Figure 5). This inevitably degrades `readdir` performance. To mitigate this, KEVINFS uses a simple tweak that rewrites meta objects to the disk. When meta objects retrieved by `ITERATE` are evicted from the page cache, KEVINFS rewrites them to the disk even if some of them are clean. All of the meta objects that are evicted together are likely to be written to the same flash pages so that the next time KEVINFS can retrieve them quickly without multiple page reads. We plan to study a way to sort meta objects inside KEVINSSD without explicitly rewriting.

5 Crash Consistency

We describe how KEVIN implements transactions to maintain consistency. KEVINFS issues fine-grained transactions by tracking dependency among KV objects so that they are updated atomically (see §5.1), and KEVINSSD supports transaction commands exploited by KEVINFS (see §5.2).

5.1 Maintaining Consistency in KEVINFS

Although an ideal file system would immediately persist data upon a write without any consistency problems, current file

systems follow a compromised model for better performance. That is, file systems provide an explicit interface to the users (*i.e.,* `fsync`) by which users can request a barrier across updates or immediate durability enforcement whenever needed. In addition, file systems such as EXT4 maintain a global transaction comprising all associated blocks with write requests during a time window, and the flush daemon atomically persists them to storage through the journaling. This mechanism prevents the out-of-execution and/or buffering of write that may lead to an inconsistent state for the file system.

KEVIN makes data durable through both user-initiated `fsync` and the flush daemon, but without the overhead associated with the journaling mechanism. This is achieved by KEVINSSD supporting fine-grained transactions.

KEVINFS only builds a transaction associated with dependent KV objects and simply transfers the information to the underlying storage. KEVINSSD then materializes given transactions to the physical medium with an SSD-internal technique (see §5.2). To this end, we extend the KV interface to support three transaction commands: `BeginTX`, `EndTX`, and `AbortTX`. Below is an example of a transaction that manages the KV objects associated with `unlink` in Table 2. KEVINFS can instruct the storage to remove meta and data objects atomically by wrapping KV commands in the same transaction:

```
BeginTX(TID);
DELETE(TID,m:4:c.txt); /* data object */
DELETE(TID,d:5); /* meta object */
EndTX(TID);
```

KEVINSSD guarantees atomicity and durability for a transaction. To ensure file-system consistency, KEVINFS preserves the order between dependent transactions. As a result, KEVINFS ensures the same level of reliability as other journaling file systems, and also offers the following desired properties.

Transaction disentanglement. The performance of a file system suffers from a phenomenon known as transaction entanglement that flushes the entire global transaction when `fsync` is requested for only a part of the buffered data. This not only increases the `fsync` latency, but lowers the effect of write buffering. Some attempted to resolve this issue by splitting a transaction into the smaller ones by files or by subtrees [29, 32]. However, it could not be effective in practice because the transaction disentanglement is impossible when data is shared across transactions. Typical file systems engrave small metadata within a fixed-sized block (*e.g.,* inode/bitmap blocks), and thus chances are high that the metadata updates in a different context happen to the same block.

In contrast, KEVINFS does not maintain any on-disk metadata shared by different files unless they are adjacent in the file system tree (*e.g.,* a parent directory and a file). This nature makes transaction disentanglement easy and likely more effective. KEVINFS basically maintains a single running transaction containing all pending KV commands and sends them at once through periodic flush daemon (a default period is 5s). However, upon `fsync`, KEVINFS forks a small transaction

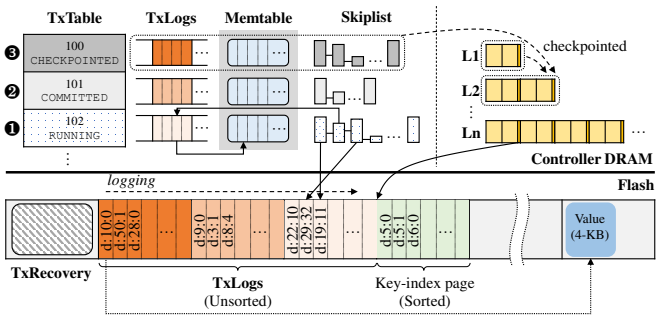


Figure 8: Transaction management of KEVINSSD

that only includes the KV objects associated with the fsynced file, thereby achieving short latency.

Syscall atomicity guarantee. Current journaling file systems do not ensure the atomicity of a syscall. Because the transaction size is limited by the remaining journal size, even a single syscall can be split into multiple transactions in some cases [51]. This is rare but possible and thus the user-level applications should employ another technique (*e.g.,* user-level journaling) to ensure an atomic write over the file system. KEVINFS has no such limitation and thus enforces the atomicity for each syscall by assuring all of the associated KV objects reside in the same transaction.

5.2 Transaction Processing in KEVINSSD

We now explain how KEVINSSD supports transaction commands. Our design is essentially based on journaling but we further optimize it to perform well with KEVINSSD.

Transaction management. Figure 8 shows the transaction management in KEVINSSD. We employ three data structures: a transaction table (*TxTable*), transaction logs (*TxLogs*), and a recovery log (*TxRecovery*). The TxTable keeps the information of transactions, while the TxLogs keep K2V indices of transaction objects. The TxLogs are stored either in the DRAM or in the flash. They are also used to keep track of K2V indices committed to L_1 in the tree. The TxRecovery is used to recover or abort transactions during the recovery.

When `BeginTx(TID)` comes, KEVINSSD creates a new entry in the TxTable, where each entry keeps a `TID`, its status, and locations of K2V indices associated with the transaction. Many transactions can be activated simultaneously as there exist multiple entries in the table. Initially, the status of the transaction is `RUNNING`, which means that it can be aborted in the event of a crash (see ❶ in Figure 8). When subsequent commands belonging to the transaction arrive, KEVINSSD keeps KV indices in the DRAM-resident TxLogs and buffers associated values in the memtable. Once the TxLogs or the memtable becomes full, KV indices or values are logged into the in-flash TxLogs or the flash. All of them are not applied to the LSM-tree yet as they can be aborted. When `EndTx(TID)` is received, the associated transaction is committed, and its status is changed to `COMMITTED` (see ❷). KEVINSSD then notifies KEVINFS that the transaction is committed. Even

though some KV indices and values can still be buffered in DRAM (*i.e.,* the in-memory TxLogs and the memtable), KEVINSSD ensures persistence by using a capacitor. Finally, committed KV indices should be reflected to the permanent data structure, the LSM-tree, through a checkpoint process.

Unfortunately, the checkpoint cost is high because committed KV indices should be inserted into L_1 in the LSM-tree. Recall that some of KV indices are stored in the in-flash TxLogs and are unsorted because they are logged in their arrival order. Thus, the checkpoint process involves extra I/Os and sorting overhead. We relieve this cost by treating committed TxLogs as part of L_1 and delaying the writing of their KV indices to the tree until the compaction between L_1 and L_2 happens. When the compaction is triggered, KV indices in the TxLogs and L_1 are flushed out to L_2 together. In this way, we can skip writing KV indices to L_1. To quickly look up KV indices in the TxLogs (which are unsorted), KEVINSSD temporarily builds a small skiplist to index K2V pairs in the TxLogs. The sorted nature of the skiplist also makes it easy to apply KV indices into the tree during the compaction.

Later, once associated KV indices are checkpointed to L_2 through the compaction, the transaction status is changed into `CHECKPOINTED` (see ❸), and the associated TxLogs and the TxTable entry occupied by them are reclaimed.

Recovery. The TxTable, buffered K2V indices, and values must be materialized to the flash regularly or when a certain event happens. In our design, KEVINSSD materializes them when a sudden crash is detected. While power is being supplied by a capacitor, it flushes out buffered K2V indices to the in-flash TxLogs and buffered values to the flash. The TxTable is updated to point to the in-flash TxLogs and is then appended to the TxRecovery. Two specific flash blocks (*e.g.,* blocks #2 and #3) are reserved for the TxRecovery and are treated as a circular log. When a system reboots, KEVINSSD scans the TxRecovery, finds the up-to-date TxTable, and checks the status of each transaction. If a transaction was already committed, it means that KEVINSSD persistently wrote KV objects to the flash before. Associated K2V indices are thus pushed into the skiplist to be searchable. The `RUNNING` transactions are aborted, and associated resources are reclaimed.

KEVINSSD supports the same level of crash consistency as EXT4 with the ordered mode, but requires much smaller I/Os by avoiding double writes. Moreover, by leveraging capacitor-backed DRAM in the controller, it further reduces the overhead of flushing out KV objects and lowers the delay of marking journal commits. The DRAM-resident TxLogs is 2 MB in size in our default setup, so a large capacitor is not required.

6 Experiments

We present experimental results on KEVIN. We seek to answer the following questions: (*i*) Does KEVIN provide high performance under various workloads, in particular, metadata intensive ones? (*ii*) How much data movement between the host and the SSD can be reduced? (*iii*) Does KEVIN provide high resistance to fragmentation? (*vi*) Does KEVIN provide benefits over existing file systems based on KV stores?

6.1 Experimental Setup

All the experiments are performed on a server machine equipped with Intel's i9-10920X CPU (12 cores running at 4.6 GHz) and 16 GB DRAM. The Linux kernel v4.15.18 is used as the operating system. Our SSD platform is based on a Xilinx VCU108 [53] equipped with a custom flash card providing 2.4 GB/s and 860 MB/s throughputs for reads and writes, respectively. The total SSD capacity is set to 128 GB. The FPGA contains controller logic to manage NAND chips and to provide the PCIe interface to interact with the server.

Our SSD platform does not have a CPU and has a similar architecture to an open-channel SSD [5]: it runs the FTL software on the host system. To emulate the limited resources of an SSD controller on an x86 host, we implement FTLs in the guest Linux OS in QEMU/KVM, completely isolated from the host that runs file systems. We assign 4 cores to the guest. 128 MB DRAM (0.1% of the SSD capacity [42, 43]) is assigned to the guest, while the rest is used by the host. The interface throughput between the host and the guest is about 8 GB/s, which is similar to that of PCIe 4.0 x4. Our default setup is biased towards existing systems considering its high interface throughput. Our system setting has a limitation. LSM-tree's compaction requires high computation power, but can easily be accelerated by FPGA or ASIC [18, 58]. Owing to the lack of those accelerators on our host system, we use the Intel i9 CPU as a sorting accelerator.

We implement two FTL schemes, the page-level FTL that uses the simple L2P indexing and the proposed KEVINSSD. The page-level FTL uses 128 MB DRAM for an L2P mapping table. KEVINSSD uses 6.5 MB of DRAM for bloom filters, 2 MB for TxLogs, 1 MB for memtables, 6 MB for in-memory index, and 112.5 MB for caching popular entries. We compare KEVINFS with four kernel file systems, EXT4 with the ordered mode, XFS, BTRFS, and F2FS.

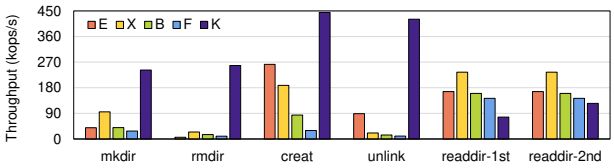
6.2 Experimental Results

We evaluate KEVIN using micro-benchmarks in §6.2.1 and carry out experiments with realistic workloads in §6.2.2. Performance analysis of aged file systems is presented in §6.2.3. The benefits of in-storage indexing are analyzed more deeply in §6.2.4. In graphs, EXT4, XFS, BTRFS, F2FS, and KEVIN are abbreviated as 'E', 'X', 'B', 'F'. and 'K', respectively.

6.2.1 Results with Micro-benchmarks

We conduct a set of experiments using three types of micro-benchmarks: metadata-only, small-file, and data-only workloads, all of which have different file/directory access patterns.

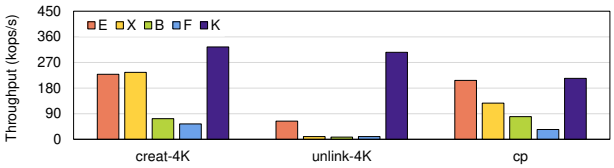
Metadata-only workloads. They create and delete a large number of empty files and directories. We use `creat`, `mkdir`, `unlink`, and `rmdir` from Filebench [48] that perform intensive updates of on-disk metadata, but do not involve any I/Os



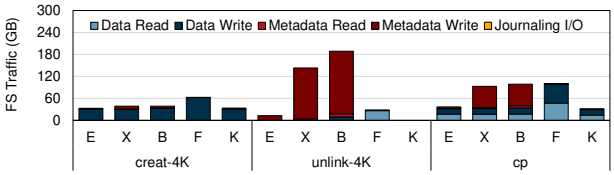
(a) Throughput

(b) File system traffic

Figure 9: Metadata intensive workloads



(a) Throughput



(b) File system traffic

Figure 11: Data & metadata workloads

on file data. A total of 8M files and directories are created and deleted. This is almost the maximum number of files and directories that can be created with EXT4's default configuration on a 128 GB disk. We also run `readdir` that iterates over a large number of directories. 4M files are stored in 800K directories. To generate sufficient I/Os, we run 16 Filebench instances in parallel. All of the file systems are initially empty.

Figure 9(a) shows results. KEVIN outperforms other systems by 6.2× on average. Aside from `readdir`, KEVIN achieves up to 43.8× better I/O throughput than EXT4. This is because KEVIN eliminates almost all of the metadata I/O traffic. Figure 9(b) depicts the amount of data moved between the host and the SSD. Compared to the existing file systems, KEVIN involves tiny data movements because it only needs to deliver small-sized KV commands.

For `readdir`, KEVIN performs poorly for the first run because it requires many reads to retrieve values (*i.e.,* inodes) from the flash pages. For its second run, KEVIN shows improved performance (from 75kops/s to 120kops/s). As explained in §4, KEVINFS rewrites a group of meta objects fetched by `ITERATE` to store them in the same flash pages, hoping that it reduces in-storage reads in the future. This optimization can increase the eviction cost slightly, but it is imperceptible to users as the I/O traffic incurred by rewrites is low and is handled in the background. Note that before the second run, we empty the inode and dentry caches to get rid of the impact of cached metadata.

KEVIN incurs internal I/Os to manage LSM-tree indices in storage, which can be categorized into three types: compaction I/Os to merge and sort K2V indices, tree lookup reads to find key-index pages, and garbage collection (GC) I/Os

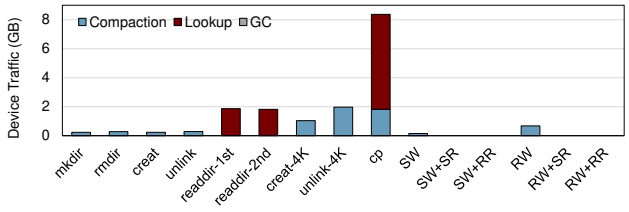
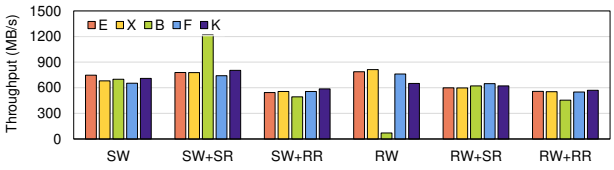


Figure 10: KEVIN I/O overheads on micro-benchmarks

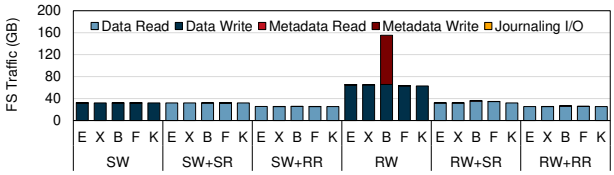
to reclaim free space. As illustrated in Figure 10, extra I/Os to manage the LSM-tree are negligible in comparison with metadata overhead in other file systems (see Figure 9(b)). Since our experiments are conducted under clean file systems, compaction and GC I/Os are almost zero. We analyze their impacts on performance in §6.2.3.

Small-file workloads. They include three scenarios: the creation (`creat-4K`) and deletion (`unlink-4K`) of 4 KB small files, as well as a copy (`cp`) of many small files. All of them create lots of data traffic on both metadata and file data. We create and delete 8M files and copy 4M files. Figure 11 shows experimental results. KEVIN exhibits the best performance when creating and removing small files thanks to its low metadata overhead. However, for `cp`, KEVINFS shows similar performance to EXT4. We observe that KEVIN shows high write throughput for small files, but the throughput of reading small files to copy is slow and becomes a bottleneck. This is owing to the relatively high tree lookup overheads. For our experiments, we run 16 Filebench instances in parallel, which cause random file reads. This eventually results in random meta object lookups on the KEVINSSD side. Moreover, as a small 4 KB file contains one subobject, KEVINSSD's compression optimizations are not effective. This is the reason why the number of reads to find key-index pages (tree lookup) is relatively high for `cp` in Figure 10. Even worse, while 4 KB file data is slightly large to be embedded in a meta object in KEVINSSD, EXT4 can directly locate a disk block where data is stored by referring to disk pointers in inodes, thereby incurring no extents lookup.

Data-only workloads. To assess how efficiently KEVIN handles a large file, we create a 32 GB file and run various I/O patterns using the FIO tool [4] on it. We first measure sequential and random write throughputs. For measuring sequential write (`SW`) throughput, we run a single FIO instance that sequentially writes 32 GB of data on a single file. For the measurement of random write (`RW`) throughput, we run 16 FIO instances that randomly write 4 KB data on a file. Over each created file (`RW` or `SW`), we run FIO instances that read data sequentially (`+SR`) or randomly (`+RR`) to measure their respective read throughputs.



(a) Throughput



(b) File system traffic

Figure 12: Data intensive workloads

Overall, KEVIN shows similar performance as the other file systems as depicted in Figure 12(a). However, for random write (`RW`), it shows slightly low throughput because of extra I/Os for compaction (see Figure 10). An interesting observation is that KEVIN exhibits excellent performance even for random read workloads (`SW+RR` and `RW+RR`). This is because KEVIN benefits from the highly compressible key format of data objects. This property enables us to cache almost all KV indices in DRAM for the 32 GB file, making it possible to achieve a sufficiently high hit ratio.

6.2.2 Results with Realistic Workloads

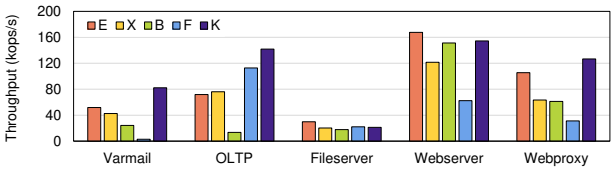
To understand the performance of KEVINFS under realistic workloads, we conduct experiments using five Filebench workloads (`Varmail`, `OLTP`, `Fileserver`, `Webserver`, and `Webproxy`), and four real applications (`TPC-C`, `clone`, `rsync`, and kernel compilation). Filebench mimics I/O behaviors of a target application that are modeled by parameters listed in Table 3. In our experiment, we use default parameters preset by Filebench, except for the number of operations.

Figures 13 and 14 show our results. For `Varmail`, KEVINFS exhibits 37% higher throughput than EXT4. `Varmail` emulates a mail server, so it performs I/Os on many small files. Metadata-intensive syscalls, `creat` and `unlink`, are frequently invoked to create and remove files. To persist user emails immediately, it calls `fsync` every time after `write`, incurring many I/Os to the journaling area.

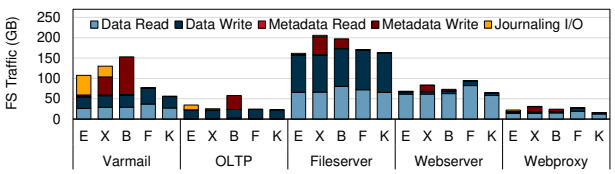
`OLTP` is a write-intensive workload in which more than 100 threads create files and append data to the files. It also frequently invokes `fdatasync`, which results in many I/Os sent to metadata and the journaling area. As a result, KEVIN exhibits 26% higher throughput than F2FS.

Table 3: Filebench parameters. C/U/R/W represents the ratio of `creat`, `unlink`, `read`, and `write` operations.

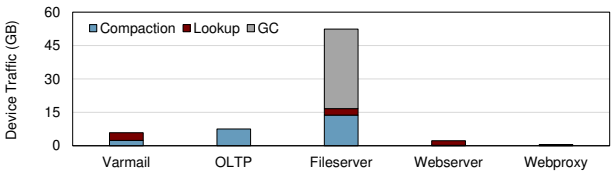
	Avg. file size	# of files	Threads	# of operations	C/U/R/W ratio
`Varmail`	16 KB	3.2 M	16	12.8 M	1:1:2:2
`OLTP`	10 MB	3.2 K	211	10 M	0:0:1:10
`Fileserver`	128 KB	800 K	50	8 M	1:1:1:2
`Webserver`	16 KB	1.6 M	100	12.8 M	0:0:10:1
`Webproxy`	16 KB	2 M	100	4 M	1:1:5:1



(a) Throughput



(b) File system traffic



(c) KEVIN I/O overheads

Figure 13: Realistic workloads from Filebench

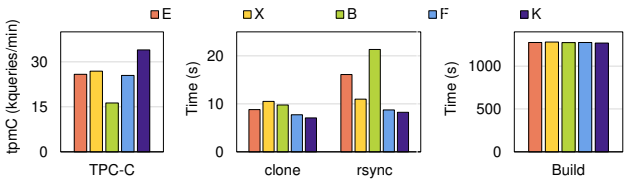


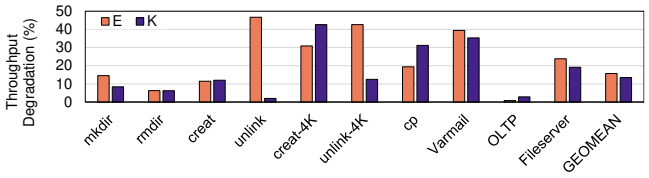
Figure 14: Application workloads

KEVINFS shows 23% lower throughput than EXT4 in `Fileserver`. `Fileserver` is a data-intensive workload that reads and writes a relatively large size of files (128 KB). It does not invoke `fsync`, so metadata updates and journaling I/Os occur only occasionally. Owing to the large amounts of data written to the disk, KEVINFS suffers from compaction overhead which slows down its performance over EXT4.

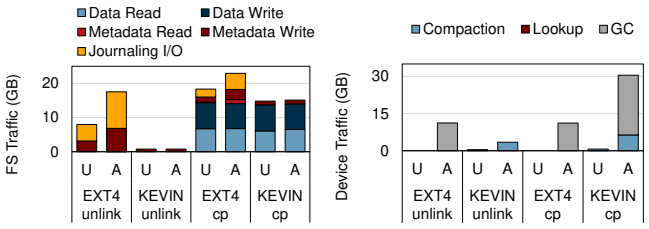
`Webserver` is a read-dominant workload issuing many reads to small files (16 KB) with few writes. Syscalls that update metadata are not invoked in `Webserver`. Although it is not preferable to KEVIN, KEVIN shows a slightly slower performance (10%) than EXT4. `Webserver` exhibits high locality in accessing files. Thanks to a high cache hit ratio (98.3%) in KEVINSSD, the number of page reads to get key-index pages is relatively small.

`Webproxy` is also a read-dominant workload, but KEVINFS exhibits higher throughput compared to EXT4. Our close examination reveals that this is owing to the high directory management overhead. `Webproxy` contains a large number of files (*e.g.,* 1M files) per directory. Whenever files are created and removed, it is necessary to update directory entries, which is costly. KEVINFS does not maintain directory entries, so its performance is not affected by directory updates.

Finally, we carry out experiments using real applica-



(a) Effect of fragmentation on performance



(b) File system traffic (c) I/O overhead of device

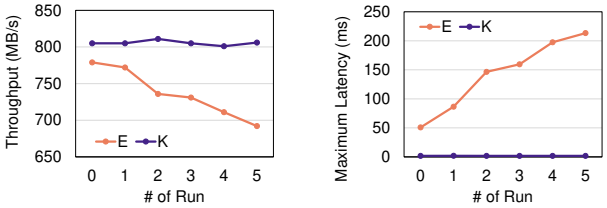
Figure 15: Write performance on aged file systems. In (b) and (c), 'U'/'A' shows unaged/aged file system performance.

tions, including `TPC-C`, `clone`, `rsync`, and kernel compilation (`build`). For `TPC-C`, MySQL is used as the DBMS engine. We create 50 data warehouses run by 100 clients. The DB size is 14 GB. The overall behavior of `TPC-C` is similar to `OLTP` in that it is write-intensive and frequently invokes `fsync`. Thus, KEVIN provides about 31% better throughput (tpmC) than EXT4. A local 3.1 GB Linux kernel repository is used as the source for both `clone` and `rsync`. They involve the creation of many small files (as `creat-4K` in §6.2.1), so KEVIN offers the best performance. A Linux kernel compilation process requires many small file reads and writes, along with directory traversals. Our results, however, reveal that the bottleneck of the kernel compilation is CPU not I/O. Consequently, all of the file systems provide similar compilation times.

6.2.3 Results under Aged File Systems

We analyze the performance of the file systems when they are aged. To age the file systems, we use Geriatrix [20] and Filebench to write more than 800 GB of data. For performance measurement, we run the same benchmarks that we use in §6.2.1. Since the file system space utilization is about 60%, we reduce the number of files and directories created by half.

Figure 15(a) shows the extent to which the file-system performance degrades after the aging process. We observe that KEVIN shows smaller performance reductions compared to EXT4 across almost all of the benchmarks. EXT4 is affected by high metadata and journaling overhead, which are exacerbated by file-system fragmentation. In the case of `unlink` in Figure 15(a), metadata and journaling I/Os increase by up to 2.2× after aging. On the other hand, there are no significant changes in file-system level I/O traffic in KEVIN. After aging, the compaction I/Os in KEVINSSD increase to 7.7×. Due in part to its very small portion in total I/Os, its negative impact on I/O performance is not huge. This confirms that KEVIN is more resistant to fragmentation. Unfortunately,



(a) FIO (b) `readdir`

Figure 16: Read performance on aged file systems

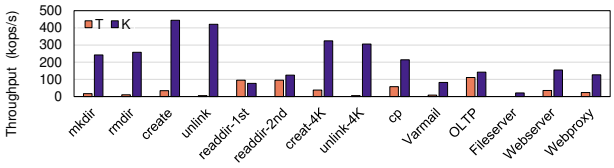
KEVIN suffers from increased compaction and GC overhead in data-intensive workloads, `creat-4K` and `cp`. Our LSM-tree indexing algorithm requires more flash space (3~10%) than the typical FTL, owing to obsolete objects staying in the tree before getting reclaimed by compaction. Thus, GC invocations occur more frequently.

To understand the impact of fragmentation on user-perceived performance, we measure read throughput and latency while varying the degree of fragmentation. The degree of fragmentation is controlled by the number of fragmentation tool runs. For each run, 128 GB data are written to the file system. We first measure sequential read throughput on a 32 GB large file (see Figure 16(a)). The read throughput of EXT4 gradually degrades as the run repeats. When the file system is clean (*i.e.,* the run 0), the file has only one extent. However, the number of extents increases to 3,798 at run 4. As explained in §4, this increases not only the tree search cost but the number of I/O requests to the disk. On the other hand, KEVIN exhibits consistent read throughput, achieving 16% higher throughput than EXT4.

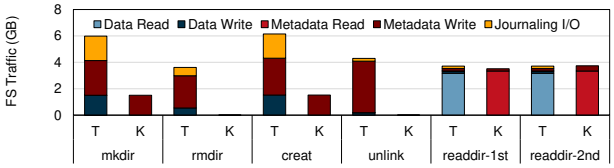
We also measure the latency of `readdir` (see Figure 16(b)). The performance of `readdir` is decided by the number of block reads to fetch inodes from the inode table. As the run repeats, the inode table is severely fragmented, and thus EXT4 involves more disk accesses to retrieve inodes. This results in an increase in latency of `readdir`. On average, KEVIN shows slower speed for `readdir` than EXT4, as in Figure 9(a). However, it is not affected by the fragmentation of the inode table and can remove data transfers to the host by fetching inodes internally. Moreover, by reading multiple meta objects at the same time through SSD's internal parallelism, it exhibits much shorter latency when the file system is aged.

6.2.4 Analyzing Effects of In-storage Indexing

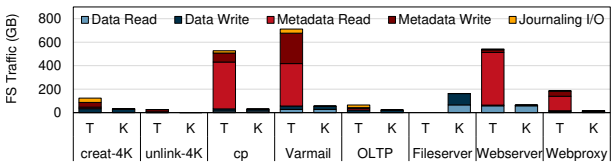
Finally, we evaluate the benefits of performing indexing operations in storage. The best way of doing this would be to move KEVINSSD's internal indexing engine to KEVINFS. Since our LSM-tree engine is currently designed to run as the flash firmware, porting this back to the kernel is a non-trivial job. As an alternative, we use TokuDB from previous KV store based file system studies [19, 56, 57]. TokuDB uses the B$^\epsilon$-tree as an internal indexing algorithm which has low computational complexity and asymptotically performs better than LSM-trees. We used the TokuDB version included in the B*e*trFS's git repository [33]. To bridge TokuDB with KEV-



(a) Throughput



(b) File system traffic for metadata workloads



(c) File system traffic for small-file and realistic workloads

Figure 17: Effects of in-storage indexing

INFS, we port the code from v3.11.10 [19] to v4.15.18 kernel. For a fair I/O traffic comparison, we turn off TokuDB's value compression feature and set the internal cache size to 4 GB. This in-kernel TokuDB operates on a block SSD formatted to EXT4 [19]. The block SSD uses a page-level FTL.

We conduct experiments with Filebench used from §6.2.1 and §6.2.2. Figure 17(a) shows I/O throughputs. In the graph, 'T' represents a setting where KEVINFS uses TokuDB as its in-kernel indexing engine in the host, while 'K' denotes the proposed KEVIN that uses the in-storage indexing engine. KEVIN shows an improvement of 7.4× on average even with `readdir` which is relatively slow.

To understand why KEVIN performs much better than KEVINFS+TokuDB, we analyze I/O traffics from the two settings, which are presented in Figures 17(b) and (c). KEVINFS+TokuDB numbers are taken from TokuDB's statistics. In KEVINFS+TokuDB, 'Data Read/Write' represents the traffic from reading and writing KV objects and 'Metadata Read/Write' is the extra indexing I/O traffic from the B^ε-tree to manage KV objects. 'Journaling I/O' includes the traffic from TokuDB's WAL logic. `Fileserver` fails to run on KEVINFS+TokuDB owing to the space overhead [19] caused by the B^ε-tree algorithm that consumes all disk space.

In the case of the write-intensive workloads, traffic differences are substantial. This is because TokuDB incurs many extra I/Os. As mentioned in §2.3, the WAL policy [13, 15, 34] has to write all KV objects to logs before materializing them to the data area. Note that this overhead can be mitigated if the late-binding journaling is used [56] which is not implemented yet in this work. In workloads such as `Varmail` that have many `fsync` calls, TokuDB has to flush the logs, increasing `fsync`'s latency. KEVIN shows 33.8× shorter latency compared to KEVINFS+TokuDB. The B^ε-tree also incurs more traffic because of its inherent behavior that transfers data from the internal node buffer to the leaf node. Operations involving many point queries such as `cp`, `Webserver`, and `Webproxy` show lower read traffic in KEVIN than KEVINFS+TokuDB. KEVIN performs indexing with the key-index page caching and compression from the storage device itself, and thus it offers fast indexing performance without any external I/Os. `readdir` shows worse performance on KEVIN. TokuDB manages all KV pairs in a sorted manner without key-value separation, and thus `ITERATE` performs quickly akin to sequential I/Os. However, when KEVIN rewrites the meta object before the second run, it shows higher performance, as it does not need to read multiple value pages. In this case, we expect the performance will be further improved if KEVIN adopts full-path indexing that globally sorts the file-system hierarchy.

7 Conclusion

In this paper, we proposed KEVIN, which improved file system performance by offloading indexing capability to the storage hardware. KEVINSSD exposed the KV interface and supported transaction commands. On top of this, we built KEVINFS, a new file system that translated VFS calls into KV objects and exploited storage capabilities to remove metadata and journaling overhead. Our results showed that, on average, KEVIN improved I/O throughput by 6.2× and reduced the I/O traffic by 74% for metadata-intensive workloads.

The idea of KEVIN can be extended in two directions. First, we focused on porting file systems over the KV device in this study. However, the proposed KV interface can be extended to support a broader range of applications, ranging from block-interface applications [8] to SQL applications [12], giving us the potential to replace the existing block I/O interface. Applications running directly over the KV device are expected to enjoy the same benefits (*e.g.,* small metadata overheads) as KEVIN. Second, KEVINFS can be implemented in the form of a user-level file system. KEVINSSD can export KV-APIs to the user-space (*e.g.,* via SPDK [54]), and KEVINFS accesses a storage device without going through the deep kernel stack. The user-level KEVINFS would be faster than existing user-level file systems because it not only has a lighter-weight architecture (*e.g.,* free from metadata management and journaling), but is also less affected by fragmentation.

Acknowledgments

We would like to thank our shepherd, Dr. Donald E. Porter, and five anonymous reviewers for all their helpful comments. This work was supported by Samsung Electronics Co., Ltd. and the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031 and NRF-2019R1A2C1090337). We thank Samsung Electronics for providing KV-SSD prototypes. (Corresponding author: Sungjin Lee)

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, 2008.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces, Crash Consistency: FSCK and Journaling*. Arpaci-Dusseau Books, 1.01 edition, 2019.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces, File System Implementation*. Arpaci-Dusseau Books, 1.01 edition, 2019.

[4] Jens Axboe. FIO: Flexible I/O Tester Synthetic Benchmark. https://github.com/axboe/fio.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 359–374, 2017.

[6] Yun-Sheng Chang and Ren-Shuo Liu. OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface. In *Proceedings of the USENIX Annual Technical Conference*, pages 1009–1024, 2019.

[7] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without Ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies*, page 9, 2012.

[8] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–953, 2019.

[9] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for next-Generation, Solid-State Drives. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 197–212, 2013.

[10] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2019.

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[12] Facebook, Inc. MyRocks: A RocksDB storage engine with MySQL. http://myrocks.io.

[13] Facebook, Inc. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org.

[14] Akira Fujita and Takashi Sato. e4defrag - Online Defragmenter for Ext4 Filesystem. https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[15] Google, Inc. LevelDB. https://github.com/google/leveldb.

[16] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *Proceedings of the USENIX Annual Technical Conference*, pages 759–771, 2017.

[17] Xiao He, Zhongxia Wang, Jingsheng Zhang, and Chengzi Ji. Research on security of hard disk firmware. In *Proceedings of International Conference on Computer Science and Network Technology*, pages 690–693, 2011.

[18] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the USENIX Annual Technical Conference*, pages 173–187, 2020.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BεtrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 301–315, 2015.

[20] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging What You See and What You Don't See. A File System Aging Approach for Modern Storage Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 691–704, 2018.

[21] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2013.

[22] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the ACM International Conference on Systems and Storage*, pages 144–154, 2019.

[23] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access File System with DevFS. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 241–256, 2018.

[24] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction Support using Compound Commands in Key-Value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, July 2019.

[25] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[26] Seung-Ho Lim, Hyun Jin Choi, and Kyu Ho Park. Journal Remap-based FTL for Journaling File System with Flash Memory. In *Proceedings of the International Conference on High Performance Computing and Communications*, pages 192–203, 2007.

[27] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 31–44, 2013.

[28] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 133–148, 2016.

[29] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 81–96, 2014.

[30] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference*, pages 221–234, 2015.

[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[32] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the USENIX Annual Technical Conference*, pages 787–798, 2017.

[33] Percona, Inc. BetrFS Repository. https://github.com/oscarlab/betrfs.

[34] Percona, Inc. Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[35] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 8–8. USENIX Association, 2005.

[36] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 147–160, USA, 2008.

[37] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can Applications Recover from fsync Failures? In *Proceedings of the USENIX Annual Technical Conference*, pages 753–767, 2020.

[38] Reiser, H. ReiserFS. http://www.namesys.com, 2004.

[39] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 145–156, 2013.

[40] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9:9:1–9:32, 2013.

[41] Samsung Electronics. Samsung Key Value SSD enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2017.

[42] Samsung Electronics. 860EVO SSD Specification. https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf, 2018.

[43] Samsung Electronics. 960PRO SSD Specification. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/, 2019.

[44] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[45] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–1, 2005.

[46] SNIA. Key Value Storage API Specification Version 1.0. https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[47] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–1, 1996.

[48] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *The USENIX Magazine*, 41, 2016.

[49] The Linux Foundation. Ext4 Filesystem documentation. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[50] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[51] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-atomic updates of application data in a linux file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 203–211, 2015.

[52] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 211–226, 2018.

[53] Xilinxm, Inc. Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html#hardware.

[54] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[55] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. RFLUSH: Rethink the Flush. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 201–210, 2018.

[56] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–14, 2016.

[57] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

[58] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 225–237, 2020.

A Artifact Appendix

Abstract

KEVIN is composed of two main elements: KEVINSSD (providing key-value interface with in-storage indexing) and KEVINFS (providing file system abstraction). The artifact is consisted of multiple Git repositories including KEVINSSD, KEVINFS and others used for evaluation for KEVIN. Please refer to the README file from https://github.com/dgist-datalab/kevin.

Scope

The artifact includes all the necessary source code required to run KEVIN as well as the benchmarks used in this paper. As it takes several weeks to run all the benchmarks used in this paper, we also provide exemplary benchmark suite with tuned parameters.

Contents

We provide four Git repositories related to KEVIN. First, KEVINFS provides abstraction of files and directories (see §3.1, §4 and §5.1). Second, KEVINSSD is the storage engine optimized for in-storage indexing using LSM-tree (see §3.2, §3.3 and §5.2). Third, BLOCKSSD is another storage engine used for comparison. Its FTL firmware uses page-level mapping and provides the block interface to the host. BLOCKSSD is used for comparison of traditional file systems in §6. Lastly, we provide the kernel source used in this paper. It is based on Linux kernel v4.15.18 and is further customized to run KEVIN+TokuDB (see §6.2.4). KEVINFS also runs on this kernel. Additionally, the DOI for the artifact includes a detailed screencast of the tool along with results with example workloads to prove the functionality of KEVIN.

Hosting

We provide the public Git URLs and commit hashes for each repository used during the artifact evaluation.

- KEVINFS

  https://github.com/dgist-datalab/kevin

  `1bd8566c580f8190364008f1a355fe337fcb6309`

- KEVINSSD

  https://github.com/dgist-datalab/KevinSSD

  `026e2a9bd274989b1324bdb9d008f2044e6d145d`

- BLOCKSSD

  https://github.com/dgist-datalab/BlockSSD

  `f944a94455f56a42ee3a888431b71d7e555b7671`

- Kernel source used with KEVIN

  https://github.com/dgist-datalab/linux/tree/kevin-4.15

  Branch: `kevin-4.15`

  `c2b106a1de494c293f33c5f130435c2eaee02dcf`

- The DOI for the artifact

  `10.5281/zenodo.4659803`

  https://zenodo.org/record/4659803

Requirements

We use the Xilinx Virtex® UltraScale™ FPGA VCU108 platform and customized NAND flash modules. The customized NAND flash modules used in this paper are not publicly or commercially available. Therefore, you may need your own NAND modules compatible with VCU108 and adequate modifications to the hardware backend (KEVINSSD and BLOCKSSD) to replicate this work.

Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes

Qing Wang, Youyou Lu*, Junru Li, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

We present Nap, a black-box approach that converts concurrent persistent memory (PM) indexes into NUMA-aware counterparts. Based on the observation that real-world workloads always feature skewed access patterns, Nap introduces a NUMA-aware layer (NAL) on the top of existing concurrent PM indexes, and steers accesses to hot items to this layer. The NAL maintains 1) *per-node partial views* in PM for serving insert/update/delete operations with failure atomicity and 2) *a global view* in DRAM for serving lookup operations. The NAL eliminates remote PM accesses to hot items without inducing extra local PM accesses. Moreover, to handle dynamic workloads, Nap adopts a fast NAL switch mechanism. We convert five state-of-the-art PM indexes using Nap. Evaluation on a four-node machine with Optane DC Persistent Memory shows that Nap can improve the throughput by up to 2.3× and 1.56× under write-intensive and read-intensive workloads, respectively.

1 Introduction

We consider the problem of making persistent memory (PM) indexes NUMA-aware. Although there has been a wealth of prior research designing high-performance PM indexes [1–16], the impacts of non-uniform memory access (NUMA) architecture to PM indexes have not been deeply explored. Due to limited DIMM slots and cores in a single CPU, NUMA architecture is a necessity for providing massive bandwidth and capacity of PM along with enormous computational power. In a NUMA machine, the CPU cores and DRAM/PM DIMMs are grouped into nodes, which connect each other via inter-node links, e.g., Intel Ultra Path Interconnect (UPI).

The NUMA problem on PM indexes is unique. First, PM suffers from more severe impacts of NUMA than DRAM. Specifically, for Intel Optane DC Persistent Memory (i.e., Optane DIMM), the first PM product, compared with local PM write, the peak bandwidth of remote ones is decreased to 59%; worse, highly concurrent remote PM writes (i.e., more than 8 threads) experience a bandwidth cliff (§2.1). Second, to guarantee failure atomicity (i.e., the system can recover to a correct state upon system crashes), a PM index should issue flush instructions for explicitly evicting data from CPU caches to PM. For data that resides on remote nodes, these flush instructions expose remote PM writes on the critical path, degrading the performance. Third, PM has limited bandwidth (1/6 and 1/3 of DRAM in terms of writes and reads, respectively [17]), making replication-based approaches impractical. Existing NUMA-aware DRAM indexes always (partially) replicate indexes across NUMA nodes and synchronize these replicas via compact operation logs [18,19]. Replication effectively reduces remote accesses; yet, since every update operation is executed at every node, *the number of local accesses is amplified significantly*. Although this amplification is not a problem for DRAM due to its extremely high local bandwidth, it is fatal for PM with low local bandwidth.

In this paper, we propose Nap (NUMA-Aware Persistent Memory Indexes), a black-box approach that converts concurrent PM indexes into NUMA-aware counterparts. Nap is based on a common observation: real-world workloads always feature skewed access patterns [20–24], where a small portion of hot items receive extremely frequent accesses. The key idea of Nap is *making hot accesses NUMA-aware*. Nap introduces a general NUMA-aware layer (NAL), which can be placed on the top of any existing concurrent PM index. The NAL absorbs accesses to *hot items*, while the underlying PM index handles accesses to other items. Specifically, NAL maintains per-node partial and crash-consistent views (PC-views) in PM, which serve insert/update/delete operations from local threads with failure atomicity. NAL does not synchronize states between PC-views, to *avoid remote PM accesses without inducing extra local PM accesses*. Such a synchronization-less approach brings two challenges: 1) serving lookup operations to hot items; 2) identifying the latest values from multiple PC-views upon recovery. For 1), NAL maintains an additional global view of hot items in DRAM.

*Jiwu Shu and Youyou Lu are the corresponding authors.
{shujw, luyouyou}@tsinghua.edu.cn

For 2), NAL adopts a version-based mechanism to order insert/update/delete operations to the same items, along with low-overhead methods of failure atomicity.

Upon workloads change, NAP can identify the new set of hot items and then switch to a new NAL quickly. The hot set identification is achieved by a combination of accurate and efficient streaming algorithms (e.g., count-min sketch [25]). To mitigate blocking of foreground index operations during NAL switch, NAP introduces a *three-phase switch*. This mechanism detects the states of access threads via a lightweight grace-period-based method. By leveraging these states, NAP divides the switch into three phases, and carefully splits tasks (e.g., initializing new NAL, flushing and recycling old NAL) into different phases. As a result, only a small portion of index operations during a small interval are blocked.

NAP approach offers several advantages. First, it is general and efficient; we convert five state-of-the-art concurrent PM indexes using NAP, and the NAP-converted counterparts boost the throughput significantly on a four-node machine. Second, since the set of hot items is always small, the extra memory consumption and recovery time induced by NAP are bounded. Our evaluation on a four-node machine running 72 threads shows that, when maintaining 100K hot items in the NAL, NAP uses less than 70MB extra DRAM/PM space, and the recovery time is less than 1 second.

NAP has some limitations. First, it targets skewed workloads but not uniform workloads, which appear relatively rarely in the real world. Second, NAP-converted PM indexes may be outperformed by a crafted NUMA-aware PM index. However, when designing and evaluating NAP, we conclude some guidelines that may benefit future specialized NUMA-aware PM indexes, among which the most remarkable is that *a NUMA-aware PM index should reduce remote PM accesses without consuming extra local PM bandwidth*.

In summary, this paper makes the following contributions:

- NAP, a black-box and practical approach that converts concurrent PM indexes into NUMA-aware counterparts.
- A set of techniques that enable NAP's fast reaction to workloads change.
- Experimental evidence showing the efficiency of NAP.

2 Background and Motivation

In this section, we firstly show that access to remote PM suffers from low performance (§2.1), and how it cripples PM indexes (§2.2). Then, we analyze why existing approaches for DRAM indexes are inefficient when applied to PM (§2.3).

2.1 NUMA Impacts on PM

PM is a new memory technology that enjoys benefits of both storage and memory: it provides byte-addressable storage with DRAM-comparable performance and high density. With the release of Optane DIMMs, the first PM product, the system

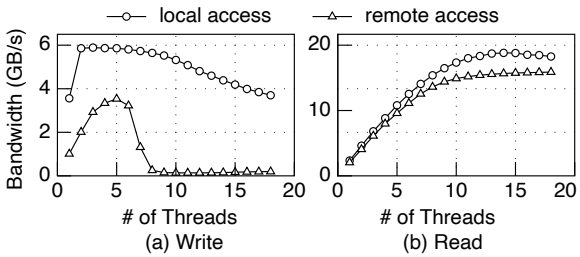


Figure 1: Bandwidth of three 128GB Optane DIMMs with varying threads. ***local access:*** *threads access Optane DIMMs that are local to them;* ***remote access:*** *threads access Optane DIMMs installed on another NUMA node. We use* `ntstore` *instructions for PM write.*

community is actively redesigning storage systems to gain full exploitation of its potential [8, 16, 26–36]. A NUMA machine with numerous CPU cores and Optane DIMMs should be an ideal architecture for fast and large-volume storage; however, this is not true, due to slow remote PM accesses (i.e., accessing PM on remote NUMA nodes).

Figure 1 reports the local/remote bandwidth of Optane DIMMs (3 Optane DIMMs and 18 CPU cores per NUMA node). Each thread performs sequential access to a 2GB PM space. We use 32-byte non-temporal stores (`ntstore`) for PM write. The peak write bandwidth of remote accesses (3.5GB/s) is only 59% of that of local accesses (5.9GB/s). Worse, the bandwidth of remote write collapses (< 250MB/s) in case of more than 8 concurrent threads. For read operations, though Optane DIMMs have a relatively smaller gap (16.9%) between local bandwidth and remote bandwidth, the extra access latency induced by inter-node links, i.e., UPI, is considerable (~100ns), exacerbating the already high PM read latency (~300ns, [17]). Based on these observations, we conclude that a high-performance PM system should avoid accessing remote PM, especially for writes.

Our experimental result is consistent with recent studies [17, 36–38]. We attribute the low performance of remote PM write to two reasons. First, `nstore` instructions may behave like cache line read-modify-write instructions, reducing the available PM bandwidth [38]. Second, due to the read-modify-write behavior, remote writes may trigger multi-socket cache coherence traffic, which induces extra PM writes [39].

2.2 NUMA Impacts on PM Indexes

By leveraging the persistence and byte-addressability of PM, PM indexes can recover instantly in the presence of power outages. Although there has been an influx of PM indexes designed for Optane DIMMs, most of them are evaluated in a single NUMA node environment [8, 11, 13, 14, 29, 42]. Here, we investigate the NUMA impacts on PM indexes by analyzing CCEH [9], a variant of extendible hashing optimized for PM. CCEH manages a set of segments, which are pointed by a global directory. As shown in Figure 2(a), when performing

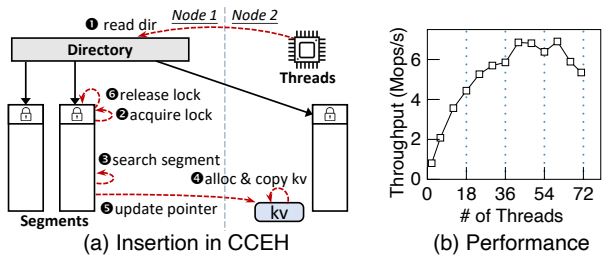


Figure 2: NUMA impacts on PM indexes, using CCEH as an example. *We use source code from [40], which relies on PMDK [41] for PM allocation and supports variable-length keys.* ***(a)*** *An insert operation. Access threads reside on node 2, while the directory and the targeted segment are on node 1. This insertion needs 2 remote reads (❶❸) and 3 remote writes (❷❺❻).* ***(b)*** *Throughput of CCEH. Each thread allocates PM space from its local node. Vertical lines show the boundaries between NUMA nodes.*

an insertion, a thread may trigger multiple times (up to 2 remote reads and 3 remote writes) of remote PM accesses. Such remote accesses can significantly degrade the performance of PM indexes. We measure the performance of CCEH under multi-node environment with a synthetic workload, where the ratio of lookup to insert/update is 1:1 and keys follow the Zipfian distribution with parameter 0.99. We use 15-byte keys and 8-byte values. Our platform is comprised of four Intel Xeon Gold 6240M CPUs (18 cores per CPU), each with three 128GB Optane DIMMs (1.5TB in total). More details of hardware configurations are shown in §6. Figure 2(b) shows the result. CCEH scales well within a single NUMA node. However, the growth rate of throughput slows down significantly when the thread number increases from 18 to 36; the main cause is remote PM accesses. When more NUMA nodes are added, i.e., thread number increases from 36 to 72, the throughput fluctuates: it increases first and then decreases. This is because that a newly added NUMA node brings extra PM bandwidth resource, boosting the throughput, but soon, slow PM remote accesses become the key performance determinant, degrading the throughput.

2.3 Limitations of DRAM-orient approaches

A natural question now arises: are existing NUMA-aware approaches for DRAM indexes still efficient when applied to PM? We give a negative answer to this question by examining Node Replication (NR) [18], a state-of-the-art approach that obtains NUMA-aware DRAM indexes. NR maintains a global shared log and per-node replicas of DRAM indexes. Using flat combining [43] within nodes, threads record their operations into the shared log, and execute the log entries to make their local replicas consistent between nodes. Three main limitations leave NR ill-suited for PM indexes.

First, obviously, NR does not consider failure atomicity,

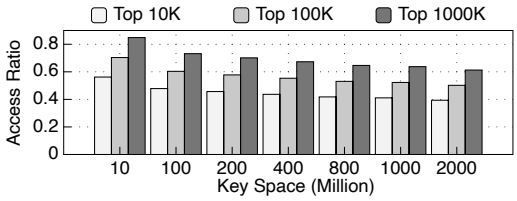


Figure 3: Access ratio of hot items (Zipfian 0.99).

which is indispensable for PM indexes. Second, NR experiences severe space overhead: for a machine with n NUMA nodes, NR consumes n times more PM due to replication. As important storage system components, PM indexes always occupy a large portion of PM space; hence, such consumption is unacceptable. Third, performance of insert/update operations is limited by PM write bandwidth of a single NUMA node. To maintain consistent replicas between nodes, each node must execute the same series of operations, which wastes precious local PM write bandwidth (only 1/6 of DRAM) and further bottlenecks the overall throughput.

3 Key Ideas

1) Making hot accesses NUMA-aware. Real-world workloads often feature Zipfian popularity distribution [20–24], where a small portion of hot items receive extremely frequent accesses. A recent study from Twitter [20] shows that their in-memory cache workloads are usually even more skewed than YCSB [44]. We design NAP to target these skewed workloads by making accesses to hot items NUMA-aware. To show potential benefits of such a design, we run a simulation to present the access ratio of hot items. The key popularity follows Zipfian distribution with parameter 0.99. From Figure 3, we observe that under a wide range of key space (from 10M to 2000M), the top 10K/100K/1000K hottest items receive more than 39%/50%/61% accesses. Hence, if we can absorb accesses to hot items (e.g., top 100K) in a NUMA-aware way, a significant percentage of remote PM accesses are avoided.

NAP introduces a NUMA-aware layer (NAL) to absorb accesses to these hot items. In addition to reducing remote PM accesses, the NAL features two advantages. First, since the set of hot items is always small (e.g., 100K), different from replication-based approaches (e.g., NR [18]), the DRAM/PM space used by the NAL is limited. Second, upon system crashes, the small-sized NAL can be recovered fast, bounding the recovery time.

2) Black-box approach. NAP exploits hotness of items to handle the NUMA problem, which enables a black-box approach for converting existing PM indexes into NUMA-aware ones. Specifically, in NAP, the NAL absorbs accesses to hot items, and an underlying PM index accommodates a large number of cold items. NAP requires no inner knowledge of the underlying PM index. Any existing PM index that is crash-consistent and thread-safe can be used; thus, NAP takes advantage of the

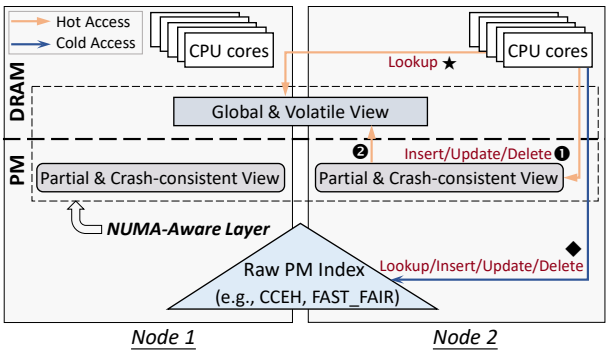


Figure 4: NAP's architecture and interactions.

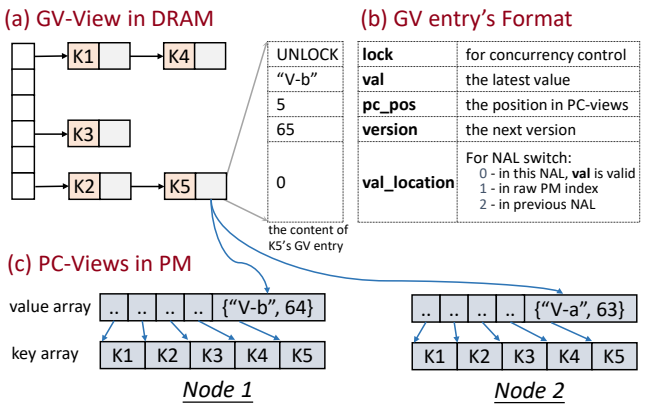


Figure 5: Structures of the GV-view and PC-views.

mature, well-tested codes of PM indexes, which are usually implemented via myriad engineering efforts.

3) Minimizing state synchronization between PM nodes. The NAL records updates to hot items into the local PM and does not synchronize PM-resident states between different NUMA nodes; thus, in addition to reducing consumption of remote PM bandwidth, no extra local PM bandwidth is consumed in NAP. To enable efficient lookup operations in such a synchronization-less approach, the NAL maintains the latest values of hot items in the DRAM.

4) Fast reaction to handle hotspot shift. Hotspots change over time, so NAP adopts several techniques to enable fast reaction. Specifically, NAP maintains the current hot items in real time. Upon detecting a new set of hot items, NAP generates a new NAL and installs it into the system in an atomic manner.

4 Design

4.1 Overview

This paper proposes NAP, an approach that converts concurrent PM indexes into NUMA-aware ones. Figure 4 presents the architecture and interactions of NAP. NAP consists of two main components: a raw PM index and a NUMA-aware layer.

- *Raw PM index*. The raw PM index can be an arbitrary existing concurrent PM index (e.g., CCEH [9], FAST_FAIR [7]), regardless of its concurrency control mechanism (lock-based or lock-free) and structure (tree-based, hashtable-based or hybrid). The raw PM index spans multiple NUMA nodes; it manages cold items (◆ in Figure 4), which account for an extremely huge proportion of the total dataset.
- *NUMA-aware layer (NAL)*. NAP steers accesses to hot items to the NAL, which contains two parts: a *global & volatile view* (i.e., GV-view, §4.2) and per-node *partial & crash-consistent views* (i.e., PC-views, §4.3). GV-view resides in DRAM, and maintains the latest values of hot items to serve lookup requests (★ in Figure 4). Per-node PC-views reside in PM. When a thread issues an insert/update/delete operation to a hot item, the PC-view in the same NUMA node absorbs the operation, and persists the operation's effect in a crash-consistent manner (❶). Then, the corresponding value in GV-view is updated (❷), to ensure the GV-view always owns the latest values of hot items. To eliminate remote PM accesses and avoid extra local PM accesses, we do not synchronize states between different PC-views, and thus each PC-view only has *partial* latest values of hot items. In case of hotspot shift, NAP can timely identify the new set of hot items (§4.4) and switch to a new version of NAL (§4.5); meanwhile, hot items in the old NAL are flushed to the underlying raw PM index.

4.2 Global & Volatile View (GV-View)

Design goals. In addition to serving lookup for hot items, the DRAM-resident GV-view is also responsible for 1) controlling concurrent accesses to the NAL, and 2) checking an item whether belongs to the hot set [1]. Thus, the design of GV-view must be lightweight and efficient.

Design details. NAP organizes the GV-view as a DRAM-resident index, which maintains the mapping from key to *GV entry* for every hot item. Figure 5(a) shows the GV-view's structure. The GV-view uses a hashtable by default; but if the raw PM index supports range query, it uses a tree-based data structure. Since the hot set is fixed unless the NAL is switched (e.g., the hot set is {*K1, K2, K3, K4, K5*} in Figure 5), the GV-view's index is constructed *entirely* during the NAL's initialization and thereafter does not make any changes to its structure. As a result, any *thread-unsafe* index with high performance is applicable (e.g., C++ `unordered_map`).

For each hot item, the associated GV entry maintains its runtime information. Figure 5(b) shows the GV entry's format, which consists of five fields: 1) a readers-writer lock to control concurrent accesses to the hot item; 2) the latest value of the item; 3) a pointer indicating where to persist the item in PC-views. 4) the version of this item, which is used for recoverability of PC-views (§4.3); 5) an enumerated value that assists in NAL switch (§4.5).

[1] To simplify exposition, we term the set of hot items as *hot set*. Here, we assume the content of hot set is known in advance (Obtaining the hot set is detailed in §4.4).

Lookup operation. In case of no NAL switch, a lookup operation is performed as the following: the access thread checks the GV-view for the targeted item; if the targeted item does not exist, the lookup is redirected to the raw PM index; otherwise, the thread acquires read lock in corresponding GV entry, copies the value, and finally releases the lock.

Range query operation. Nap complicates the range query, because items for a targeted range may exist in the GV-view and raw PM index simultaneously. An access thread performs a range query as the following: it searches the GV-view, getting the items in the targeted range (S_1); then, it obtains the S_2 by invoking the range query interface of the raw PM index; finally, it merges S_1 and S_2 (if an item exists in both S_1 and S_2, we leave the one in S_1), returning the result. Like FAST_FAIR [7] and P-Masstree [8], the range query operations in Nap are not atomic with concurrent insert/update/delete operations; if a system (e.g., database) atop Nap requires a higher isolation level (e.g., *repeatable read*), it needs to implement next-key locking or version mechanisms [45].

4.3 Partial & Crash-consistent View (PC-View)

Design goals. The per-node PM-resident PC-views absorb update/insert/delete operations and ensure the effects of these operations can survive power outages. PC-views have two design goals: 1) *Recoverability*. The states between PC-views are inconsistent, and thus Nap must be able to identify the latest values upon recovery. 2) *Low-overhead failure atomicity*. To guarantee failure atomicity, we must explicitly persist data with flush instructions (e.g., `clflush`, `clwb`, and `clflushopt`) and avoid store reordering with fence instructions (e.g., `sfence`). Minimizing the usage of these expensive instructions is key for high performance.

Design details. Nap organizes each per-node PC-view into two PM-resident arrays: a read-only *key array* and a writable *value array* (Figure 5(c)). The key array stores all the keys of the hot set. The value array reserves a *PC entry* for each hot item to record values. A hot item's PC entries are specified via the `pc_pos` field of the corresponding GV entry; for example, in Figure 5, the 5th PC entry in each PC-view belongs to `K5`. Note that each PC entry contains a pointer to the associated key in the key array, to make the NAL recoverable.

Because two threads may update the same hot item but manipulate different PC-views, values of hot items are inconsistent between PC-views. To identify the latest values upon recovery, we adopt a simple version-based mechanism. Each hot item has a monotonically increasing 64-bit version, which is recorded in the GV-view (`version` field in Figure 5). The most significant bit of a version is *deletion marker*.

Insert / Update operation. In case of no NAL switch, an insert/update operation is performed as following steps:

1) The access thread searches the GV-view for the targeted item; if the targeted item does not belong to the hot set, the operation is redirected to the raw PM index.

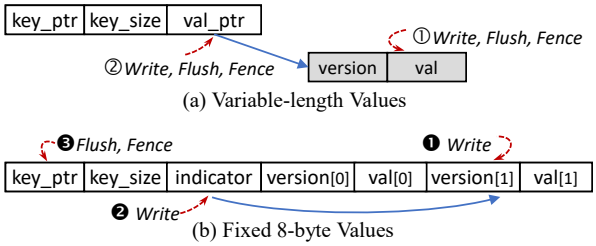


Figure 6: The structure of two types of PC entry. `key_ptr` *points to corresponding key in the key array and* `key_size` *stands for the size of the key.* ***(a)*** *For variable-length values, we use copy-on-write for failure atomicity. Each PC entry is 24-byte. The grey space of* [*version*,*val*] *is allocated from PM.* ***(b)*** *For fixed 8-byte values, we adopt a lightweight two-incarnation toggle mechanism. Each PC entry is 49-byte (*`indicator` *is 1-byte, every other field is 8-byte) and cache-line-aligned, and contains two incarnations of* ⟨*value*,*version*⟩ *pair.*

2) The thread acquires the targeted item's write lock in the GV-view, then obtains a new version.
3) The thread persists the version with the new value (i.e., ⟨*value*, *version*⟩ pair) atomically into the targeted PC entry in the local NUMA node.
4) The thread updates the volatile value in the GV-view (for future lookup operations), and finally releases the lock.

Delete operation. A delete operation has the same process as an insert/update operation, except for the above Step 3): the access thread sets the deletion marker of the obtained version and persists it into its local PC-view.

Using the version-based mechanism, we can accurately identify the latest value for a hot item from multiple PC-views: the value with maximal version (without deletion marker) is the latest; if the deletion marker of the maximal version is set, the corresponding hot item has been deleted. For example, in Figure 5(c), with the maximal version, **"`V-b`"** in the PC view of node 1 is the latest value of `K5`.

Now we describe how to guarantee failure atomicity of update to ⟨*value*,*version*⟩ pair with low overhead. Nap adopts two different mechanisms to efficiently support variable-length values and fixed 8-byte values, respectively.

For variable-length values, we leverage copy-on-write (CoW) to update ⟨*value*,*version*⟩ pair; Figure 6(a) shows the corresponding PC entry. The access thread firstly allocates free PM space and copies ⟨*value*,*version*⟩ pair to it; then, the thread flushes the pair via `clflushopt` instructions followed by a `sfence` (①); finally, the thread updates 8-byte pointer atomically to the address of ⟨*value*,*version*⟩ pair, flushes the pointer via a `clwb`, and issues `sfence` to ensure the persistence is completed (②). We use `clflushopt` (which invalidates flushed cache lines) rather than `clwb` (which does not perform cache invalidation) for ⟨*value*,*version*⟩ pair, so as to save CPU cache space for other operations; this is because that values in PC-views are only read during recovery.

NAP designs a *two-incarnation toggle mechanism* for fixed 8-byte values, which is very common in PM indexes [8] (8-byte value is usually a pointer indicating the location of real data). Figure 6(b) shows the structure of the corresponding PC entry, which is 49-byte and cache-line-aligned. There are two incarnations of 8-byte values and 8-byte versions, and an indicator pointing to the valid incarnation. When writing a new ⟨*value*, *version*⟩ pair, the access thread first copies the pair into the invalid incarnation (❶), which can be calculated according to the indicator (e.g., if the indicator points to the first incarnation, the second one is invalid). Then, the thread toggles the indicator (❷), letting it point to the updated incarnation. Finally, the thread issues a `clwb` to the PC entry followed by a `sfence` (❸). Compared to the CoW, the two-incarnation toggle mechanism saves a flush instruction and a fence instruction, enabling its efficiency. We do not need a fence before toggling the indicator, because writes to the same cache line reach PM *in program order* under TSO (total store order) architecture of Intel CPUs [7, 10, 28]. Of note, although each PC entry takes up a 64-byte PM space to enforce cache line alignment, the PM consumption is limited; this is because the hot set is small.

4.4 Hot Set Identification

Design goals. In real-world workloads, the hot set keeps changing over time [21]; thus, NAP requires to identify the hot set in real time. The design goals of identifying hot set are 1) minimizing interferences with foreground index operations, and 2) small memory footprint in the face of infinite streams of index operations.

Design details. NAP uses a dedicated *switch thread* for hot set identification, to detach this process from the critical path of index operations. Figure 7 shows how the switch thread interacts with access threads and identifies the hot set.

Each access thread maintains a circular *record buffer* to publish its access patterns. To reduce interferences caused by hot set identification, access threads use sampling and make writes to record buffers coordination-free. Specifically, every several operations (e.g., 32), an access thread writes a ⟨*timestamp*, *key*⟩ pair into the record buffer, where *timestamp* is a 64-bit number generated via `rdtsc` instructions and *key* is the key of current index operation. The access thread blindly appends ⟨*timestamp*, *key*⟩ pairs to the circular buffer, regardless of whether the overwritten data has been consumed by the switch thread (i.e., no coordination with the switch thread).

With the help of a count-min sketch [25] and a min heap, the switch thread digests record buffers in following repeated three steps.

1) The switch thread chooses a record buffer in a round-robin manner, and fetches a batch (e.g., 8) of new ⟨*timestamp*, *key*⟩ pairs from it; this batched fetch reduces cache line movements. Two types of ⟨*timestamp*, *key*⟩ pairs are considered invalid: i) the *timestamp* is less than maximal timestamp that has been read from corresponding record buffer, indicating we approach the tail of the record buffer; thus, the fetch stops. ii) (*current time* − *timestamp*) is greater than a threshold value (e.g., 100ms), indicating this pair is too stale; thus, the pair is skipped. Note that although the timestamps generated via `rdstc` are not strictly synchronized between CPU cores [46, 47], it has not caused any visible impacts for NAP.
2) For each key fetched from record buffers, the switch thread leverages a count-min sketch to update and estimate its access frequency. The count-min sketch is memory efficient, since it only uses a few small arrays. Sampling used by access threads filters out most infrequent keys, avoiding overflow of the sketch [48].
3) The min heap maintains the current hot set in the form of ⟨*key*, *frequency*⟩ pairs that are ordered by the *frequency* field. The size of the heap has an upper bound (e.g., 10,000), which can be configured. For a key fetched from record buffers (we call it `K`, and call its estimated frequency `F`), if it is already in the heap, the switch thread updates the corresponding frequency field to `F`; otherwise, the switch thread inserts the ⟨`K`, `F`⟩ pair into the heap. If the heap is full and `F` is greater than the frequency of heap root, the thread replaces the pair in the heap root with the ⟨`K`, `F`⟩. Every time the heap is modified, we need to adjust its structure to enforce its ordering property.

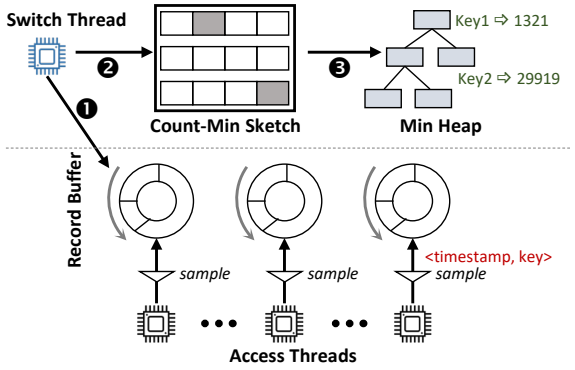


Figure 7: Hot set identification. *Access threads publish their access patterns into record buffers with sampling. The switch thread uses a count-min sketch to estimate frequency of keys and a min heap to maintain the current hot set.*

Periodically (e.g., per 1 second), the switch thread compares the heap with the hot set being used by current NAL. If there is a big difference between them, i.e., the proportion of different keys exceeds 25%, the switch thread triggers a NAL switch (§4.5) with the new hot set (i.e., keys in the heap). All statistics data, including the count-min sketch and the min heap, are cleared periodically.

Handling uniform workloads. NAP minimizes overhead induced by the NAL under uniform workloads. Specifically, the switch thread detects uniform workloads, under which it initializes an empty NAL (with 0-sized GV-view). For index

```
1  void Switch_NAL(new_hotset) {
2
3    // Phase 1, initialize the new NAL and install it.
4    NAL_new = Lazy_initialize_NAL(new_hotset, cur_NAL);
5    cur_NAL, pre_NAL = NAL_new, cur_NAL; // logging
6
7    // Phase 2, flush previous NAL into the PM index
8    Wait_for_grace_period();
9    Flush(pre_NAL);
10
11   // Phase 3, release the space used by previous NAL
12   gc_NAL, pre_NAL = pre_NAL, NULL; // logging
13   Wait_for_grace_period();
14   Collect_garbage(gc_NAL);
15   gc_NAL = NULL; // 8-byte atomic write
16 }
```

Listing 1: Switching to a new NAL. *Global pointers* `cur_NAL`, `pre_NAL` *and* `gc_NAL` *are stored in PM. Line 5 is protected via a global seqlock to ensure access threads can get a snapshot of* ⟨`cur_NAL`, `pre_NAL`⟩.

operations, access threads check the size of GV-view before searching it, which only incurs less than five CPU cycles. The switch thread can use two signals to identify uniform workloads: ① items in the heap receive less than 10% of all accesses; ② the hottest item in the heap receives comparable accesses (i.e., within 3×) to the coldest.

4.5 NAL Switch

Design goals. NAP switches to a new NAL for handling dynamic workloads. The design goals of the NAL switch lie in two aspects. First, NAP must minimize the blocking of foreground index operations during NAL switch, to avoid latency spikes. Second, the data races between the switch thread and access threads should be addressed carefully, to guarantee the consistency of the whole system.

Design details. NAP introduces a *three-phase switch*, which is fast and does not block most of foreground index operations. Its key idea is: the switch thread detects the states of access threads via a grace-period-based method (inspired by epoch-based reclamation [49]), to ensure its modifications are visible for all ongoing and future index operations.

Listing 1 shows the procedure of the NAL switch, which consists of three phases:

1) Initialize a new NAL. The switch thread initializes the new NAL according to the new hot set (line 4, NAL_{new}; we term the current NAL as NAL_{old}). Specifically, the switch thread constructs the GV-view and per-node PC-views; the PC-views are persisted for failure atomicity. For now, the GV-view of NAL_{new} only records locations of values of hot items (i.e., in raw PM index or in NAL_{old}), rather than the values themselves, by setting the `val_location` field in GV entries (Figure 5(b)). Such a *lazy initialization* is necessary for correctness: if we directly copy the latest values to the GV-view of NAL_{new}, the concurrent insert/update/delete operations to raw PM indexes or NAL_{old} will make the value in NAL_{new} stale, violating the correctness of future lookups to NAL_{new}.

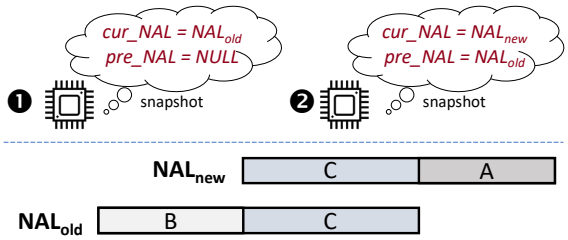


Figure 8: Different access threads see different system states. *A, B and C each stand for an exclusive set of items.*

Then, the switch thread makes NAL_{new} visible to access threads, by setting global pointers `cur_NAL` and `pre_NAL` to NAL_{new} and `cur_NAL`, respectively (line 5). To ensure that access threads always see the atomic effect of this operation, the line 5 is protected via a global seqlock [50]. Before performing an index operation, the access thread saves a snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair under the protection of the seqlock, and accesses NAL according to the snapshot. The seqlock minimizes cache coherence traffic at the reader-side (i.e., access threads).

At this time, the different ongoing index operations may have saved different snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair, as shown in Figure 8: type ❶ access threads only see the NAL_{old} and do not realize the concurrent NAL switch; type ❷ access threads see the both NAL_{new} and NAL_{old}. For type ❶ threads, they manipulate NAL_{old} and workflow of index operations is the same as cases of no NAL switch (§4.2 and §4.3). The index operations becomes a bit complicated for type ❷ threads:

i) For an insert/update/delete operation, if the targeted item belongs to NAL_{new}, NAL_{new} absorbs this operation like the case of no NAL switch (§4.3); besides, the thread copies the value into the corresponding GV entry, and updates the `val_location` field to `0` in order to indicate the value can be served for future lookups. If the targeted item falls in NAL_{old} (range `B` in Figure 8), the operation is blocked until the global pointer `pre_NAL` becomes `NULL` (i.e., phase 3 of the three-phase switch, see below); then, the operation is retried. Otherwise, the operation is redirected to the raw PM index.

ii) For a lookup operation, the thread checks GV-view of NAL_{new}, GV-view of NAL_{old}, and the raw PM index one by one. In the case that the targeted item falls in NAL_{new}, the thread checks the `val_location` field: if the value can not be served from the NAL_{new} (i.e., `val_location` is not `0`), the thread fetches the value from NAL_{old} (for range `C` in Figure 8) or the raw PM index (for range `A`) according to the `val_location` field. Range query operations experience the same workflow: access threads search NAL_{new}, NAL_{old} and the raw PM index in order, then merge results.

2) Flush NAL_{old}. In this phase, the switch thread first waits for a *grace period* to ensure all access threads become type ❷ (line 8). Our grace period mechanism is simple: each ac-

cess thread publishes its states into a slot in a global array; a slot consists of two fields: a boolean `running` and a 64-bit `cnt`. The access thread sets its `running` and increases `cnt` when starting an index operation (before saving the snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair), and resets the `running` when completing the operation. The switch thread probes the global array until every access thread is out of index operations (`running` is false) or has finished an index operation (`cnt` is changed). After this grace period, all the access threads realize the concurrent NAL switch for ongoing and future index operations, i.e., they are type ❷ threads; hence, the NAL_{old} will never be modified (recall that insert/update/delete operations to NAL_{old} are blocked for type ❷ threads). Now, the switch can flush the latest values in the GV-view of NAL_{old} to the raw PM index rapidly (via invoking interfaces of the raw PM index) without considering any data race (line 9).

3) Recycle NAL_{old}. Now, the NAL_{new} and the raw PM index reflect complete and consistent states of the system. The switch thread needs to recycle the DRAM/PM space occupied by NAL_{old}. It first saves the NAL_{old} into a global pointer `gc_NAL` and sets the `pre_NAL` to `NULL` (line 12). Then, the switch thread waits for a grace period to ensure no ongoing and future lookup operations are performed on NAL_{old} (line 13). Finally, the DRAM and PM space used by NAL_{old} is released safely (line 14), and `gc_NAL` is set to `NULL` (line 15). The access threads that realize the null `pre_NAL` are in a normal condition without any blocking; for a lookup operation to NAL_{new}, if the targeted value is not in the GV-view due to lazy initialization, the access thread fetches the value from the raw PM index, saves it to the GV-view, and updates corresponding `val_location` field to `0`.

In the above three-phase switch, the insert/update/delete operations to a part of NAL_{old} (i.e., range `B` in Figure 8) are blocked during the phase 2. Such a blocking has only a small impact on the system for two reasons. First, since the new hot set is maintained by NAL_{new}, items in the range `B` is cold, receiving a negligible percentage of accesses. Second, since the hot set is small and flushing items from NAL_{old} to the raw PM index is data-race-free, the phase 2 is fast.

Failure atomicity. The switch thread guarantees failure atomicity of three global pointers: `cur_NAL`, `pre_NAL`, and `gc_NAL`. These three pointers are allocated in PM and persisted when modified. The switch thread also maintains a small PM undo log. For line 5 and line 12 of Listing 1, the switch thread records undo log entries for atomicity. For line 15, an 8-byte atomic write is enough.

4.6 Recovery

Recovery in Nap is simple. First, we invoke the recovery procedure of the underlying raw PM index. Second, by scanning the undo log and global pointers (i.e., `cur_NAL`, `pre_NAL` and `gc_NAL`), we construct the valid version of these pointers. Third, we flush the PC-views of NALs pointed by `pre_NAL` (if not null) and `cur_NAL` in order; the latest values in PC-views of each NAL are identified by versions (§4.3). Finally, we free the PM space of PC-views in NALs pointed by these three pointers, avoiding the memory leak.

4.7 Correctness

4.7.1 Definitions

- `IL_RAW`: isolation level of the underlying raw PM index.
- `IL_NAP`: isolation level of the Nap-converted index.

4.7.2 Isolation Guarantee

Theorem 1. *For range queries, `IL_NAP` is equal to the lower level of one between `IL_RAW` and read committed.*

Proof. In Nap, a range query merges committed results from the NAL and raw PM index *without coordination*, so it is not atomic with concurrent updates. Hence, range queries reach up to read committed.

Theorem 2. *For point queries, `IL_NAP` is equal to `IL_RAW`.*

Proof. For hot items managed by NALs (i.e., NAL_{new} and NAL_{old}), Nap enforces linearizability for point queries to them. There are four cases for two conflicting operations.

- If two conflicting operations target the same NAL, readers-writer locks in the NAL serialize them.
- If a thread updates an item in NAL_{old}, future lookups [2] to NAL_{new} can see the value due to the lazy initialization.
- If a thread updates an item in NAL_{new}, it means the NAL_{new} has been installed. Hence, all future lookups will see the NAL_{new} and get the correct value.
- If two conflicting operations OP_1 and OP_2 perform updates on NAL_{old} and NAL_{new}, respectively, all future lookups will see OP_2, which means OP_1 happens before OP_2 in the linearizable history. This is legal since it is impossible that OP_1 is invoked after OP_2's response.

4.7.3 Failure Atomicity

Theorem 3. Nap*-converted indexes do not change failure atomicity semantic of raw PM indexes.*

Proof. Nap ensures that lookups after recovery can find the latest committed updates to NALs. First, in a single PC-view, Nap adopts the two-incarnation toggle mechanism and CoW for atomic persistence. Second, among multiple PC-views in a NAL, Nap stores values along with increasing versions, which are used for accurately identifying the latest values upon recovery. Third, changes to the global PM pointers `cur_NAL` and `pre_NAL` are protected by undo logging; upon recovery, we first flush the NAL pointed by `cur_NAL` and then the one pointed by `pre_NAL`, so as to ensure only the latest values appear in the raw PM index.

[2] For an operation, its *future* operations are operations that are invoked after its response.

5 Implementation

We have implemented NAP in C++ (~2000 lines of code). NAP provides a template class in the form of "`template<T> class Nap`", where `T` is a wrapper class for a concurrent PM index with specific index operation interfaces invoked by NAP. Our programming experience shows that converting a PM index using NAP needs roughly 30 lines of wrapper class codes. We use C++ `unordered_map` to organize the GV-view by default; if the underlying raw PM index supports range query, we use C++ `map`.

We leverage PMDK [41] to manage PM space. Specifically, for each NUMA node, we initialize a PMDK pool, from which NAP allocates PM space for PC-views. To reduce expensive PMDK allocation upon CoW (§4.3), we adopt a simple customized allocator. Each thread requests 1MB chunks from its local PMDK pool, and allocates PM for CoW using classic slab mechanism. The addresses of chunks are recorded in the PM, and the allocator metadata is maintained in the DRAM. Upon recovery, after flushing the PC-views into the underlying raw PM index, NAP frees these used chunks.

6 Evaluation

In this section, we use a number of microbenchmarks and applications to evaluate NAP, seeking to answer the following questions:

- How does NAP-converted PM indexes compare with original PM indexes? (§6.2)
- How does NAP perform when value size is variable? (§6.3)
- How does NAP react to dynamic workloads? (§6.4)
- How do the characteristics of workloads and NUMA configurations affect the performance of NAP? (§6.5)
- How does NAP compare with Node Replication? (§6.6)
- What are the overheads incurred when using NAP? (§6.7)
- What is the benefit of NAP to real applications? (§6.8)

6.1 Experimental Setup

The experiments are conducted on a 4-socket (NUMA node) machine. Each NUMA node is populated with an 18-core Intel Xeon Gold 6240M CPUs, three 128GB Optane DIMMs and three 32GB DDR4 DIMMs, resulting in a machine with 72 CPU cores, 1.5TB PM and 384GB DRAM. Our machine runs Ubuntu 18.04 with Linux kernel version 5.4.0.

Unless otherwise stated, for NAP, the size of the hot set is configured to 100K, and the switch thread tries to perform the NAL switch per 0.2 seconds. Each per-core record buffer is 300KB. The count-min sketch contains 3 counter arrays, each with 32-bit 850,000 counters. The sampling interval is 32.

Workloads. We leverage a YCSB-like benchmark to evaluate the performance of PM indexes. The benchmark contains five types of workloads: 1) *write-intensive*: 50% lookup and 50% update/insert, 2) *read-intensive*: 95% lookup and 5% update/insert, 3) *write-only*: 100% update/insert, 4) *read-only*: 100% lookup, and 5) *scan-intensive*: 95% range query and 5% update/insert. By default, the key space (i.e., the range of keys) is 200 million and the key popularity follows a Zipfian distribution with parameter 0.99 (the default setting in YCSB [44]). For each experiment, we first load 16 million items then perform the workloads, which contains 64 million index operations. The ratio of insert operations to update operations is about 1:3. We use 15-byte keys and 8-byte values.

6.2 Real Indexes

Using NAP, we convert five state-of-the-art PM indexes:

- *CCEH* [9]. An extendible hashtable that is structured as a set of segments pointed by a global directory. It uses readers-writer locks for concurrency control.
- *Clevel* [11]. A lock-free version of level hashing [12], which is organized as two bucket arrays.
- *P-CLHT* [8]. PM version of CLHT [51], which is a linked-list-based hashtable. It supports lock-free lookups and uses bucket-grained locks for other operations.
- *P-Masstree* [8]. PM version of Masstree [52], a trie-like concatenation of B+ tree nodes. It adopts lock-free lookups and lock-based writes.
- *FAST_FAIR* [7]. A PM B+ tree with lock-free lookups and lock-based writes.

For CCEH, Clevel, and P-CLHT, we use the source code from [40], which relies on PMDK for PM allocation and supports variable-length keys. We modify the code to make each thread allocate PM from its local PMDK pool. For CCEH, we replace the global directory lock with an in-DRAM distributed readers-writer lock [53], avoiding its scalability issues. For P-Masstree and FAST_FAIR, we use the source code from [54] and modify the code for allocation with PMDK; besides, we improve range query implementations by making them return both keys and values. Of note, we do not use our customized allocator (§5) for these indexes; this is because the customized allocator cannot provide failure atomicity for each (de)allocation operation due to its DRAM-resident metadata.

Throughput under write/read-intensive workloads. Figure 9 shows the throughput of these PM indexes under write-intensive and read-intensive workloads, and we make the following observations:

First, compared with the original indexes, NAP-converted indexes yield much better scalability under both write-intensive and read-intensive workloads. Specifically, in four-node environment (i.e., 72 threads), NAP improves the throughput by 1.26× (FAST_FAIR) to 2.3× (CCEH) for write-intensive workloads and 1.18× (P-Masstree) to 1.56× (P-CLHT) for read-intensive workloads. This is because the NAL of NAP absorbs 45~54% operations, where the per-node PC-views eliminate the remote PM writes and the GV-view eliminates the remote PM reads. Note that the global GV-view induces remote DRAM accesses; yet, remote DRAM accesses ex-

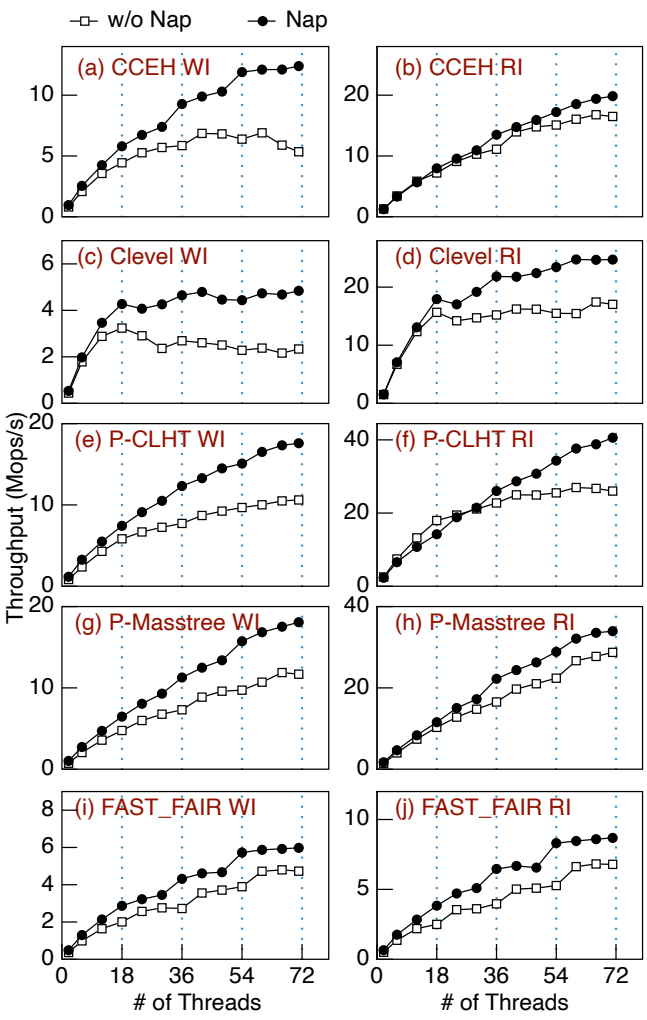


Figure 9: Throughput under write/read-intensive workloads. ***WI****: write-intensive workloads;* ***RI****: read-intensive workloads. Vertical lines show the boundaries between NUMA nodes.*

hibit much higher performance than remote PM accesses: 5.7× higher throughput for writes (20GB/s : 3.5GB/s) and 2× lower latency for reads (200ns : 400ns).

Second, even within a single NUMA node, Nap-converted indexes outperform the original ones (except P-CLHT in read-intensive workloads). This is mainly because 1) For lookup operations, the GV-view avoids the latency of PM reads. 2) For insert/update operations, the two-incarnation toggle mechanism of PC-views minimizes the overhead of PM writes. For P-CLHT, a highly optimized hashtable for cache locality, most of lookup operations are met in CPU caches under read-intensive workloads within a NUMA node, enabling its high performance. Hence, it outperforms the Nap-converted version slightly, which induces overheads of searching the GV-view for every lookup operations.

Third, compared with tree-based PM indexes, hashtable-based PM indexes are more vulnerable to NUMA architectures (particularly for Clevel, Figure 9(c) and (d)). These hashtables always use several continuous and large arrays for fast indexing (e.g., the global directory of CCEH, bucket arrays of Clevel and P-CLHT). For threads that do not reside on the same NUMA nodes with these arrays, almost all PM accesses to these arrays are remote, limiting the available PM bandwidth and further deteriorating the performance. The worst one is Clevel, because it only uses two bucket arrays for indexing; by contrast, in addition to global arrays, CCEH uses segments and P-CLHT uses linked list, which can be allocated on different NUMA nodes, increasing the available PM bandwidth of PM indexes.

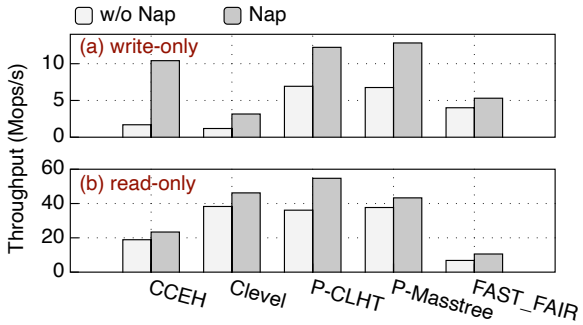


Figure 10: Throughput under write/read-only workloads. *We run 72 threads spanning 4 NUMA nodes.*

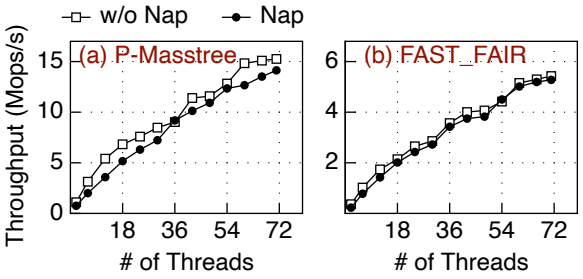


Figure 11: Throughput under scan-intensive workloads.

Throughput under write/read-only workloads. Figure 10 shows the throughput under write-only and read-only workloads. Due to space limitations, we only reports results of 72 threads. Nap boosts the throughput by 1.32× (FAST_FAIR) to 6.15× (CCEH) for write-only workloads and 1.15× (P-Masstree) to 1.55× (FAST_FAIR) for read-only workloads. Such improvement results from the NAL, which handles hot items in an efficient and NUMA-aware manner.

Throughput under scan-intensive workloads. Figure 11 shows the range query performance of P-Masstree and FAST_FAIR. We set the query range to 10. With 72 threads spanning 4 NUMA nodes, Nap reduces the throughput of P-Masstree and FAST_FAIR by 3% and 14%, respectively. This is because Nap needs to search both the GV-view and the raw PM index; yet, with the good locality of the GV-view and low latency of DRAM, the extra overhead is bounded.

Latency. Figure 12 depicts the latency distribution of P-

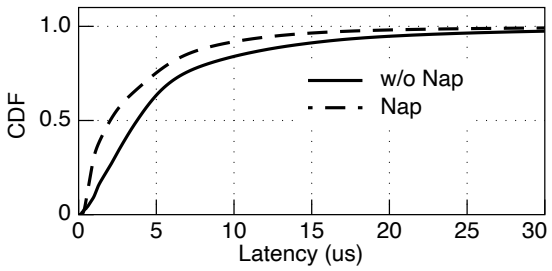


Figure 12: Latency distribution (P-CLHT, 72 threads, write-intensive workloads). *The 50th and 99th latencies of the original index are 3.77µs and 49.95µs (not shown in the figure), respectively. The 50th and 99th latencies of* NAP*-converted index are 2.04µs and 27.64µs, respectively.*

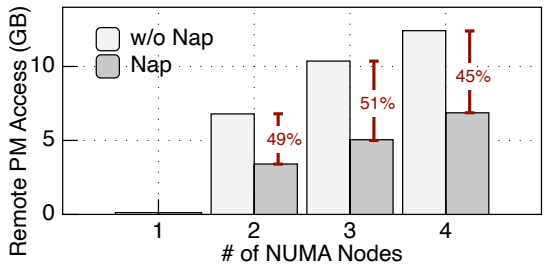


Figure 13: The amount of data via remote PM accesses (P-CLHT, write-intensive workloads). *We run 18, 36, 54, and 72 threads to measure results under different NUMA nodes.*

CLHT under write-intensive workloads. The number of access threads is 72. Due to space limitations, we omit other PM indexes that have similar results. NAP decreases the median latency by 46% (from 3.77μs to 2.04μs) and the 99th percentile latency by 45% (from 49.85μs to 27.64μs). The improvement is mainly from the per-node PC-views, which eliminate remote PM writes for hot items, reducing the possibility of multiple threads within a node access remote PM simultaneously (recall that when multiple threads write remote PM, the bandwidth collapses, affecting the access latency, Figure 1).

Quantitative measurement of remote PM accesses. We use Intel's PCM tools [55] to measure the remote PM accesses. The `pcm.x` sub-tool provides the amount of data through UPI links and the `pcm-numa.x` sub-tool monitors remote DRAM accesses. Leveraging the two sub-tools, we calculate the remote PM accesses of P-CLHT under write-intensive workloads. Figure 13 reports the result. NAP reduces remote PM accesses by 45% to 51%, enabling its high performance.

6.3 Variable-length Values

This experiment tests variable-length values, which trigger CoW in NAP. We run P-CLHT and randomly select the value size from 8 bytes to 256 bytes. Figure 14 presents the result, from which we make two observations. First, due to more flush and fence instructions in CoW, NAP's throughput

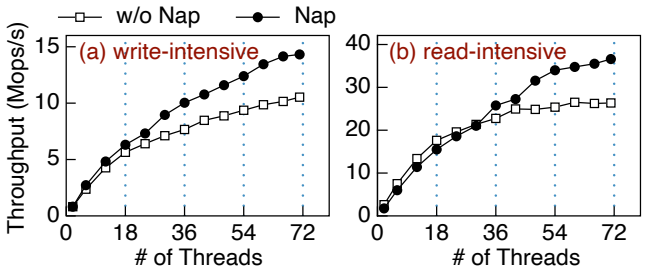


Figure 14: Throughput of P-CLHT. *The value size is randomly selected from 8 bytes to 256 bytes .*

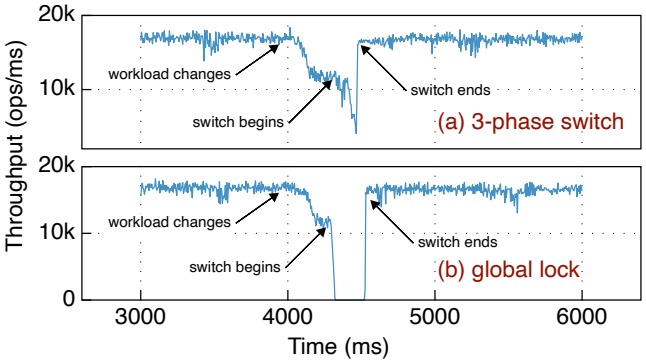


Figure 15: Throughput over time with workloads change (P-Masstree, 71 threads, write-intensive workloads).

degrades (compared with Figure 9(e) and (f)). Second, NAP-converted P-CLHT still outperforms P-CLHT by 1.36× and 1.39× under write-intensive and read-intensive workloads, respectively. This is because NAP mitigates remote PM accesses and adopts low-overhead customized allocator for CoW.

6.4 Dynamic Workloads

In this experiment, we evaluate NAP's ability to react to dynamic workloads by changing the popularity of keys. We compare our three-phase switch mechanism with a conservative mechanism that uses a global readers-writer lock: the switch is protected by the write lock, and every index operation is protected by the read lock. To avoid the cache thrashing among access threads caused by the centralized global lock, we apply per-core reader indicator [53]. We run NAP-converted P-Masstree under write-intensive workloads with 71 threads (one core is reserved to record total throughput per 5ms). Figure 15 shows the throughput over time. The workload changes at time 4s. Since the NAL can not absorb the accesses to current hot set, the throughput drops. After about 200~300ms, NAP identifies the new hot set (recall that the switch period is 0.2s, §6.1), and triggers the NAL switch. In our three-phase switch, the throughput can be maintained more than 10K ops/ms for about 130ms, then drops to 4K~8K ops/ms for about 35ms. This is because the three-phase switch only blocks some insert/update operations to a part of old NAL

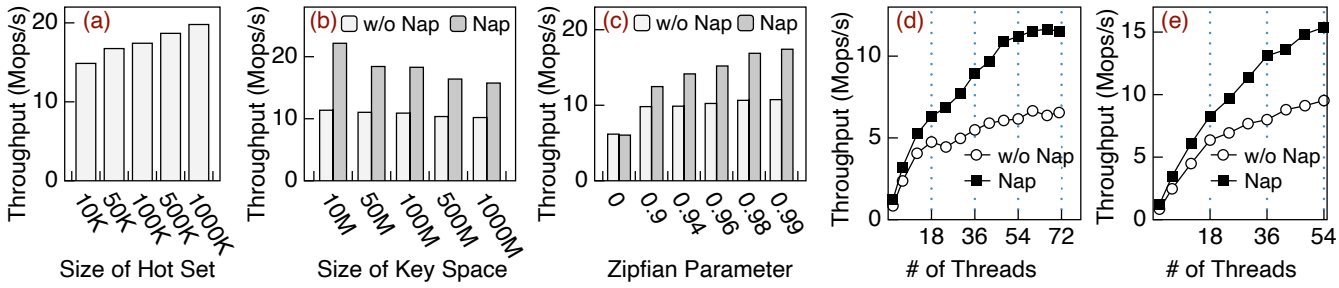


Figure 16: Sensitivity Analysis (P-CLTH, write-intensive). ***(a)*** *Varying the size of hot set (72 threads).* ***(b)*** *Varying the size of key space (72 threads).* ***(c)*** *Varying the Zipfian parameter (72 threads).* ***(d)*** *two Optane DIMMs per NUMA node.* ***(e)*** *four Optane DIMMs per NUMA node.*

during phase 2. However, when using the global lock, the system is unavailable (i.e., throughput is 0) for about 195ms. To sum up, NAP is robust enough to react to dynamic workloads quickly without sacrificing availability.

6.5 Sensitivity Analysis

Size of hot set. Figure 16(a) shows how the configured hot set size affects the NAP's performance. As the size of hot set increases from 10K to 1M, the throughput grows by 1.33×, and the percentage of operations absorbed by the NAL increases from 43% to 63%. Yet, using a large hot set consumes more PM/DRAM space and prolongs the time of NAL switch and system recovery.

Size of key space. Figure 16(b) presents the throughput of P-CLHT and its NAP-converted version with varying key space. As the key space increases, the number of hot items increases, degrading the throughput of NAP which maintains a fixed-size hot set. Even for a very large key space, i.e., 1000 million, NAP can boost the throughput by 1.55×, which demonstrates that NAP can handle large-scale workloads.

Skewness of workloads. Figure 16(c) shows how the skewness of workloads affects NAP's performance. We make three observations. First, with increasing skewness, NAP's improvement over original indexes grows. This is because the NAL can absorb more index operations. For the medium skewness case (i.e., 0.9 Zipfian parameter), NAP boosts the throughput by 1.27×. Second, under uniform workloads (i.e., 0 Zipfian parameter), throughput of both indexes drops, since there are more insert operations in uniform workloads, leading the P-CLHT to resize frequently. Third, the throughput of both indexes is almost the same under uniform workloads. This is because NAP handles uniform workloads by initializing an empty NAL, which induces negligible overhead.

Different NUMA configurations. Here, we change NUMA configurations by adding/removing Optane DIMMs, and show how the available PM bandwidth affects NAP. We get two new NUMA configurations: i) 2 Optane DIMMs per node; ii) 4 Optane DIMMs per node (only 3 nodes due to the total of 12 DIMMs). Figure 16(d) and (e) show the results of i) and ii), respectively. With 2 Optane DIMMs per node, the available PM bandwidth drops and remote PM access suffers lower write bandwidth, degrading the throughput of PM indexes; yet, under this configuration, by mitigating remote PM accesses, NAP boosts the throughput of the original index by 1.76×, which is higher than improvement under default 3-DIMMs-per-node configuration (1.66×, Figure 9(e)). Under 4-DIMMs-per-node configuration, NAP outperforms the original index by 1.62×. Overall, NAP is efficient under different NUMA configurations.

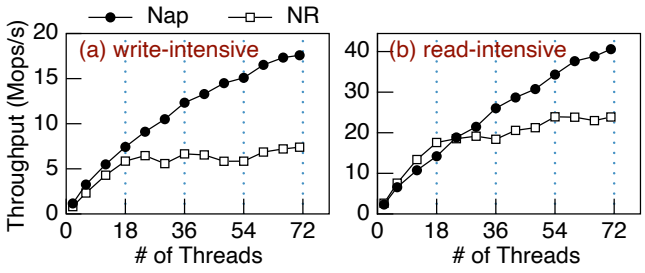


Figure 17: Performance of NAP and NR (P-CLHT).

6.6 Comparison with NR

We compare NAP with Node Replication (NR) [18], to present some key insights of designing NUMA-aware PM indexes. We put the shared log of NR in the DRAM and disable log recycle. Figure 17 shows the throughput of NUMA-aware P-CLHT converted by NAP and NR. Note that NR-converted P-CLHT is not crash-consistent: upon crash, the shared log is lost and P-CLHT on different NUMA nodes may be inconsistent. In case of 72 threads, NAP outperforms NR by 2.34× and 1.69× under write-intensive and read-intensive workloads, respectively. The inefficiency of NR on PM indexes stems from two reasons. First, by maintaining consistent replicas between NUMA nodes, each insert/update operation consumes n times more PM bandwidth (n is the number of NUMA nodes), limiting the throughput. Second, NR leverages flat combin-

DRAM				PM
record buffers	count-min sketch	min heap	GV-view	PC-views
21.1MB	9.7MB	4.2MB	3.6MB	30.1MB
Altogether, **38.6MB** DRAM and **30.1MB** PM				

Table 1: Consumption of DRAM and PM in Nap. *We ignore some very small usage, such as the 64-byte persistent undo log used by the switch thread.*

Index Type	CCEH	Clevel	P-CLHT	P-Masstree	FAST_FAIR
Time (ms)	477	522	432	306	963

Table 2: Recovery time.

ing [43] (a technique that uses a combiner to execute a batch of collected updates) to handle updates within a node. Flat combining can mitigate cache thrashing but restrict concurrency to a single thread; yet, the single-thread performance of PM indexes is much lower than that of DRAM indexes, due to expensive flush/fence instructions and high PM read latency. Combining previous experimental results (§6.2), we can conclude that the most important performance determinant of NUMA-aware PM indexes is precious PM bandwidth of both local and remote accesses (rather than cache thrashing); thus, like Nap, a NUMA-aware PM index should reduce remote PM accesses without consuming extra local PM bandwidth.

6.7 Overheads of Nap

The overheads of Nap lie in two aspects: memory consumption and recovery time.

Memory consumption. Table 1 shows the memory consumption by Nap in our evaluation (4 NUMA nodes and 72 threads), and the total memory consumption is less than 70MB. Specifically, since our NAL only maintains the hot set, the size of the min heap, GV-view and PC-views are limited. Besides, by using sampling, the small-sized count-min sketch and per-core record buffers are enough.

Recovery time. Table 2 reports the recovery time of Nap-converted PM indexes. Due to the limited size of NAL, the recovery time is bounded, which is less than one second.

6.8 Real Application

To show the benefits that a Nap-converted PM index can bring to real applications, we build a networked PM-based key-value store. The key-value store uses eRPC [56] for network communication, P-CLHT for indexing and PMDK for allocation of key-value pairs. Such a key-value store can be used for in-memory caching to reduce the total cost of ownership (comparing with DRAM-based memcached) and alleviate the impact of failures [28].

In this experiment, we use our four-node machine as the server and the other 5 machines as clients. Each machine is equipped with a Mellanox ConnectX-6 NIC (200Gbps); due to the limited bandwidth of PCIe 3.0×16, the available bandwidth of the NIC is about 13GB/s. The key-value size follows the Facebook ETC pool [23, 57]. The key popularity follows a Zipfian distribution with parameter 0.99. We consider a write-intensive workload (50% PUT). Figure 18 shows the throughput with varying clients threads. By using Nap, the throughput is improved by 1.1× under low loads (i.e., 30 client threads) and 1.49× under high loads (i.e., 180 client threads), demonstrating practical benefits of Nap.

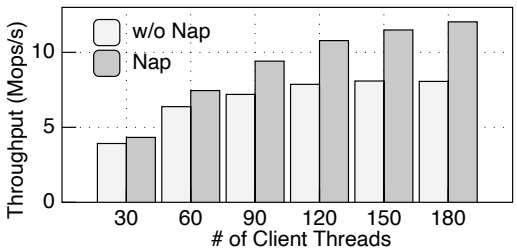


Figure 18: Throughput of a networked PM-based key-value store (write-intensive, Zipfian 0.99, 72 threads on the server). *Key-value size follows Facebook ETC workloads.*

7 Discussion

Generality of the Nap approach. Even if microarchitectures of hardware (e.g., CPU) evolve and remote PM write can deliver high bandwidth, Nap is still capable of boosting PM indexes under multi-node servers for two reasons. First, since Nap reduces remote accesses significantly, highly concurrent accesses to the same NUMA nodes can be avoided, mitigating contention in the same memory controllers and Optane DIMM XPBuffers; it is well known that such contention degrades the PM performance severely [17, 39]. Second, Nap lowers latency of index operations: for lookup operations, the GV-view eliminates remote PM reads (400ns) by using less expensive remote DRAM reads (200ns); for other operations, per-node PC-views replace remote PM writes with local ones.

Alternative designs. We discuss alternative designs to NUMA-aware PM indexes, and why we do not adopt them.

1) *Use per-core logs.* In this solution, each thread logs its updates into its local PM node and builds a global DRAM-resident index for lookups. This solution has three issues. First, considering the high bandwidth of PM, using a dedicated core for log recycle is insufficient to digest fast-growing logs; thus, we must use foreground threads or multiple dedicated cores to do this task, which has negative impact on CPU usages and performance. Second, to recycle logs, we must flush items (include hot items) into the underlying PM index, inducing remote accesses. Third, the global DRAM-resident index consumes large DRAM space.

2) *Abandon NAL switch and maintain per-node PM caches*

as PC-views. This solution adopts the architecture of NAP but abandons NAL switch. Instead, it keeps the hot set in per-node PM caches and evicts cold items at the runtime. This solution comes with three drawbacks. First, designing an ideal replacement method is difficult: if we maintain a global hotness-list for cache replacement, the multicore scalability issue happens; if we maintain a hotness-list for each set (set-associative cache), a hot item may be evicted, inducing unnecessary remote accesses. Second, when evicting a cold item from a PM cache (very common events), we must enforce failure atomicity of the cache, yielding extra performance overhead. Third, to guarantee correct lookups and recovery, all items in every PM cache should be presented in the GV-view, which complicates the execution logic. For example, when removing an item from the GV-view, we need to clear corresponding items in all PM caches.

Takeaways. We present our main takeaways from this work. 1) *A fast NUMA-aware PM index must reduce remote PM accesses without consuming extra local PM bandwidth*. The limited PM bandwidth adds a new dimension to the NUMA problem, which frustrates traditional replication-based approaches designed for DRAM indexes.

2) *We conjecture that we cannot design a NUMA-aware PM index that is optimal in ❶ minimizing remote PM accesses, ❷ not inducing extra local PM accesses and ❸ constant DRAM/PM consumption.* NAP achieves a sweet spot by leveraging the characteristics of common skewed workloads: it meets ❷ and ❸, and partially meets ❶ (the remote PM accesses to cold items cannot be reduced).

8 Related Work

PM indexes. A large body of work exists for PM indexes with the ultimate goal of minimizing overheads of failure atomicity and improving concurrency [1–16, 29, 58]. Among them, RECIPE [8], Pronto [29] and TIPS [58] propose general conversion methods. Specifically, RECIPE can convert concurrent DRAM indexes that meet a set of conditions into PM indexes; Pronto persists DRAM data structures via asynchronous semantic logging; TIPS can convert any concurrent DRAM index into PM index with durable linearizability guarantee. To the best of our knowledge, NAP is the first work that addresses NUMA problems of PM indexes.

NUMA problems on PM. Several recent studies observe pronounced NUMA impacts on Optane DIMMs [17, 37, 38]. Xu et al. [59] provide NUMA-aware interfaces to NOVA file system [60], which can set the preferred NUMA node for a file. Wang et al. [61] alleviate the NUMA issues of PM file systems by thread migration. Assise [27], a distributed PM file system, uses on-die DMA engines for remote PM writes, to bypass hardware cache coherence. These approaches for file systems cannot be easily applied to PM indexes, because PM indexes 1) use a set of fixed interfaces, 2) are shared by numerous threads, and 3) generate lots of small-sized writes.

NUMA-aware systems. There has been also work migrating NUMA impacts for DRAM indexes, locks, operating systems, and IO devices. NR [18] replicates data structures and synchronizes replicas between NUMA nodes by a shared log. NrOS [62] improves NR's scalability by allowing multiple shared logs and multiple per-node combiners. HydraList [19] and NUMASK [63] are crafted DRAM indexes that replicate index search layer (exclude index data) across NUMA nodes; compared with NR, these two indexes reduce memory consumption, but increase remote memory accesses due to shared index data. Lots of NUMA-aware locks are proposed [64–68], and most of them feature a hierarchical structure and try to keep the lock ownership within the same node. Linux automatically migrates data pages across NUMA nodes to reduce remote data access [69]. Besides, Carrefour [70] supports page replication, which can alleviate traffic hotspots and eliminate remote accesses. Further, Mitosis [71] transparently replicates and migrates page-tables across NUMA nodes to accelerate page-table walks. IOctopus [72] addresses the NUMA effects on IO devices by unifying PCIe functions to a logic one. Different from the above systems, the NUMA-aware PM indexes are unique for the limited PM bandwidth and requirements of failure atomicity.

Hotness-aware systems. Hotspots can be seen everywhere in the real world. There are two lines of work: 1) mitigating the effects of hotspots, and 2) leveraging hotspots to boost system performance. In the aspect of the former, lots of systems mitigate the load imbalance across back-end servers by using high-performance caches to handle lookup operations to hot items [48, 73–75]. In the aspect of the latter, HotRing [21] designs an in-memory hashtable that can move pointers to make hot items be served with fewer memory accesses. Like HotRing, NAP regards hotspots as an opportunity to boost system performance, but targets NUMA-aware PM indexes.

9 Conclusion

In this work, we have designed, implemented, and evaluated NAP, a black-box approach that converts concurrent PM indexes into NUMA-aware counterparts. NAP uses a NUMA-aware layer to absorb accesses to hot items, which eliminates remote PM accesses without inducing extra local PM accesses. NAP significantly boosts the performance of PM indexes on multi-node machines.

Acknowledgements

We sincerely thank our shepherd Changwoo Min for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300), and Huawei (Grant No. YBN2019125112).

References

[1] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[2] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Shimin Chen and Qin Jin. Persistent B$^+$-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[4] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 167–181, USA, 2015. USENIX Association.

[5] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 257–270, USA, 2017. USENIX Association.

[6] Faisal Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, Dhruva R. Chakrabarti, and M. Scott. Dalí: A Periodically Persistent Hash Map. In *DISC*, 2017.

[7] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[8] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[10] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 441–454, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[12] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.

[13] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.*, 13(4):421–434, December 2019.

[14] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.*, 13(7):1078–1090, March 2020.

[15] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[16] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16. USENIX Association, February 2021.

[17] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[18] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the*

Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Ajit Mathew and Changwoo Min. HydraList: A Scalable in-Memory Index Using Asynchronous Updates and Partial Replication. *Proc. VLDB Endow.*, 13(9):1332–1345, May 2020.

[20] Juncheng Yang, Yao 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, November 2020.

[21] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[22] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. 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, Santa Clara, CA, February 2020. USENIX Association.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, page 1–7, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms*, 55(1):58–75, April 2005.

[26] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.

[27] 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, November 2020.

[28] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1029–1046. USENIX Association, November 2020.

[29] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[32] 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.

[33] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), October 2020.

[34] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with

timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95. USENIX Association, February 2021.

[37] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304–315, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China*, SIGMOD '21. ACM, 2021.

[39] Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, 2021.

[40] PMDK Implementation of Clevel, CCEH and P-CLHT. https://github.com/chenzhangyu/Clevel-Hashing/, 2020.

[41] Persistent Memory Development Kit. https://pmem.io/pmdk/, 2020.

[42] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.

[43] Danny Hendler, Itai Incze, Nir Shavit, and Moran 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 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[44] 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.

[45] 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, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[46] 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, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[48] 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*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, UK, 2004.

[50] Sequential Locks. https://www.kernel.org/doc/html/latest/locking/seqlock.html, 2020.

[51] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[52] 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*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Distributed Reader-Writer Mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2020.

[54] Implementation of P-Masstree and FAST_FAIR. https://github.com/utsaslab/RECIPE/, 2020.

[55] Processor Counter Monitor (PCM). https://github.com/opcm/pcm, 2020.

[56] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[57] Diego Didona and Willy Zwaenepoel. Size-Aware Sharding for Improving Tail Latencies in in-Memory Key-Value Stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 79–93, USA, 2019. USENIX Association.

[58] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *Proceedings of the 2021 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '21, USA, 2021. USENIX Association.

[59] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[60] 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.

[61] Ying Wang, Dejun Jiang, and Jin Xiong. NUMA-Aware Thread Migration for High Performance NVMM File Systems. In *36th International Conference on Massive Storage Systems and Technology*, MSST '20, 2020.

[62] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021.

[63] Henry Daly, A. Hassan, M. Spear, and R. Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In *DISC*, 2018.

[64] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 241–252, 2003.

[65] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[66] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 215–226, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-Aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 603–615, USA, 2017. USENIX Association.

[68] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[69] AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/, 2020.

[70] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[71] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. IOctopus: Outsmarting Nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 101–115, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[74] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 31–44, USA, 2016. USENIX Association.

[75] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 143–157, USA, 2019. USENIX Association.

Rearchitecting Linux Storage Stack for μs Latency and High Throughput

Jaehyun Hwang
Cornell University

Midhul Vuppalapati
Cornell University

Simon Peter
UT Austin

Rachit Agarwal
Cornell University

Abstract

This paper demonstrates that it is possible to achieve μs-scale latency using Linux kernel storage stack, even when tens of latency-sensitive applications compete for host resources with throughput-bound applications that perform read/write operations at throughput close to hardware capacity. Furthermore, such performance can be achieved without any modification in applications, network hardware, kernel CPU schedulers and/or kernel network stack.

We demonstrate the above using design, implementation and evaluation of `blk-switch`, a new Linux kernel storage stack architecture. The key insight in `blk-switch` is that Linux's multi-queue storage design, along with multi-queue network and storage hardware, makes the storage stack conceptually similar to a network switch. `blk-switch` uses this insight to adapt techniques from the computer networking literature (*e.g.*, multiple egress queues, prioritized processing of individual requests, load balancing, and switch scheduling) to the Linux kernel storage stack.

`blk-switch` evaluation over a variety of scenarios shows that it consistently achieves μs-scale average and tail latency (at both 99$^{\text{th}}$ and 99.9$^{\text{th}}$ percentiles), while allowing applications to near-perfectly utilize the hardware capacity.

1 Introduction

There is a widespread belief in the community that it is not possible to achieve μs-scale tail latency when using the Linux kernel stack. A frequently cited argument is that, due to its high CPU overheads, Linux is struggling to keep up with recent $10-100\times$ performance improvements in storage and network hardware [17, 38]; the largely stagnant server CPU capacity further adds to this argument. In addition, many in the community argue that the resource multiplexing principle is so firmly entrenched in Linux that its performance stumbles in the common case of multi-tenant deployments [22, 38, 42]—when latency-sensitive L-apps compete for host compute and network resources with throughput-bound T-apps, Linux fails to provide μs-scale tail latencies. These arguments reflect a broad belief that, despite Linux's great success, it has emerged as the core bottleneck for modern applications and hardware.

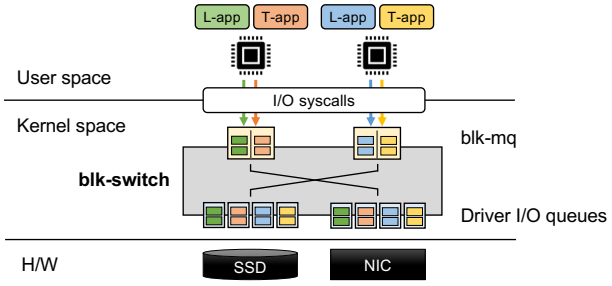


Figure 1: **The key insight in `blk-switch` design: Linux's per-core block layer design, along with modern multi-queue storage and network hardware, makes the storage stack conceptually similar to a network switch.**

This paper focuses on storage stacks used by applications to access data on local and/or remote servers. We show that it is possible to achieve μs-scale tail latency using Linux, even when applications perform read/write operations at throughput close to hardware capacity. Furthermore, low latency and high throughput can be simultaneously maintained even when tens of L-apps and T-apps compete for host resources at each of the compute, storage and network layers of the kernel stack. Finally, such performance can be achieved without any modifications in applications, network and storage hardware, kernel CPU schedulers and/or kernel network stack; all that is needed is to rearchitect the Linux storage stack.

`blk-switch` is a new Linux storage stack architecture for μs-scale applications. The key insight in `blk-switch` is that Linux's per-core block layer queues [19, 27], combined with modern multi-queue storage and network hardware [8], makes the storage stack conceptually similar to network switches (Figure 1). Building upon this insight, `blk-switch` adapts classical techniques from the computer networking literature (*e.g.*, multiple egress queues, prioritized processing of individual requests, load balancing along multiple network connections, and switch scheduling) to the Linux storage stack.

To realize the above insight, blk-switch introduces a per-core, multi-egress queue block layer architecture for the Linux storage stack (Figure 1). Applications use standard Linux APIs to specify their performance goals, and to submit read/write requests (§3). Underneath, for each application class, blk-switch creates an "egress" queue on a per-core basis that is mapped to a unique queue of the underlying device driver (that is, storage driver for local storage access, or remote storage driver for remote storage access). Such a multi-egress queue design allows blk-switch to decouple ingress (application-side block layer) queues from egress (device-side driver) queues since requests submitted at an ingress queue on a core can now be processed at an egress queue at any of the cores. blk-switch merely acts like a "switch"—at each individual core, blk-switch steers requests submitted at the ingress queue of that core to one of the egress queues, based on application performance goals and load across cores.

blk-switch integrates three ideas within such a switched architecture to simultaneously enable μs-scale tail latency for L-apps and high throughput for T-apps. First, blk-switch maps requests from L-apps to the egress queue on that core, and processes the outstanding requests in a prioritized order; that is, at each individual core, requests in L-app egress queues are processed before requests in T-app egress queues. This ensures that L-apps observe minimal latency inflation due to head-of-line blocking from T-app requests. However, strict prioritization at each core can lead to starvation of T-apps due to transient load (bursts of requests from an L-app on the same core) or due to persistent load (multiple contending L-apps on the same core). To avoid starvation during transient loads, blk-switch exploits the insight that decoupling the application-side queues from device-side queues, and interconnecting them via a switched architecture enables efficient realization of different load balancing strategies, even at the granularity on individual application requests. blk-switch thus uses request steering to load balance requests from T-apps across corresponding egress queues at all available cores. To avoid starvation due to persistent loads, blk-switch uses application steering, that steers application threads across available core at coarse-grained timescales with the goal of minimizing persistent contention between L-apps and T-apps. At its very core, the two steering mechanisms in blk-switch highlight the conceptual idea that load balancing within the Linux storage stack can be applied at two levels of abstraction: individual requests and individual threads; and, both of these are necessary to simultaneously achieve μs-scale latency for L-apps and high throughput for T-apps—the former enables efficient handling of transient loads, and the latter enables efficient handling of persistent loads on individual cores.

We have implemented blk-switch within the Linux storage stack. Our implementation is available at: https://github.com/resource-disaggregation/blk-switch. We evaluate blk-switch over a wide variety of settings and workloads, including in-memory and on-disk storage, single-threaded and multi-threaded applications, varying load induced by L-apps and T-apps, varying read/write ratios, varying number of cores, and with RocksDB [9], a widely-deployed storage system. Across all evaluated scenarios (except for sensitivity analysis against number of cores and T-app load), we find that blk-switch achieves μs-scale average and tail latency (at both 99th and 99.9th percentiles, or P99 and P99.9, respectively), while allowing applications to nearly saturate the 100Gbps link capacity, even when tens of applications contend for host resources. In comparison to Linux, blk-switch improves the average and the P99 latency by as much as 130$\times$ and 24$\times$, respectively, while maintaining $84-100\%$ of Linux's throughput. We also compare blk-switch to SPDK, a widely-deployed state-of-the-art userspace storage stack. We find that SPDK achieves good tail latency and high throughput when each application runs on a dedicated core; in the more realistic scenario of applications sharing server cores, in comparison to SPDK, blk-switch improves the average and P99 tail latency by as much as 12$\times$ and 18$\times$, respectively, while achieving comparable or higher throughput; as we will discuss, this is because polling-based userspace stacks like SPDK do not interpolate very well with Linux kernel CPU schedulers. When compared to both Linux and SPDK, blk-switch achieves similar or higher improvements for P99.9 tail latency. All these benefits of blk-switch can be achieved without any modifications in the applications, Linux CPU scheduler (blk-switch uses the default CFS scheduler), Linux network stack (blk-switch uses Linux kernel TCP/IP stack), and/or network hardware.

2 Understanding Existing Storage Stacks

This section presents a deep dive into the performance of two state-of-the-art storage stacks—Linux (including remote storage stack [29]) and SPDK (a widely-deployed userspace stack). We first describe our setup (§2.1), and then discuss several results and insights (§2.2). Our key findings are:

- Despite significant efforts in improvement of Linux storage stack performance (per-core queues [19], per-core storage and network processing [29], etc.), existing Linux-based solutions suffer from high tail latencies due to head-of-line blocking, especially in increasingly common multi-tenant deployments [31, 52], that is, when L-apps compete for host resources with T-apps that perform high-throughput reads/writes to remote storage servers. Intuitively, such scenarios result in a complex interference at three layers of the stack—compute, storage, and network—requiring careful orchestration of host resources to achieve μs-scale tail latency, while sustaining throughput close to hardware capacity. Existing Linux-based solutions fail to efficiently achieve such orchestration. For instance, even with one L-app competing with one T-app, we observe tail latency inflation of as much as 7$\times$ (when compared to isolated case, where the L-app runs on a dedicated server).

Table 1: **The storage stack, network stack and CPU scheduler used in the evaluated systems.**

System	Storage stack	Network stack	CPU scheduler
Linux	kernel, i10 [29]	kernel TCP	kernel CFS
SPDK	userspace	kernel TCP	kernel CFS

- Polling-based storage stacks (*e.g.*, SPDK) can achieve low latency and high throughput when each application is given a dedicated core. However, when multiple applications share a core, polling-based stacks that use kernel CPU schedulers suffer from undesirable interactions between the storage stack and the kernel CPU scheduler. Even when one L-app shares a core with one T-app, we observe 5× tail latency inflation *and* 2.4× throughput degradation, when compared to the respective isolated cases; both of these get worse with increasing number of applications sharing a core (108× tail latency inflation and 6.2× throughput degradation for the case of four L-apps sharing a core with one T-app). Prioritizing L-apps does not help—while tail latency inflation can be avoided, throughput for T-apps drops to near-zero with just one L-app.

2.1 Measurement Setup

The storage stack, the network stack and the CPU schedulers used in evaluated systems are summarized in Table 1. Linux uses block multi-queue design with per-core software queues mapped to underlying device driver queues (NVMe driver for local storage access, and i10 [29] queues for remote storage access). SPDK is a polling-based system, where applications poll on their I/O queues (for local storage access) and/or on their respective TCP sockets (for remote storage access); underneath, SPDK uses its own driver for accessing remote storage devices over TCP.

In §5, we evaluate these systems over different storage devices, workloads, and experimental setups. This section focuses on a specific setting: a single-core setup where one T-app contends with an increasing number of L-apps to execute read requests on remote in-memory storage connected via a 100Gbps link. This setting allows us to both hide high NVMe SSD access latencies, and dive deeper into factors contributing to individual system performance. Latency-sensitive L-apps generate 4KB requests and throughput-bound T-apps generate large requests; to ensure a fair comparison, for each individual system, we set the "ideal" load and request size for T-apps using the knee point on the latency-throughput curve for that system (see discussion in §5.1 for more details, including information about network and storage hardware).

We measure average and P99 tail latency for L-apps and throughput-per-core for T-apps for both isolated (where each application has host resources to itself) and shared scenarios (where all applications share host resources). An ideal system would maintain the isolated-latency for L-apps, with minimal impact on isolated throughput for T-apps.

2.2 Existing Storage Stacks: Low latency *or* high throughput, but not both

We start by discussing the isolated performance for each of the systems (shown in Figure 2 in the leftmost bars). Here, Linux achieves P99 tail latency of 118μs and throughput-per-core of 26Gbps; when compared to Linux, SPDK achieves 5× lower latency, and 1.5× higher throughput. While these results are not surprising in comparison, some interesting numbers stand out in an absolute sense. In particular, the absolute numbers for Linux—118μs P99 tail latency (comparable to our NVMe SSD access latency) and > 25Gbps throughput-per-core—may be surprising. We attribute these to several relatively recent optimizations in the Linux storage stack (*e.g.*, blk-mq [19] and CPU-efficient remote storage stacks [29]).

High tail latencies due to lack of prioritization: head-of-line (HoL) blocking. In early incarnations of Linux storage stacks, requests submitted at all cores were processed at a single queue, resulting in contention across cores as well as HoL blocking due to requests submitted across cores. Today's Linux alleviates these issues using per-core block layer queues [19]; however, we find that HoL blocking can still happen at the block layer queues (rare) or at the device driver queues (more prominent). This is because the Linux storage stack [19, 29] uses a single per-core non-preemptive thread to process all requests submitted on that core. When multiple applications contend on a core, this results in high tail latencies for L-apps due to HoL blocking caused by large requests from T-apps; we observe as much as 7× higher latencies in Figure 2. Figure 3(a) shows that, as expected, the impact of HoL blocking increases linearly with the request size of T-apps.

High tail latencies due to lack of prioritization: fair CPU scheduling. We find that polling-based designs do not interplay well with the default kernel CPU scheduler—Completely Fair Scheduler (CFS)—that allocates CPU resources equally across applications sharing the core (albeit, at coarse-grained millisecond timescales, referred to as "timeslices"). Polling completely utilizes the core; thus, the scheduler deallocates the core from an application only when the application has used its share of the core. As a result, requests from L-apps initiated at the boundary of L-app timeslices are the ones whose latency is impacted the worst since these would not be processed until the application's next timeslice. As a result, even when one L-app contends with one T-app, SPDK suffers from 5× inflation in tail latency when compared to the isolated case; the latency inflation increases to 108× and higher when multiple L-apps share the core with a T-app. Since CPU is fairly shared across contending applications, such polling-based systems not only suffer from latency inflation but also from degraded throughput for T-apps proportional to the number of applications contending at the core.

The impact on tail latency depends on two factors: (1) length of individual timeslices; and (2) the time gap between successive timeslices. The former determines the number of

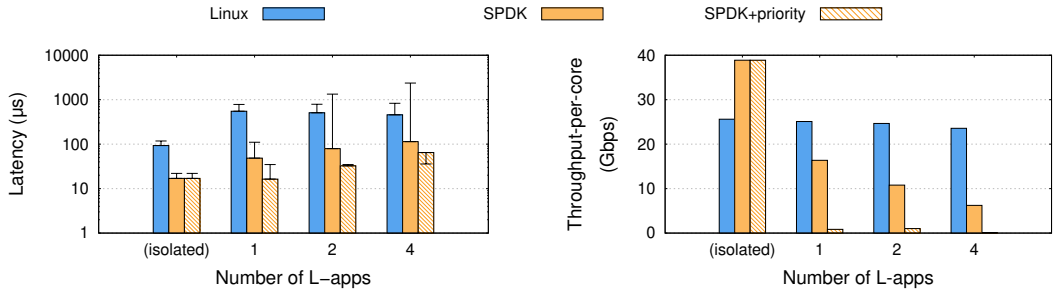


Figure 2: **When each application runs in isolation (isolated case, with no other applications on the server), existing storage stacks can achieve low latency *and* high throughput. However, when multiple applications compete for host resources, performance of existing storage stacks stumbles**—they are either unable to maintail μs-scale latency (Linux and SPDK), or are unable to maintain high throughput (SPDK+priority). With increasing number of L-apps contending with the T-app, performance degrades further. See §2.2 for discussion.

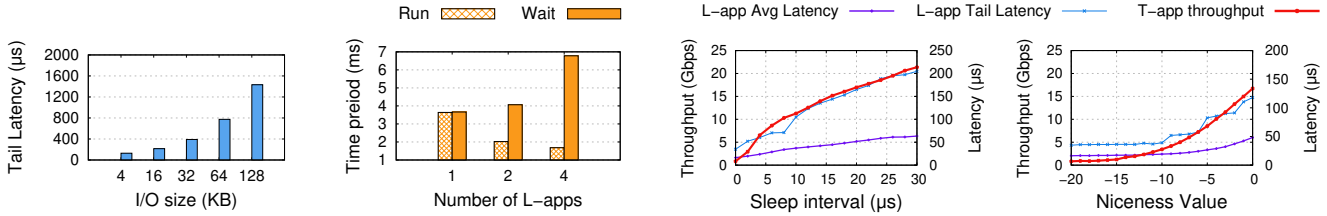


Figure 3: **Root cause for the trends in Figure 2. (left–right) (a): Linux** suffers from high tail latency due to HoL blocking. **(b): SPDK** suffers from high tail latency and low throughput since CPU scheduler performs fair scheduling of CPU resources, resulting in increasingly higher waiting times and increasingly lower runtimes for each application. **(c, d): SPDK+priority** suffers from complete starvation of T-app; increasing the sleep interval and/or niceness value of L-apps leads to an increase in T-app throughput, at the cost of increased average and tail latency for L-apps. Detailed discussion in §2.2.

requests that can be processed within a single timeslice (these requests will achieve near-optimal latency), and the latter determines the amount of "waiting time" for requests that could not be processed within the timeslice in which they were submitted. We measure these two factors in our experiments by examining CFS scheduler traces. In Figure 3(b), the "Run" bar shows the average length of the timeslice given to each L-app, and the "Wait" bar shows the average time gap between consecutive timeslices of each L-app. We observe that as the number of L-apps increases, the length of individual L-app timeslices decreases, and the wait time increases. This leads to (1) a larger latency impact for requests at the boundary of timeslices, hence inflation in tail latency; and, (2) a larger fraction of requests being impacted by the gap between consecutive timeslices, hence inflation in average latency.

Near-zero throughput due to strict prioritization: starvation in polling-based designs. Linux CPU scheduler allows prioritization of L-apps. Unfortunately, polling-based designs do not interplay well with prioritization either. We rerun SPDK results above but with L-apps having higher priority (niceness value -20) than T-app. The corresponding results, referred to as "SPDK+priority" in figures, show that such prioritization results in two undesirable effects: (1) complete starvation of T-apps—since L-apps have higher priority and are always active due to their polling-based design, the scheduler does not preempt these applications; and (2) if more than a single L-app contend on a core, CPU resources are shared fairly across these applications, resulting in increased average latency. We note that tail latency is not impacted much when the number of L-apps is increased. This is because, when given higher priority, L-apps get longer timeslices, and are able to process more requests in each timeslice, leaving only a small fraction of requests to be impacted by the gap between consecutive timeslices. Hence, while the waiting time between timeslices increases, the effect is not visible at P99 (higher percentiles see significant inflation). This is also the reason for the case of four L-apps in Fig. 2: the tail latency is worse than the average (since the latency distribution is extremely skewed towards higher percentiles).

In Figure 3(c), we re-run the single L-app and T-app case, this time making the L-app sleep for a certain interval after submitting requests, and vary this interval. When the L-app sleeps, it yields, allowing the T-app to get scheduled. As can be seen, increasing the sleep interval leads to an increase in T-app throughput. However, it comes at the cost of increasing tail latency for L-apps. In Figure 3(d), we repeat the single L-app and single T-app experiment, but with varying the L-app priority by adapting the niceness value (lower niceness implies higher priority): T-app's niceness value is set to 0, and we vary L-app niceness value from -20 (highest priority) to 0. CFS allocates timeslices to processes based on the niceness value. Hence, with increasing niceness values, the L-app gets a smaller share of CPU cycles, leading to an increase in the T-app's share. As a result, T-app's throughput increases but only at the cost of inflated latency for the L-app.

3 blk-switch Design

As mentioned earlier, blk-switch builds upon the insight that Linux's per-core block layer queues [19, 27], combined with modern multi-queue storage and network hardware [8], makes the storage stack conceptually similar to network switches. To realize the above insight, blk-switch introduces a "switched" architecture for the Linux storage stack that allows requests submitted by an application to be steered to and processed at any core in the system. In §3.1, we describe this switched architecture, and how it enables the key technique in blk-switch to achieve low latency for L-apps—prioritized processing of individual requests. In §3.2 and in §3.3, we describe how decoupling the application-side queues from device-side queues, and interconnecting them via blk-switch's switched architecture enables efficient realization of different load balancing strategies to achieve high throughput for T-apps.

Before diving deeper into blk-switch design details, we make two important notes. First, we describe blk-switch design using a single target device (local and/or remote storage server) since, similar to Linux, blk-switch treats each target device completely independently. Second, blk-switch does not require modifications in applications and/or system interface—applications submit I/O requests to the kernel via standard APIs such as `io_submit()`. Similar to any other system that provides differential service, blk-switch must identify application goals. Being within the Linux kernel makes this task easy for blk-switch: it uses the standard Linux ionice interface [6] that allows setting a "scheduling class" for individual applications/processes (without any changes in applications and/or kernel request submission interface). In the current implementation (§4), blk-switch uses two of the ionice classes to differentiate L-apps from T-apps. It is easy to extend blk-switch to support additional application requirements—for instance, applications that require both low latency and high throughput can use an additional application class (using ionice) to specify their performance goal, and blk-switch can be extended in a manner that each core not only appropriately prioritizes but also performs load balancing for requests for such applications. In addition, the ionice interface also allows applications to dynamically change their class, if performance goals change over time (*e.g.*, from latency-sensitive to throughput-sensitive requests). Note that ionice is only for the storage stack interpretation, and is different from CPU scheduling priority classes.

3.1 Block Layer is the New Switch

Linux storage stack architecture, in particular the block layer, has evolved over time. In early incarnations of Linux storage stacks, requests submitted at all cores were processed at a single queue. In today's Linux, block layer uses a per-core queue (blk-mq [19]) where requests submitted by all applications running on that core are processed. We refer to these per-core block layer queues as *ingress* queues. Today, these ingress queues are directly mapped to the driver queues (storage device driver for local storage access, or remote storage driver [21, 29] for remote storage access)[1]. Introduction of per-core ingress queues in Linux storage stack resolved contention across cores; however, since all requests submitted to an ingress queue are processed at the same core, it can lead to high tail latency due to head-of-line blocking at the driver queues when L-apps and T-apps submit requests to the same ingress queue (Figure 2). blk-switch's architecture avoids this using a multi-egress queue design, that we describe next.

Multiple egress queues. blk-switch introduces a per-core, multi-egress queue block layer architecture for the Linux storage stack. For each class of application running on the server, blk-switch creates an "egress" queue on a per-core basis. Each of these egress queues is mapped to a unique queue of the underlying device driver—storage driver for local storage access, and remote storage driver [29] with a dedicated network connection for remote storage access. blk-switch assigns a dedicated kernel thread for processing each individual egress queue and assigns priorities to these threads based on application performance goals. For instance, in the case of L-apps and T-apps, blk-switch assigns highest priority to the thread processing L-app requests (both in transmit and receive queues); thus, at each individual core, the kernel CPU scheduler will prioritize the processing of L-app requests over T-app requests, immediately preempting the T-app request processing thread. As a result, the latency inflation observed by L-app requests over the isolated case is minimal: in addition to the necessary overhead of a context switch, the only source of latency is other L-app requests on that core.

Decoupling request processing from application cores. Existing block layer multi-queue design tightly couples request processing to the core where the application submits the request. While efficient when cores are underutilized, such a design could result in suboptimal core utilization: if a core `C0` is overloaded and another core `C1` is underloaded, current storage stacks do not utilize `C1` cycles to process requests submitted at `C0`.

blk-switch exploits its multi-egress queue design to enable a switched architecture that alleviates this limitation (Figure 4): it allows requests submitted at a core to be steered from the ingress queue of that core to any of the other cores' egress queues (for that application class), be processed on that core, and responses returned on that core to be rerouted back to the appropriate application core. Decoupling request processing from application cores has some overheads (both in terms of latency and CPU), but allows blk-switch to ef-

[1] Modern storage devices have multiple hardware queues and corresponding drivers allow creating a large number of queues (*e.g.*, NVMe standard allows creating as many as 64k queues); in case of multiple hardware devices, each device has its own set of queues. Similarly, modern remote storage stacks [29] also create one driver queue per-core for each remote server.

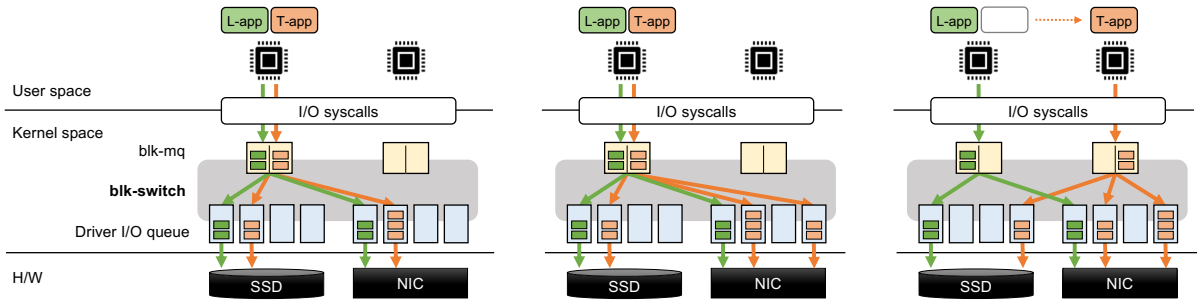


Figure 4: **An illustration of `blk-switch`'s design. (left)** multi-egress queue architecture: the first two and the last two queues on each device are for the left and the right core, respectively (one for each application class); **(center)** request steering mechanism: upon transient congestion on left core NIC queue, requests are steered on the queue corresponding to the right core ; **(right)** application steering mechanism: upon persistent congestion on the left core NIC queue, T-app is steered to the right core. See §3 for discussion.

ficiently utilize all cores in the system. For instance, in the case of L-apps and T-apps, `blk-switch` can steer requests to and process them at lightly-loaded cores, improving throughput for T-apps. Moreover, among requests processed on each core, `blk-switch` continues to provide isolation: prioritized processing of requests in L-app egress queues ensures that requests from L-apps are always prioritized over other requests.

We will discuss in §4 how existing block layer infrastructure (*e.g.*, `bio` and `request` data structures) enable efficient implementation of such a switched architecture with minimal modifications. The rest of the section describes `blk-switch` mechanisms to efficiently exploit this switched architecture to achieve high throughput for T-apps.

3.2 Request Steering

Decoupling processing of individual requests from application cores via `blk-switch`'s switched architecture enables efficient realization of different load balancing strategies. In this subsection, we describe one such strategy used in `blk-switch` for efficiently handling transient loads on individual cores—request steering.

Transient loads can result in temporarily starving T-app requests, *e.g.*, when a burst of (high-priority) L-app requests end up temporarily consuming all CPU cycles on a core, or when multiple L-apps on a core end up generating requests at the same time, or when large requests from one T-app block requests from other T-apps on that core to be processed, etc. Under such transient loads, `blk-switch` uses request steering to load balance the load on the system across the available cores—it steers T-app requests at ingress queues of transiently overloaded cores to the corresponding egress queues on other cores at the granularity of individual T-app requests. Importantly, `blk-switch` performs request steering only for throughput-bound applications. Request steering incurs some overheads (*e.g.*, latency due to reduced data locality, and CPU overheads due to request steering processing and due to contention among cores for accessing the same egress queue), but it is a good tradeoff to make for T-apps: during transient loads, `blk-switch` is able to efficiently utilize available CPU cycles at other cores to maintain T-app throughput. Figure 4(center) shows an example.

Algorithm 1 : `blk-switch` request steering framework.

`request` processing on local core (for destination T):

```
1: if load on local core < threshold then
2:     Move the request to local core's egress queue
3: else
4:     candidates ← cores with egress queue to T
5:     for each core ∈ candidates do
6:         if load on the core > threshold then
7:             remove core from candidates
8:     Randomly pick two cores in candidates
9:     Move the request to the core with smaller load
```

Making request steering decisions requires an estimate of instantaneous load on individual cores in the system. For T-apps where I/O is the main bottleneck, `blk-switch`'s multi-egress queue design enables an efficient approach—using the instantaneous sum of bytes of outstanding requests for the T-app egress queue to determine instantaneous per-core load and to steer requests to lightly-loaded cores. For such applications, instantaneous sum of bytes of outstanding requests is a good indicator of the presence of congestion in the end-to-end datapath, as congestion at any point will eventually build up the amount of bytes of outstanding requests in T-app egress queues. In our implementation for T-app requests that perform data access to remote storage servers, we use a default threshold of 16×64KB based on the latency-throughput curve for T-apps [30]. However, without any additional mechanisms, such an approach could lead to imperfect request steering decisions since it does not take into account the many other important factors (*e.g.*, queueing delay, request type being read/write, compute-I/O ratios, etc.); there is a large body of research on estimating load on the cores [15, 22, 46], and any of these mechanisms can be incorporated within `blk-switch` decision making.

Algorithm 1 shows a general framework for `blk-switch`'s request steering mechanism. `blk-switch` performs request

steering at the granularity of individual requests. Upon a request submission, `blk-switch` first checks if the local core is available: if the load on the local core is less than `threshold`, the local core is considered available and the request is enqueued in its egress queue. This is to ensure that `blk-switch` only incurs the overhead of request steering when necessary. If the local core is overloaded, `blk-switch` uses a mechanism based on power-of-two choices [41] to select a core to steer the request to. Among egress queues to the same destination (as described in §3.1), it randomly chooses two of these cores, and steers the request to the core with the lower load. The power-of-two choices is efficient as (1) at most two egress ports need to be examined when the local core is overloaded; and (2) it reduces contention between cores on queues since two cores are unlikely to write requests to the same core at the same time.

We provide details about `blk-switch`'s request steering implementation in §4. `blk-switch` does not implement request steering at the remote storage server side; if there is transient congestion at the remote storage server, then corresponding egress queues at the application side will build up. In that case, our application-side request steering algorithm will not pick this egress queue, and will forward the requests to queues at other cores. Thus, application-side request steering alone is enough to deal with transient congestion at both the application and the remote storage server.

3.3 Application Steering

The benefits of request steering at a per-request granularity can be overshadowed if each request submitted at a core has to be steered to other cores, *e.g.*, due to persistent load on a core due to multiple contending L-apps submitting requests at that core. For instance, if L-apps generate requests at low but consistent loads, frequent context switching between L-app and T-app request processing threads leads to reduced throughput. Similarly, if two high-load T-apps are contending on a core, it is better to move one of them to a less utilized core, avoiding long-term overheads of request steering.

To handle such persistent loads, `blk-switch` observes that load balancing within the Linux storage stack can be done at two levels of abstraction: individual requests and individual threads—while the former enables efficient handling of transient loads, the latter enables efficient handling of persistent loads. Thus, under such persistent loads, `blk-switch` performs application steering, that is CPU allocation to individual application threads by steering threads from persistently overloaded cores to one of the underutilized cores. Figure 4 shows an example. `blk-switch` performs application steering at coarse-grained timescales (in our implementation, default is 10 milliseconds) since it is required only for handling persistent loads. Note that application steering is performed at the granularity of individual application threads. Unlike request steering, `blk-switch` implements a version of application steering at both the application side and at the remote storage server; for the latter, it steers threads that perform processing at `blk-switch`'s receive-side egress queues.

Algorithm 2 : `blk-switch` application steering framework.

$\hat{L}_c$: weighted average load induced by L-apps at core c.
$\hat{T}_c$: weighted average load induced by T-apps at core c.
$L^\star$: threshold on weighted average load induced by L-apps

L-apps:

1: `candidates` $\leftarrow$ all cores with $0 < \hat{L}_c < L^\star$
2: $c^\star \leftarrow$ core in `candidates` with minimum $\{\hat{L}_c + \hat{T}_c\}$
3: Move the application to $c^\star$

T-apps:

1: `candidates` $\leftarrow$ all cores with $\hat{L}_c$ less than local core
2: $\hat{c} \leftarrow$ core in `candidates` with minimum $\hat{T}_c$
3: Move the application to $\hat{c}$

For application steering, `blk-switch` uses a framework similar to request steering with minor modifications (Algorithm 2). Unlike the request steering framework, `blk-switch`'s application steering explicitly takes into account the weighted average load on the core induced by L-apps. This is due to two reasons. First, application steering is performed to reduce long-term contention between L-apps and T-apps; thus, we want T-apps to be steered to the core with low weighted average load induced by L-apps (with an additional constraint that the weighted average of T-app load on the new core is lower than the current core). Together, this ensures that steering the T-app does not increase the number of context switches (the new core has lower L-app load), and also that the new core's T-app load is lower than that of the current core, thus minimizing contention among T-apps. Second, we also want to potentially place multiple L-apps on the same core in order to further reduce interference between L-apps and T-apps—colocating L-apps on a core will not negatively impact their performance as long as L-apps generate low weighted average load on the core. The second modification is for the case of applications performing data access on remote storage servers: we now use a default threshold of $L^\star = 100$KB to ensure that only a small number of L-apps are aggregated on the same core.

4 `blk-switch` Implementation Details

We have implemented `blk-switch` in Linux kernel 5.4. Throughout the implementation, our focus was to reuse existing kernel storage stack infrastructure as much as possible. To that end, our current implementation adds just 928 lines of code—530 in blk-mq layer, 118 at device driver layer, and 280 for target-specific functions at remote storage layer. In this section, we summarize the core components of Linux kernel implementation that `blk-switch` uses, along with some of the interesting `blk-switch` implementation details.

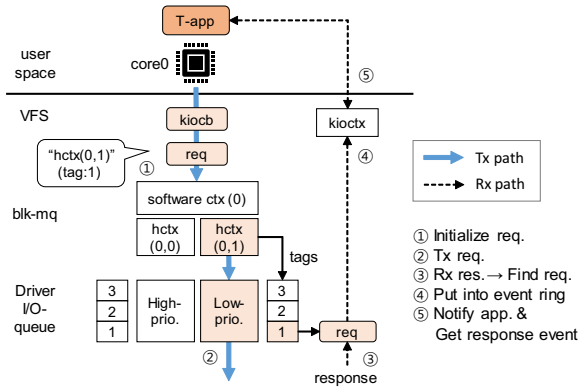
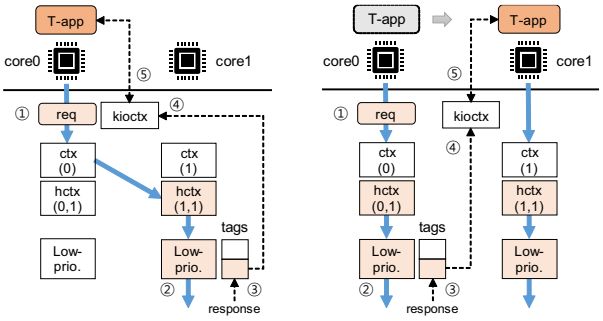


Figure 5: **Request datapath in `blk-switch` for T-app.** A request from T-app is forwarded to T-app egress queue obtaining a tag from that I/O queue. Linux maintains several data structures to enable forwarding back the response to the right application. `blk-switch` uses the same infrastructure.



(a) Request steering (b) Application steering

Figure 6: **Request datapath in `blk-switch`.** (a) **(w/ request steering):** request is steered to the queue on core1 via `hctx`(1,1) acquiring a tag from the steered queue. The response comes back to the steered queue on core1. (b) **(w/ application steering):** When an application is moved from core0 to core1, the *in-flight* request, sent before application steering, comes back on core0. `blk-switch` finds the corresponding `kioctx` via the tag and wakes up the application.

Linux block layer overview. We describe how the Linux storage stack works with the asynchronous I/O interface [4] (see Figure 5, but ignore prioritization). Before creating I/O requests, application needs to setup an I/O context using `io_setup()`, which creates a `kioctx` structure at VFS layer. This kioctx includes (1) a ring buffer where request completion events are enqueued (so that the application pulls them up later asynchronously); and, (2) application process information to wake up the application whenever a new completion event is ready. Each `kioctx` is associated with a context identifier. When application submits a request with the context identifier, the VFS layer creates `kiocb` that represents the I/O request and finds the corresponding `kioctx` using the identifier. `kiocb` has a pointer for the `kioctx`. The block layer creates a `bio` instance, based on `kiocb`, and encapsulates it in a `request` instance: this includes a hardware context (`hctx`) that is associated with one of the device-driver I/O queues. Before forwarding the `request` to the device-driver queue, the block layer needs to get a tag number. `tags` is an array of request pointers, and its size is the same as the queue depth of the driver queue.The block layer maintains a bitmap to keep track of the occupancy of the tags. When all tags are occupied (i.e., the driver queue is full), then the block layer needs to wait for a tag to be available. After getting the tag, the request is sent to the driver queue associated with hctx.

After I/O processing at the device, the response is returned to the kernel with the same tag number. The kernel finds the corresponding `request` instance from the `tags` array using the tag number. The tag number is released, and `kiocb` from the bio instance is extracted to find the `kioctx`. Finally, the completion event is enqueued into the ring buffer of `kioctx` and a notification is sent to the application.

`blk-switch` request steering implementation. Since each `hctx` is regarded as an egress queue, the main goal of the request steering algorithm is to select a non-congested `hctx` across cores if the local one is congested. `blk-switch` maintains the per-core load required for request steering (updating on a per-request basis). After that, the `request` will obtain a tag from the steered `hctx`. Once the `request` is enqueued into the corresponding driver queue, the following driver-level and block-layer receive processing will be done on the core that is associated with the steered `hctx`. When the response comes back to the kernel from the device, we are able to find the steered request instance from the `tags`; thus, going back to the original `kioctx` is straightforward as the `kioctx` can be extracted from the `request` instance (Figure 6(a)). The kernel sends a wake-up signal to the application running on the core associated with the ingress port via the `kioctx`.

`blk-switch` application steering implementation. Upon application steering deciding to move the application to a new core, `blk-switch` invokes the `sched_setaffinity` kernel function to execute the move. Once this is done, requests generated by the steered application will be submitted to the ingress queue on the new core. `blk-switch` maintains the weighted average per-core load required for application steering (updating on a per-request basis). It is easy to maintain application semantics even when there are "in-flight" requests during application moving from one core to another. `blk-switch` forwards the "in-flight" requests to the right application by exploiting the `tags` (Figure 6(b)); similar to the request steering, `blk-switch` is able to find the original `kioctx` that keeps track of the application's location and thus can wake up the associated application. Therefore, the responses can be delivered to the right application.

5 Evaluation

We now evaluate `blk-switch` performance, with the goal of understanding the envelope of workloads where `blk-switch` is able to provide μs-scale average and tail latency, while maintaining high throughput for T-apps. To do so, we evaluate

blk-switch across a variety of scenarios and workloads with varying amount of load induced by L-apps and T-apps, number of cores, read/write sizes, read/write ratios and storage settings (in-memory and SSD). In each evaluated scenario, a number of latency-sensitive applications (#L-apps) compete for host resources with a number of throughput-bound applications (#T-apps) that perform large read/write requests on remote storage servers. We describe the individual settings inline.

We describe our evaluation setup in §5.1, followed by a detailed discussion of our results in §5.2 and §5.3. Finally, in §5.4, we provide a detailed breakdown of how each design aspect of blk-switch contributes to its overall performance.

5.1 Evaluation Setup

blk-switch focuses on rearchitecting the storage stack for μs-scale latency and high throughput. Thus, our evaluation setup focuses on scenarios where performance bottlenecks are pushed to the storage stack—that is, where systems are bottlenecked by processing of storage requests, and not by network bandwidth.

Evaluated Systems. We compare blk-switch performance with Linux and widely-deployed userspace storage stack (SPDK) [51] (the CPU scheduler, storage stack and TCP/IP stack used for Linux and SPDK are shown in Table 1). For accessing data in remote servers, we make one modification in Linux: rather than using its native NVMe-over-TCP driver, we use i10 [29], a state-of-the-art Linux-based remote storage stack since it provides much higher throughput (using its default parameters, at the cost of introducing $\sim 50-100\mu$s latency at low loads); for accessing data on remote servers with SPDK, we use its native support for NVMe-over-TCP [13]. We apply core affinity to applications in Linux since that provides best performance. SPDK pins threads to cores by default since it makes use of DPDK's Environment Abstraction Layer (EAL). For both Linux and SPDK, we evenly distribute the applications across cores to the extent possible. For blk-switch, we use its default parameters (§3).

Hardware setup. All our experiments are run on a testbed with two servers directly connected via a 100Gbps link. The servers have a 4-socket NUMA-enabled Intel Xeon Gold 6234 3.3GHz CPU with 8 cores per socket, 384GB RAM and a 1.6TB Samsung PM1735 NVMe SSD. Both servers run Ubuntu 20.04 (kernel 5.4.43). To achieve CPU-efficient network processing for all evaluated systems (since all of them use Linux kernel network stack), we enable TCP Segmentation Offload (TSO), Generic Receive Offload (GRO), packet coalescing using Jumbo frames (9000B), and accelerated Receive Flow Steering (aRFS). To minimize experimental noise, we disable irqbalance and dynamic interrupt moderation (DIM) [10]. Finally, we disable hyper-threading since doing so maximizes performance for all evaluated systems.

We present results for both in-memory storage (RAM block device) and on-disk storage (NVMe SSD). Except for SSD and RocksDB experiments, we use the former due to three reasons. First, unlike on-disk storage, in-memory storage allows us to evaluate scenarios where T-apps generate load close to our network hardware capacity (100Gbps). Second, a single NVMe SSD can be saturated using two cores [29]; in-memory storage, on the other hand, allows us to evaluate scalability of blk-switch (and other systems) with larger number of cores. Finally, our NVMe SSDs have an access latency of $\sim 80\mu$s, which hides a lot of latency benefits of userspace stacks; we find it a fairer comparison to use in-memory storage to hide such high latencies.

Performance metrics. We evaluate system performance in terms of average and tail latency for L-apps, total throughput of all applications, and throughput-per-core calculated as "total throughput / core utilization" (we take the maximum of the application-side and the storage server-side core utilization when computing core utilization). Unless mentioned otherwise, we present results for average latency (shown by bars) and P99 tail latency (shown by top whiskers) since, as we will show, SPDK has significantly worse P99.9 tail latency.

Default workload. To generate loads for L-apps and T-apps, we use the standard methodology, where applications submit storage requests to the underlying system in a closed-loop (that is, the I/O depth of the application specifies a maximum number of outstanding requests). For Linux, we use FIO [16] that uses the lightweight libaio interface. For SPDK, we use its default benchmark application, perf (while FIO has been ported to SPDK, it has higher overheads compared to the lightweight perf application). These benchmarking applications are used to evaluate system performance to again push the bottlenecks to the underlying system (since real-world storage-based applications can have high overheads); nevertheless, we also evaluate blk-switch with RocksDB [9], a prominently used storage system.

L-apps generate 4KB read/write requests with an I/O depth of 1. To ensure that each system is running at its "knee-point" in its latency-throughput curve, we use the optimal T-app operating point for each system—for RAM block device, the optimal (request size, I/O depth) for T-apps is as follows: Linux (64KB, 32), SPDK (128KB, 8), and blk-switch (64KB, 16). While our default setup uses the above request sizes and I/O depths, we also present sensitivity analysis against varying I/O depths and request sizes. Finally, we use the random read workload in our default setup, and also present results for varying read/write ratios.

Unless stated otherwise, we give each system 6 cores on a single NUMA node. We use six cores for each system because we observed that, when given more than 6 cores, Linux ends up being bottlenecked by network bandwidth (that is, it can saturate the 100Gbps link in our testbed) in several of our experimental scenarios. Nevertheless, we also show performance with varying number of cores.

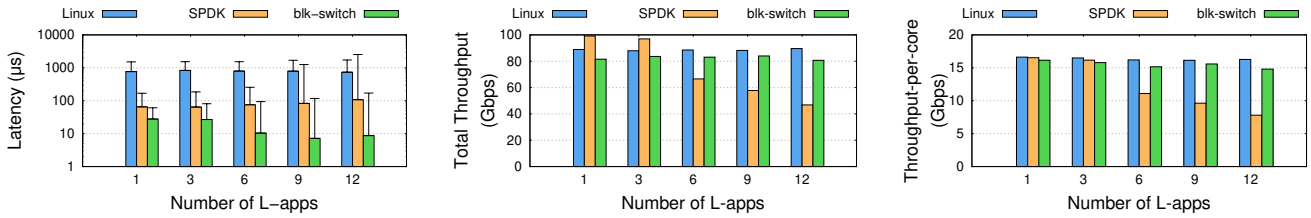


Figure 7: **`blk-switch` achieves μs-scale average and tail latency for L-apps and high throughput for T-apps even with tens of L-apps competing for host compute and network resources with T-apps.** As we increase the number of L-apps, both Linux and SPDK fail to simultaneously achieve low latency and high throughput for reasons discussed in §2. Linux achieves high throughput, but at the cost of high average and tail latencies; SPDK, on the other hand, suffers from both high tail latency and low throughput. Detailed discussion in §5.2.

5.2 Goal: Low-Latency *and* High-Throughput

Recall that an ideal system would ensure that both L-apps and T-apps observe performance close to the respective isolated performance (that is, when the application has all the host and storage server resources to itself).

Impact of increasing number of L-apps competing for host resources with T-apps (Figure 7 and Figure 8). For this experiment, each system is given six cores, and executes requests from six T-apps and varying number of L-apps.

Linux and SPDK performance trends are similar to Figure 2 in §2. Linux suffers from high average and tail latencies, but maintains high throughput even with increasing number of L-apps. SPDK achieves high throughput when number of L-apps is less than the number of cores; however, it suffers from inflated latency and degraded throughput with increasing number of L-apps (significantly degraded performance with just six L-apps). We already discussed the root cause for this behavior for each system in §2; however, for both Linux and SPDK, we observe slightly worse latency and throughput-per-core relative to that observed in Figure 2. Digging deeper, we found that both of these are due to increased L3 cache miss rates. Specifically, since the cores used by the systems are on the same NUMA node, they share a common L3 cache; the resulting increased contention for L3 cache leads to higher cache miss rate—for $x = 1$ in Figure 2, cache miss rates for Linux and SPDK are 1.12% and 3.68%, respectively, but for $x = 6$ in Figure 7, cache miss rates increase to 34% and 63%. Higher cache miss rates lead to an increase in the per-byte CPU overhead for kernel TCP processing (mainly due to data copy), resulting in lower throughput-per-core. Interestingly, for Linux, this also leads to higher latency inflation for L-apps (when comparing $x = 6$ in Figure 7 to $x = 1$ in Figure 2), as each T-app request takes a larger number of CPU cycles to process, hence exacerbating the effect of HoL blocking. Figure 8 single-threaded case shows the P99.9 tail latency for all systems for the $x = 6$ data point in Figure 7. Both Linux and SPDK exhibit high P99.9 tail latency, but SPDK in particular observes significantly worse P99.9 tail latency (33× higher than the P99). As discussed in §2, this is because L-app requests processed at the boundary of time slices are impacted, and this effect is prominently visible in higher percentiles.

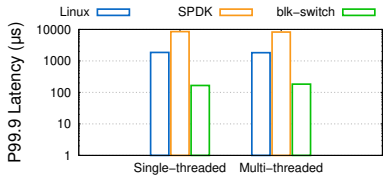


Figure 8: **The P99.9 tail latency corresponding to $x = 6$ in Figure 7 and Figure 9.**

`blk-switch` consistently achieves μs-scale latency for L-apps, even with 12 L-apps competing for host resources with 6 T-apps. In comparison to Linux, `blk-switch` achieves $28-110\times$ better average latency, $10-25\times$ better P99 tail latency and $6-15\times$ better P99.9 tail latency; in comparison to SPDK, `blk-switch` achieves $2-12\times$ better average latency, $2-15\times$ better P99 tail latency and $33-101\times$ better P99.9 tail latency. `blk-switch` achieves all these latency benefits while sacrificing $5-10\%$ throughput relative to Linux. `blk-switch` achieves such performance benefits using a combination of its techniques: it first performs application steering to isolate L-apps to a subset of cores, and to distribute T-apps over the remaining cores. This results in slightly increased tail latency for L-apps compared to a single L-app case, but significantly reduces context switching overheads when compared to L-apps and T-apps sharing individual cores. Further, `blk-switch` performs request steering to utilize unused L-app cores for processing T-app requests opportunistically. Finally, separation of I/O queues along with prioritization enables maintaining low latency for L-apps even when T-app requests are steered to the L-app cores. Note that prioritization of I/O queue processing also leads to `blk-switch` having slightly better average and tail latencies when compared to the isolated Linux latency in Figure 2; however, this is not fundamental.

We observe a somewhat surprising and counter-intuitive benefit of `blk-switch`'s application steering mechanism that steers L-apps onto a small number of cores—for example, in Figure 7, `blk-switch`'s average latency reduces with increasing number of L-apps. This is because of better packet aggregation opportunities through TSO/GRO and Jumbo frames: as more L-apps are steered on the same core, they begin to

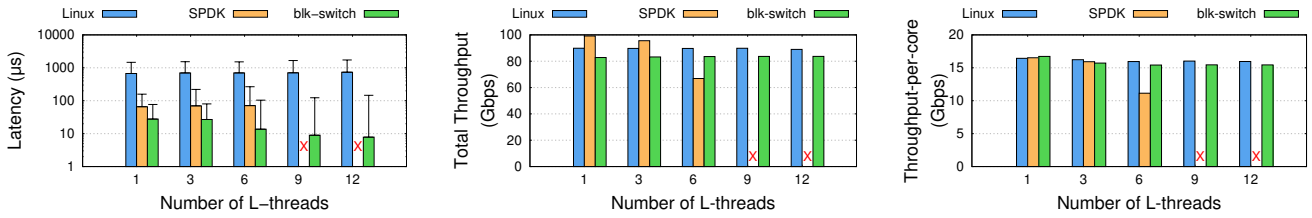


Figure 9: **`blk-switch` achieves μs-scale average and tail latency for `L`-apps and high throughput for `T`-apps even when tens of `L`-app threads compete for host compute and network resources with `T`-apps.** We observe the same trend as in Figure 7 for each system; the only difference is that SPDK does not support more application threads than the number of cores.

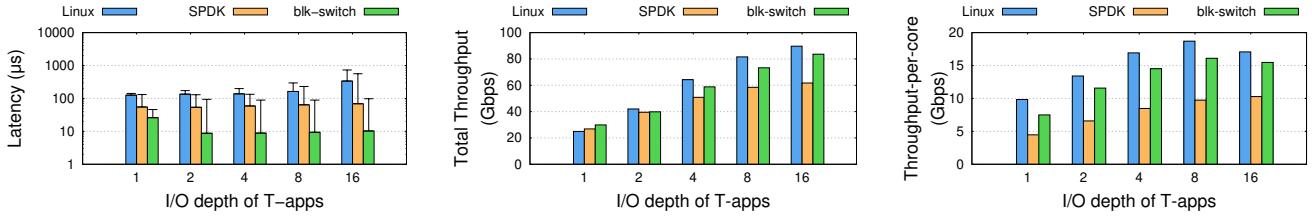


Figure 10: **As the load induced by `T`-apps increases, `blk-switch` continues to achieve low latency and high throughput.** For reasons discussed in §2, Linux and SPDK fail to simultaneously achieve low latency and high throughput: Linux suffers from high latency due to HoL blocking; SPDK experiences increasingly higher latency and lower throughput as the load induced by `T`-apps increase.

share the same egress queue and hence the same underlying TCP connection (recall that `blk-switch` maintains a single per-core egress queue for each application class); as a result, more `L`-app requests can be aggregated, resulting in lower per-request processing overheads, and improved average latency.

Impact of increasing number of `L`-app threads competing for host resources with `T`-app threads (Figure 9 and Figure 8). We now evaluate the performance of existing storage stacks for multi-threaded applications. To do this, we slightly modify the evaluation setup from Figure 7 experiment: we now use one `T`-app with six `T`-threads and one `L`-app with varying number of `L`-threads (varying from 1 to 12). Note that, while the recent SPDK NVMe-oF target implementation supports user-level threads [13], SPDK's `perf` benchmark application running on the host-side does not support user-level threads; as a result, it does not support creating more threads than the number of cores in the system (for each individual application). As one would expect, Figure 9 and Figure 8 results show exactly the same trend as single-threaded applications.

Impact of increasing the load induced by `T`-apps (Figure 10). We now evaluate the performance of each system with varying load induced by `T`-apps. There are two ways to vary the load induced by `T`-apps—by varying I/O depth, and by varying request sizes. Since our setup uses TSO/GRO, these two mechanisms to vary the load induced by `T`-apps lead to essentially the same set of results. We present and discuss results for the former here; the latter can be found in [30]. For this experiment, we fix the number of `L`-apps and `T`-apps to 6 each, and increase the I/O depth for `T`-apps. The request size for `T`-apps is now fixed to 64KB for all systems.

Linux and SPDK show trends similar to previous results. Average and tail latencies for `L`-apps increase with increased contention for host resources (in these results, increased contention is due to higher load induced by `T`-apps). As one would expect, for both of these systems, total throughput and throughput-per-core for `T`-apps increases with an increase in load induced by `T`-apps. `blk-switch` handles contention differently from both of these systems: by prioritizing `L`-app requests, and using request and application steering to efficiently load balance `T`-app requests across unused cores. Thus, `blk-switch` continues to maintain μs-scale latency with increase in `T`-app load—in comparison to Linux, `blk-switch` achieves $5-33\times$ lower average and $2-8\times$ lower tail latency; in comparison to SPDK, `blk-switch` achieves $2-7\times$ lower average and $1.3-6\times$ lower tail latency. `blk-switch`'s mechanisms for handling contention results in a slightly different tradeoff in terms of `T`-app performance. When the load induced by `T`-apps is small, `blk-switch` reduces Linux latency without any degradation in throughput (since it does not pay the overheads of request steering at low loads); at higher loads, `blk-switch` continues to achieve low latency, but observes 10% lower throughput than Linux due to the overheads of request steering.

We note that `blk-switch` average latency improves with load induced by `T`-apps. For smaller loads, `blk-switch`'s application steering does not steer `L`-apps on to a subset of cores (as in previous experiments), leaving `L`-apps evenly distributed across available cores. As a result, `blk-switch` does not get to exploit the benefits of reduced per-request processing overheads (due to TSO/GRO and jumbo frames) associated with aggregating multiple `L`-apps on the same core.

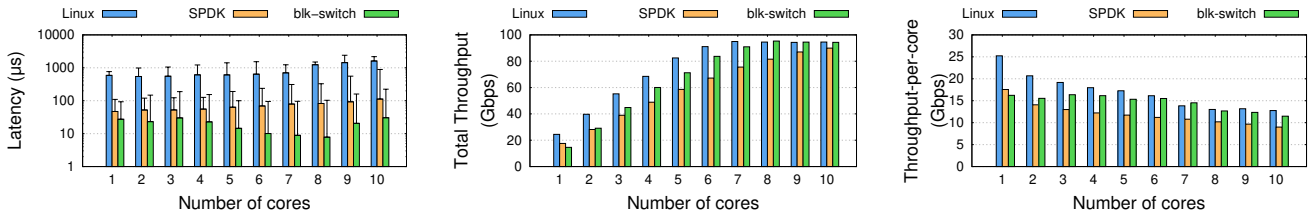


Figure 11: `blk-switch` **maintains its μs-scale average and tail latency for** `L`**-apps with varying number of cores, even when scheduling across NUMA nodes.** For small number of cores, compared to Linux, `blk-switch` trades off improvements in latency with slightly reduced throughput (due to request prioritization, and fewer opportunities for application and request steering). For smaller number of cores, SPDK achieves low latency; as the number of cores are increased, SPDK starts suffering from inflated tail latency and degraded throughput.

Impact of number of cores (Figure 11). We now evaluate the performance of all systems with varying number of cores, including the case when the cores belong to different NUMA nodes. The challenge with doing this evaluation is that, if `T`-apps were to not interfere with `L`-apps, ∼4 cores would be sufficient to saturate the network bandwidth (as can be inferred from the isolated case in Figure 2); thus, to understand the performance with increasing number of cores, we have to ensure that `L`-apps and `T`-apps continue to contend at host storage and network processing resources rather than competing for network bandwidth. Thus, we use the following evaluation strategy. Our servers have eight cores on each NUMA node; for each data point up to x = 8 on the x-axis (x = number of cores used for that data point), we use the cores on the same NUMA node and for the last two data points, we use two additional cores from one of the other NUMA nodes. For each data point, we run a total of x `L`-apps and x `T`-apps to ensure that the system is neither lightly-loaded nor overloaded. With this setup, we are able to evaluate for larger number of cores—Linux, `blk-switch` and SPDK now become network bandwidth bottlenecked at 7, 8 and 10 cores, respectively.

Linux and SPDK performance can be explained using our prior insights. As the number of cores increase, Linux experiences increasingly higher latency but is able to achieve high throughput; SPDK, on the other hand, suffers from increasingly higher latency, and relatively lower throughput.

For the single core case, `blk-switch` improves Linux's latency, but at the cost of 40% lower throughput (similar to SPDK); this is due to lack of request steering and application steering opportunities, and due to prioritization being the dominant mechanism for isolation. As the number of cores increase, `blk-switch` starts exploiting the benefits of request and application steering—it achieves μs-scale latency as in earlier experiments, while getting throughput increasingly closer to Linux (with 7 cores, it is only 4.2% worse than Linux; for 8 or more cores, `blk-switch`'s throughput matches Linux as the network link is saturated). For number of cores between 3 and 8, we see a reduction in `blk-switch`'s average latency due to higher opportunities to exploit the benefits of TSO, GRO and jumbo frames (due to application steering aggregating increasingly more `L`-apps on same subset

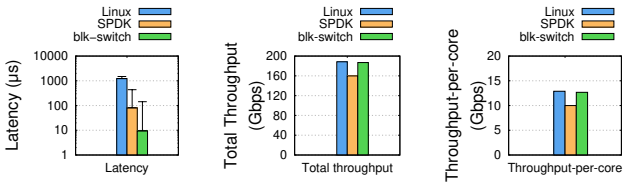


Figure 12: `blk-switch` **is able to maintain low average and tail latencies even when applications operate at throughput close to** 200**Gbps**. The experiment uses 16 `L`-apps and 16 `T`-apps running across 16 cores from two NUMA nodes.

of cores). Beyond 8 cores, we see slight increase in average and tail latency for `blk-switch` because of NUMA effects.

Besides latency results, there are several other interesting observations to be made in Figure 11(center). First, `blk-switch` is able to completely saturate a 100Gbps link using 8 cores, at which point it is bottlenecked by network bandwidth. Since the server has many more cores, we expect that these cores will allow `blk-switch` to maintain its performance with future NICs that have larger bandwidths (we show this for 200Gbps network bandwidth setup below). Second, while the total throughput of `blk-switch` scales well with the number of cores, it has slightly lower total throughput compared to Linux for smaller number of cores. This is due to application steering resulting in `T`-apps being steered away from `L`-apps, and the `L`-apps cores observing transient underutilization when request steering decisions are imperfect. Under such imperfect decisions, fewer number of cores are available for `T`-app request processing. However, as the number of cores increase, the benefits of reduced context switching (due to lower contention between `L`-app and `T`-app requests after application steering) start to offset core underutilization resulting in similar or even higher throughput when compared to other systems. Finally, Figure 11(right) demonstrates that all systems experience reduced throughput-per-core with increasing number of cores. We found that this is due to an increased number of L3 cache misses with increase in total throughput as the number of cores is increased.

Performance beyond 100**Gbps (Figure 12).** We now evaluate the performance of all systems in the Terabit Ethernet

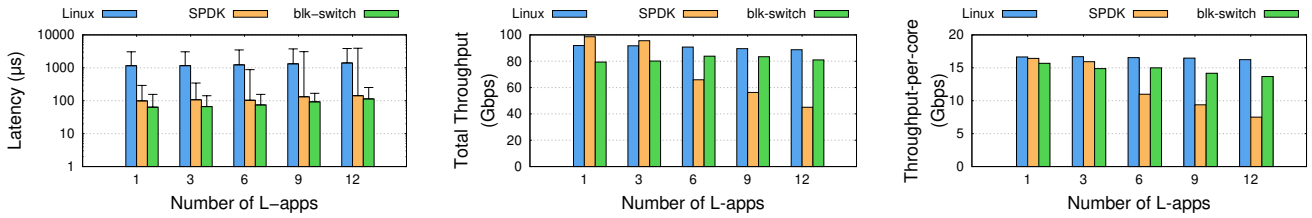


Figure 13: **For experiments with SSDs (corresponding to Figure 7), `blk-switch` latency is largely overshadowed by SSD access latency.** Rest of the trends are similar to those in Figure 7.

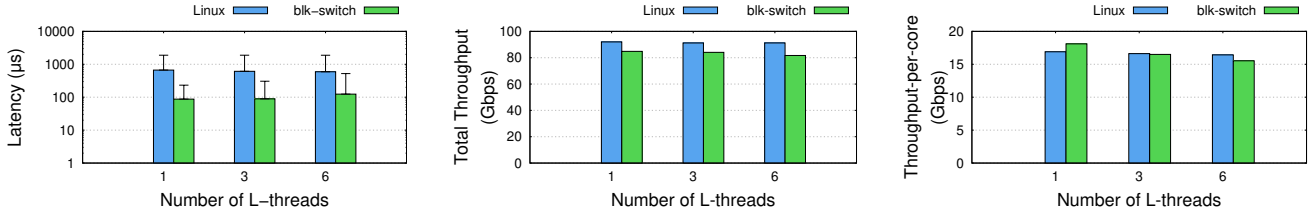


Figure 14: **Evaluation results with RocksDB:** `blk-switch` performance benefits over Linux are similar to previous results.

regime (above 100Gbps). For this we installed an additional NIC on each of the two servers in our setup, and connected these NICs with an additional 100Gbps link, enabling a total of 200Gbps network bandwidth between the servers. The two NICs on each server are attached to separate NUMA nodes. We use all of the cores on both of these NUMA nodes (total of 16), while running 16 L-apps and 16 T-apps. The performance trends remain identical to previous results — `blk-switch` is able to maintain μs-scale average and tail latency (10μs average, 143μs P99 , and 296μs P99.9), while nearly saturating the 200Gbps network bandwidth (within 1% of Linux).

Performance with different storage access latency. We repeat the experiment shown in Figure 7, but with L-app requests being executed on an NVMe SSD (T-app requests are still executed in-memory). The access latency of our SSD ($\sim$80μs) causes increase in average latencies for all systems, but the performance trends among the evaluated systems remain identical to earlier results. Importantly, `blk-switch`'s latency is largely overshadowed by SSD access latency.

Additional results. We present several additional results in [30], including performance with varying request sizes for T-apps, varying read/write ratios, applications that access data distributed between local and remote storage servers, and bursty application workloads.

5.3 RocksDB with `blk-switch`

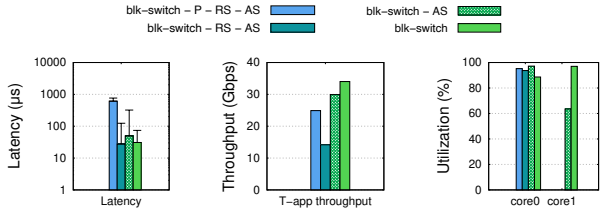
We now evaluate `blk-switch` with RocksDB [9], a widely-deployed storage system, as the L-app. We mount a remote SSD block device at the host-side with XFS file system (only Linux and `blk-switch` support mounting a file system). We setup RocksDB to use the mounted XFS file system backed by remote SSD device and enable direct I/O. To generate workload for RocksDB, we use the `db_bench` benchmarking tool with ReadRandom workload and 4KB request sizes, with an I/O depth of 1 for each thread. We colocate a T-app that accesses remote RAM block device using FIO [16], as before. We run this benchmark on 6 cores, with 6 T-app threads and varying number of L-app threads.

Figure 14 shows that both Linux and `blk-switch` achieve slightly higher latency compared to previous results due to RocksDB's higher application-layer overheads. However, in comparison, `blk-switch` achieves over an order of magnitude latency reduction when compared to Linux, while sacrificing throughput by at most 10%. Furthermore, `blk-switch` maintains these benefits even with increasing number of L-app threads competing for host resources with T-app threads.

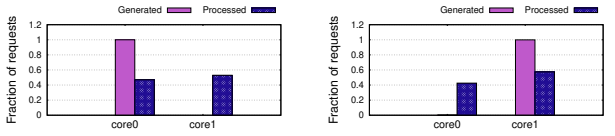
5.4 Understanding `blk-switch` Performance

We now quantify the contribution of each of `blk-switch`'s mechanisms to its overall performance. To do so, we run a simple microbenchmark: we start the experiment with one L-app and one T-app on `core0`, and set the I/O depth of T-app to be 32. We then add `blk-switch` mechanisms (prioritization, request steering and application steering) incrementally.

Figure 15 shows that each of `blk-switch`'s mechanism contributes to its overall performance. Enabling prioritization only reduces tail latency by an order of magnitude (Figure 15(a)), but at the cost of lower T-app throughput on `core0` (Figure 15(b)); since request and application steering are disabled, strictly prioritizing processing of L-app requests results in reduced throughput due to larger number of context switches. As shown in Figure 15(c) and Figure 15(d), enabling request steering with prioritization allows the T-app to achieve high T-app throughput by utilizing spare capacity on less loaded cores (by steering T-app requests from heavily loaded `core0` and processing these requests at `core1`); however, this comes at the cost of slight increase in latency for



(a) Latency (b) T-app throughput (c) Core utilization



(d) Fraction of requests generated and processed at each core, **before** application steering.

(e) Fraction of requests generated and processed at each core, **after** application steering.

Figure 15: **Contribution of different techniques in `blk-switch` to its overall performance.** (`blk-switch`-P-RS-AS) is `blk-switch` with all mechanisms disabled; we then cumulatively enable prioritization (`blk-switch`-RS-AS), request steering (`blk-switch`-AS), and application steering (`blk-switch`). See discussion in §5.4.

L-apps (albeit, still μs-scale)—due to non-trivial CPU overheads of request steering and non-real-time prioritization in Linux kernel CPU schedulers, some of the L-app requests get blocked by the thread doing request steering. This problem is alleviated by `blk-switch`'s application steering algorithm (Figure 15(e))—it steers the T-app away from the L-app, allowing `blk-switch` to simultaneously achieve low latency and high throughput.

6 Related Work

We have already compared `blk-switch` with state-of-the-art Linux-based and widely-deployed userspace storage stacks. We now compare and contrast `blk-switch` with other closely-related systems.

Existing storage stacks. There is a large and active body of research on designing storage stacks that target various goals, including fairness [1, 2, 7, 26], deadlines [5, 7], prioritization [3], and even policy-based storage provisioning and management [24, 39, 47, 49]. However, none of these stacks target μs-scale latency. Furthermore, many of them can have high CPU overheads (for high-performance storage devices, the standard recommendation in Linux is to use no scheduler [26]), especially for applications that perform operations on remote storage servers [14, 23, 25, 50]. Recent work on storage stacks for remote data access [12, 29] achieves high CPU efficiency and throughput; however, as we have shown in our evaluation, they fail to achieve low latency in multi-tenant deployments when latency-sensitive and throughput-bound applications compete for host resources.

User-space stacks. We have already performed evaluation against SPDK, a widely-deployed state-of-the-art user-space storage stack. Our evaluation focuses on using SPDK with Linux kernel CPU scheduler and network stack, and highlights the poor interplay with SPDK's polling-based architecture and Linux CPU scheduler. It is possible to overcome some of these limitations by integrating SPDK with high-performance user-space or RDMA-based network stacks [13, 18, 32, 35–37, 40], user-space CPU schedulers [34], or both [22, 42–45]. However, with the exception of [22, 42], these user-space network stacks and CPU schedulers either do not provide μs-scale isolation in multi-tenant deployments, or require dedicated cores for each individual L-app resulting in potentially high core underutilization. The state-of-the-art among these user-space stacks [22, 42] demonstrate that by carefully orchestrating compute resources across L-apps and T-apps, it is possible to simultaneously achieve μs-scale latency and high throughput. However, they currently provide fewer features than Linux and require modifications in applications. `blk-switch` shows that it is possible to simultaneously achieve μs-scale latency and high throughput without any modifications in applications, Linux kernel CPU scheduler and/or network stack.

Hardware-level isolation. There has also been work on achieving performance isolation by exploiting hardware-level mechanisms in NVMe SSDs [20, 28, 48], including mechanism specification in the NVMe standard [11, 33]. These are complementary to `blk-switch`'s goals that focuses on software bottlenecks.

7 Conclusion

Using design, implementation and evaluation of `blk-switch`, this paper demonstrates that it is possible to achieve μs-scale tail latency using Linux, even when tens of latency-sensitive applications compete for host resources with throughput-bound applications that access data at throughput close to hardware capacity. The key insight in `blk-switch` is that Linux's multi-queue storage design, along with multi-queue network and storage hardware, makes the storage stack conceptually similar to a network switch. `blk-switch` uses this connection to adapt techniques from the computer networking literature (*e.g.*, prioritized processing of individual requests, load balancing, and switch scheduling) to the Linux kernel storage stack. `blk-switch` is implemented entirely within the Linux kernel storage stack, and requires no modification in applications, network and storage hardware, kernel CPU schedulers and/or kernel network stack.

Acknowledgments

We would like to thank our shepherd, Adam Belay, and the OSDI reviewers for their insightful feedback. This work was supported in part by NSF 1704742, NSF 1900457, a Google faculty research scholar award, a Sloan fellowship, the Texas Systems Research Consortium, and a grant from Samsung.

References

[1] BFQ (Budget Fair Queueing) — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[2] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/.

[4] Linux Asynchronous I/O. https://oxnz.github.io/2016/10/13/linux-aio/.

[5] Linux blk-mq scheduling framework. https://lwn.net/Articles/708465/.

[6] Linux ionice. https://linux.die.net/man/1/ionice.

[7] Linux Kernel/Reference/IOSchedulers - Ubuntu Wiki. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers.

[8] David S. Miller, Linux Multiqueue Networking. http://vger.kernel.org/~davem/davem_nyc09.pdf, 2009.

[9] Facebook Inc., RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/, 2015.

[10] Mellanox Technologies: Dynamically-Tuned Interrupt Moderation (DIM). https://community.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2019.

[11] NVM Express 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2019.

[12] NVM Express over Fabrics 1.1. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf, 2019.

[13] SPDK User Guides for NVMe over Fabrics. https://spdk.io/doc/nvmf.html, 2020.

[14] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[15] R. Apte, L. Hu, K. Schwan, and A. Ghosh. Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization. In *USENIX HotCloud*, 2010.

[16] J. Axboe. Flexible IO Tester (FIO) ver 3.13. https://github.com/axboe/fio, 2019.

[17] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[18] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[19] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *ACM SYSTOR*, 2013.

[20] M. Bjørling, J. Gonzalez, and P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *USENIX FAST*, 2017.

[21] N. Express. NVM Express over Fabrics 1.0 Ratified TPs. https://nvmexpress.org/, 2018.

[22] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*, 2020.

[23] P. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[24] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-Defined Storage for Multi-tenant Object Stores. In *USENIX FAST*, 2017.

[25] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*, 2017.

[26] M. Hedayati, K. Shen, M. L. Scott, and M. Marty. Multi-Queue Fair Queuing. In *USENIX ATC*, 2019.

[27] C. Hellwig. High Performance Storage with blk-mq and scsi-mq. https://events.static.linuxfound.org/sites/events/files/slides/scsi.pdf.

[28] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *USENIX FAST*, 2017.

[29] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*, 2020.

[30] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. https://github.com/resource-disaggregation/blk-switch/techreport, 2021.

[31] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. N. H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX ATC*, 2018.

[32] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, 2014.

[33] K. Joshi, K. Yadav, and P. Choudhary. Enabling NVMe WRR support in Linux Block Layer. In *USENIX HotStorage*, 2017.

[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] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, 2019.

[36] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[37] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *ACM ASPLOS*, 2017.

[38] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM SoCC*, 2014.

[39] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *USENIX NSDI*, 2015.

[40] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kokonov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[41] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001.

[42] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), Nov. 2015.

[44] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[45] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-Aware Thread Management. In *USENIX OSDI*, 2018.

[46] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *ACM SoCC*, 2011.

[47] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the Storage Stack Like a Network. In *USENIX FAST*, 2016.

[48] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. Mansouri Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *ACM ISCA*, 2018.

[49] 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.

[50] 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.

[51] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A development kit to build high performance storage applications. In *IEEE CloudCom*, 2017.

[52] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *ACM Eurosys*, 2013.

Optimizing Storage Performance with Calibrated Interrupts

Amy Tai[‡*] Igor Smolyar[†*] Michael Wei[‡] Dan Tsafrir[†‡]

[†]*Technion – Israel Institute of Technology* [‡]*VMware Research*

Abstract

After request completion, an I/O device must decide either to minimize latency by immediately firing an interrupt or to optimize for throughput by delaying the interrupt, anticipating that more requests will complete soon and help amortize the interrupt cost. Devices employ adaptive interrupt coalescing heuristics that try to balance between these opposing goals. Unfortunately, because devices lack the semantic information about which I/O requests are latency-sensitive, these heuristics can sometimes lead to disastrous results.

Instead, we propose addressing the root cause of the heuristics problem by allowing software to explicitly specify to the device if submitted requests are latency-sensitive. The device then "calibrates" its interrupts to completions of latency-sensitive requests. We focus on NVMe storage devices and show that it is natural to express these semantics in the kernel and the application and only requires a modest two-bit change to the device interface. Calibrated interrupts increase throughput by up to 35%, reduce CPU consumption by as much as 30%, and achieve up to 37% lower latency when interrupts are coalesced.

1 Introduction

Interrupts are a basic communication pattern between the operating system and devices. While interrupts enable concurrency and efficient completion delivery, the costs of interrupts and the context switches they produce are well documented in the literature [7, 30, 66, 73]. In storage, these costs have gained attention as new interconnects such as NVM Express™ (NVMe) enable applications to not only submit millions of requests per second, but up to 65,535 concurrent requests [18, 21, 22, 25, 75]. With so many concurrent requests, sending interrupts for every completion could result in an interrupt storm, grinding the system to a halt [40, 55]. Since CPU is already the bottleneck to driving high IOPS [37, 38, 41, 42, 43, 46, 69, 76, 81, 82], excessive interrupts can be fatal to the ability of software to fully utilize existing and future storage devices.

Typically, interrupt coalescing addresses interrupt storms by batching requests into a single interrupt. Batching, however, creates a trade-off between request latency and the interrupt rate. For the workloads we inspected, CPU utilization increases by as much as 55% without coalescing (Figure 12(d)), while under even the minimum amount of coalescing, request latency increases by as much as 10× for small requests, due to large timeouts. Interrupt coalescing is disabled by default in Linux, and real deployments use alternatives (§2).

This paper addresses the challenge of dealing with exponentially increasing interrupt rates without sacrificing latency. We initially implemented adaptive coalescing for NVMe, a dynamic, device-side-only approach that tries to adjust batching based on the workload, but find that it still adds unnecessary latency to requests (§3.2). This led to our core insight that device-side heuristics, such as our adaptive coalescing scheme, cannot achieve optimal latency because the device lacks the semantic context to infer the requester's intent: is the request latency-sensitive or part of a series of asynchronous requests that the requester completes in parallel? Sending this vital information to the device bridges the semantic gap and enables the device to interrupt the requester when appropriate.

We call this technique *calibrating*[1] interrupts (or simply, cinterrupts), achieved by adding two bits to requests sent to the device. With calibrated interrupts, hardware and software collaborate on interrupt generation and avoid interrupt storms while still delivering completions in a timely manner (§3).

Because cinterrupts modifies how storage devices generate interrupts, supporting it requires modifications to the device. However, these are minimal changes that would only require a firmware change in most devices. We build an emulator for cinterrupts in Linux 5.0.8, where requests run on real NVMe hardware, but hardware interrupts are emulated by interprocessor interrupts (§4).

Cinterrupts is only as good as the semantics that are sent to the device. We show that the Linux kernel can naturally annotate all I/O requests with default calibrations, simply by inspecting the system call that originated the I/O request (§4.1). We also modify the kernel to expose a system call interface that allows applications to override these defaults.

In microbenchmarks, cinterrupts matches the latency of state-of-the-art interrupt-driven approaches while spending

[*]Denotes co-first authors with equal contribution.

[1]To calibrate: to adjust precisely for a particular function [53].

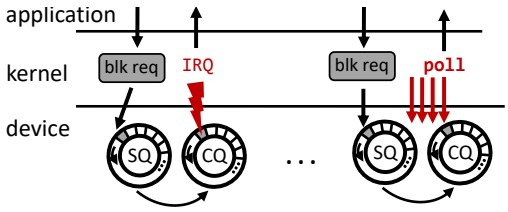


Figure 1: NVMe requests are submitted through submission queues (SQ) and placed in completion queues (CQ) when done. Applications are notified of completions via either interrupts or polling.

30% fewer cycles per request and improves throughput by as much as 35%. Without application-level modifications, cinterrupts uses default kernel calibrations to improve the throughput of RocksDB and KVell [48] on YCSB benchmarks by as much as 14% over the state-of-the-art and to reduce latency by up to 28% over our adaptive approach. A mere 42-line patch to use the modified syscall interface improves the throughput of RocksDB by up to 37% and reduces tail latency by up to 86% over traditional interrupts (§5.5.1), showing how application-level semantics can unlock even greater performance benefits. Alternative techniques favor specific workloads at the expense of others (§5).

Cinterrupts can, in principle, also be applied to network controllers (NICs), provided the underlying network protocol is modified to indicate which packets are latency sensitive. We demonstrate that despite being more mature than NVMe drives, NICs suffer from similar problems with respect to interrupts (§2). We then explain in detail why it is more challenging to deploy cinterrupts for NICs (§6).

2 Background and Related Work

Disks historically had seek times in the milliseconds and produced at most hundreds of interrupts per second, which meant interrupts worked well to enable software-level concurrency while avoiding costly overheads. However, new storage devices are built with solid state memory which can sustain not only millions of requests per second [25, 75], but also multiple concurrent requests. The NVMe specification [57] exposes this parallelism to software by providing multiple queues, up to 64K per device, where requests, up to 64K per queue, can be submitted and completed; Linux developers rewrote its block subsystem to match this multi-queue paradigm [9]. Figure 1 shows a high-level overview of NVMe request submission and completion.

Numerous kernel, application, and firmware-level improvements have been proposed in the literature to unlock the higher request rate of these devices [13, 39, 46, 48, 60, 65, 82, 83], but they focus on increasing I/O *submit* rate without directly addressing the problem of higher *completion* rate.

Lessons from Networking. Networking devices have had much higher completion rates for a long time. For example, 100Gbps networking cards can process over 100 million packets per second in each direction, over 200× that of a typical NVMe device. The networking community has devised two main strategies to deal with these completion rates: interrupt coalescing and polling.

To avoid bombarding the processor with interrupts, network devices apply interrupt coalescing [74], which waits until a threshold of packets is available or a timeout is triggered. Network stacks may also employ polling [16], where software queries for packets to process rather than being notified. IX [7] and DPDK [33] (as well as SPDK [67]) expose the device directly to the application, bypassing the kernel and the need for interrupts by implementing polling in userspace. Technologies such as Intel's DDIO [19] or ARM's ACP [56] enable networking devices to write incoming data directly into processor caches, making polling even faster by turning MMIO queries into cache hits. The networking community has also proposed various in-network switching and scheduling techniques to balance low-latency and high-throughput [3, 7, 35, 49].

Storage is adopting networking techniques. The NVMe specification standardizes the idea of interrupt coalescing for storage devices [57], where an interrupt will fire only if there is a sufficient threshold of items in the completion queue or after a timeout. There are two key problems with NVMe interrupt coalescing. First, NVMe only allows the aggregation time to be set in 100μs increments [57], while devices are approaching sub 10μs latencies. For example, in our setup, small requests that normally take 10μs are delayed by 100μs, resulting in a 10× latency increase. Intel Optane Gen 2 devices have latencies around 5μs [24], which would result in a 20× latency increase. The risk of such high latency amplification renders the timeout unusable in general-purpose deployments where the workload is unknown.

Second, even if the NVMe aggregation granularity were more reasonable, both the threshold and timeout are *statically* configured (the current NVMe standard and its implementations have no adaptive coalescing). This means interrupt coalescing easily breaks after small changes in workload—for example, if the workload temporarily cannot meet the threshold value. The NVMe standard even specifies that interrupt coalescing be turned off by default [58], and off-the-shelf NVMe devices ship with coalescing disabled.

Indeed, despite the existence of hardware-level coalescing, there are still continuous software and driver patches to deal with interrupt storms through mechanisms such as fine-tuning of threaded interrupt completions and polling completions [11, 47, 50]. Mailing list requests and product documentation show that Azure observes large latency increases when using aggressive interrupt coalescing to deal with interrupt storms, which they try to address with driver mitigations [26, 47]. Because of the proprietary nature of Azure's solution, it is unclear whether their interrupt coalescing is standard NVMe coalescing or some custom coalescing that they develop with hardware vendors.

Polling is expensive. μdepot [43] and Arrakis [61] deal

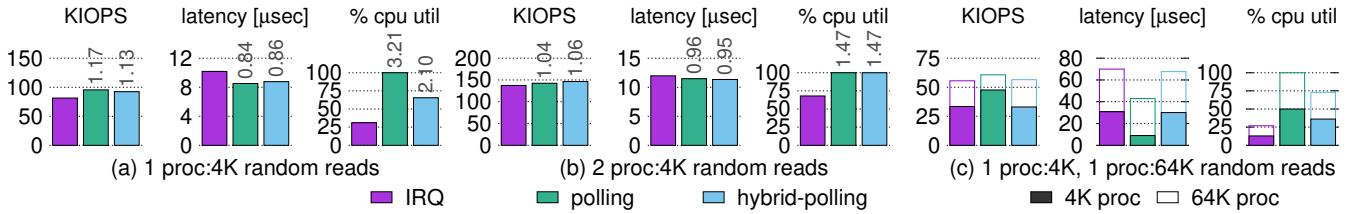


Figure 2: Hybrid polling is not enough to mitigate polling overheads. All experiments run on a single core. (a) With a single thread of same-sized requests, hybrid polling effectively reduces the CPU utilization of polling while matching the performance of polling. (b) With more threads, hybrid polling reduces to polling in terms of CPU utilization without providing significant performance improvement over interrupts. (c) With variable I/O sizes, hybrid polling has the same throughput and latency as interrupts for both I/O sizes while using 2.7x more CPU. Labels show performance relative to IRQ.

with higher completion rates by resorting to polling. Directly polling from userspace via SPDK requires considerable changes to the application, and polling from either kernel-space or userspace is known to waste CPU cycles [42, 77, 82]. FlashShare only polls for what it categorizes as low-latency applications [82], but acknowledges that this is still expensive. Cinterrupts exposes application semantics for interrupt generation so that systems do not have to resort to polling.

Even hybrid polling [79], which is a heuristic-based technique for reducing the CPU overhead of polling by sleeping for a period of time before starting to poll, is insufficient, breaking down when requests having varying size [5, 41, 45].

Figure 2 compares the performance and CPU utilization of hybrid polling, polling, and interrupts for three benchmarks on an Intel Optane DC P4800X [23].[2] We note that in all cases, polling provides the lowest latency because the polling thread discovers completions immediately, at the expense of 100% CPU utilization. When there is a single thread submitting requests through the read syscall (Figure 2(a)), hybrid polling does well because request completions are uniform. However, when more threads or I/O sizes are added, as in Figures 2(b)-(c), hybrid polling still has 1.5x-2.7x higher CPU utilization than interrupts without providing noticeable performance improvement. These results match other findings in the literature [5, 6, 41, 45].

Even though polling wastes cycles, it can provide lower latency than interrupts, which is desirable in some cases. As such, cinterrupts coexists with kernel-side polling, such as in Linux NAPI for networking [15, 16], which switches between polling and interrupts based on demand.

Heuristics-based Completion. vIC [2] tries to moderate virtual interrupts by estimating I/O completion time with heuristics. It primarily relies on inspecting the number of "commands-in-flight" to determine whether to coalesce interrupts, also employing smoothing mechanisms to ensure that the coalescing rate does not change too dramatically. To prevent latency increase in low-loaded scenarios, vIC also eliminates interrupt coalescing when the submission rate is below a certain threshold. vIC is a heuristic-based coalescing algorithm, similar to our adaptive algorithm (Section 3.2). Consequently, vIC also lacks semantic information necessary to align interrupt delivery.

[2] See Section 5.1 for a detailed description of our experimental setup.

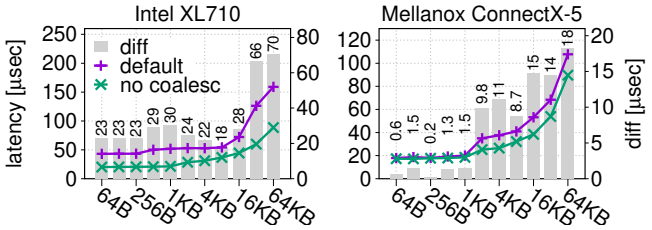


Figure 3: NICs employ adaptive heuristics that try to minimize interrupt overheads without unnecessarily hurting latency. The inherent imperfection of these heuristics is demonstrated using Intel and Mellanox NICs servicing the netperf request-response benchmark, which ping-pongs a message of a specified size. The labels show the latency difference between the default NIC scheme and a no-coalescing policy, which minimizes latency for this particular workload, but which harms performance for more throughput-oriented workloads.

NICs and their software stack are higher-performing and more mature than NVMe drives and their corresponding stack. As with NVMe devices, NICs must balance two contradictory goals: (1) reducing interrupt overhead via coalescing to help throughput-oriented workloads, while (2) providing low latency for latency-sensitive workloads by triggering interrupts upon incoming packets as soon as possible. Notably, NICs employ more sophisticated, adaptive interrupt coalescing schemes (implemented inside the device and helped by its driver). Yet, in general-purpose settings that must accommodate arbitrary workloads, NICs are unable to optimally fire and coalesce interrupts, despite their maturity.

Figure 3 demonstrates that heuristics in even mature devices cannot optimally resolve the interrupt delivery problem for all workloads. Two NICs, Intel XL710 40 GbE [20] and Mellanox ConnectX-5 100 GbE [71], run the standard latency-sensitive netperf TCP request-response (RR) benchmark [34], which repeatedly sends a message of a specified size to its peer and waits for an identical response. In this workload, the challenge for the NIC is identifying the end of the in-

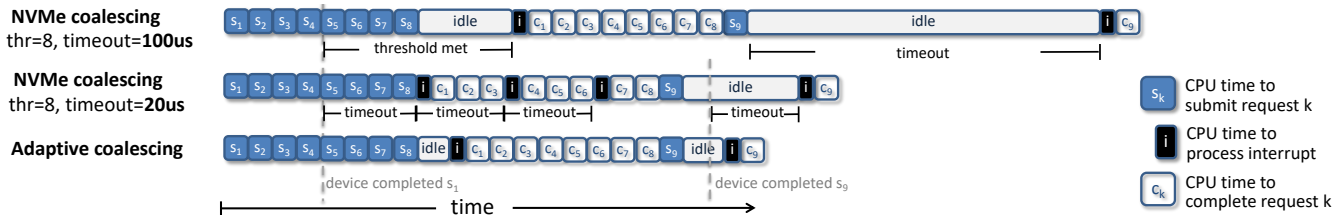


Figure 4: NVMe coalescing with its current 100μs granularity (top row) causes unusable timeouts when the threshold is not met (c_9). Even if the timeout granularity were smaller (middle row), NVMe coalescing cannot adapt to the workload. For bursts of requests, the smaller timeout will limit the throughput with interrupts ($c_1 - c_8$), while bursts that do not meet the threshold must still wait for the timeout (c_9). Note that idle periods occur when the CPU is done submitting requests, but is waiting for the device to complete I/O.

coming message and issuing the corresponding interrupt. The Intel NIC heuristic results in increased latency regardless of message size, and the Mellanox NIC heuristic adds latency if the message size is greater than 1500 bytes, the maximum transmission unit (MTU) for Ethernet, because the message becomes split across multiple packets.

Knowledgeable users with admin privileges may manually configure the NIC to avoid coalescing, which helps identify message boundaries and thus yields better results for this *specific* workload. But such a configuration is ill-suited for general-purpose setups that must reasonably support co-located throughput-oriented workloads as well.

Exposing Application-Level Semantics. Similar to [39, 78, 82], cinterrupts augments the syscall interface with a few bits so applications can transmit information to the kernel. Cinterrupts further shares this information with the device, which is only also done in [82], which shares with the device SLO information used to improve internal device caching.

3 Cinterrupts

3.1 Design Overview

The initial design of cinterrupts focused on making NVMe coalescing adapt to workload changes. Our first contribution captures this intuition with an adaptive coalescing strategy to replace the static NVMe algorithm (§3.2).

While our adaptive coalescing strategy improves over static coalescing, there are still cases that an adaptive strategy cannot handle, such as workloads with a mix of latency-sensitive and throughput-sensitive requests. The adaptive strategy also imposes an inevitable overhead from detecting when a workload has changed.

This observation led to the core insight of cinterrupts: device-level heuristics for coalescing will always fall short due to a semantic gap between the requester and the device, which sees a stream of requests and cannot determine which requests require interrupts in order to unblock the application. To bridge the semantic gap, the application issuing the I/O request should always inform the device when it wishes to be interrupted. Cinterrupts takes advantage of the fact that this semantic information is easily accessible in the storage stack and available at submission time.

Note on Methodology. The results throughout this section are obtained on a setup fully described in Section 5.1. We use an Intel Optane DC P4800X, 375 GB [23], installed in a Dell PowerEdge R730 machine equipped with two 14-core 2.0 GHz Intel Xeon E5-2660 v4 CPUs and 128 GB of memory running Ubuntu 16.04. The server runs cinterrupts' modified version of Linux 5.0.8 and has C-states, Turbo Boost (dynamic clock rate control), and SMT disabled. We use the maximum performance governor.

All results are obtained with our cinterrupts emulation, as described in Section 4.2.1. Our emulation pairs one dedicated core to one target core. Each target core is assigned its own NVMe submission and completion queue. All results in this section are run on a single target core.

3.2 Adaptive Coalescing

Ideally, an interrupt coalescing scheme adapts dynamically to the workload. Figure 4 shows that even if the timeout granularity in the NVMe specification were smaller, it is still *fixed*, which means that interrupts will be generated when the workload does not need interrupts ($c_1 - c_8$), while completions must wait for the timeout to expire (c_9) when the workload does need interrupts.

Instead, as shown in the bottom row of Figure 4, the adaptive coalescing strategy in cinterrupts observes that a device should generate a single interrupt for a burst, or a sequence of requests whose interarrival time is within some bound.

Algorithm 1 shows the adaptive strategy. The burst detection happens on Line 6, where the timeout is pushed out by Δ every time a new completion arrives. In contrast, NVMe coalescing cannot detect bursts because it does not dynamically update the timeout, which means it can only detect bursts of a fixed size. To bound request latency, the adaptive strategy uses a *thr* that is the maximum number of requests it will coalesce into a single interrupt (Lines 14-15). This is necessary for long-lived bursts to prevent infinite delay of request completion. With Algorithm 1, a device will emit interrupts when either it has observed a completion quiescent interval of Δ or *thr* requests have completed. In §5, we explain how device manufacturers and system administrators can determine

Algorithm 1: Adaptive coalescing strategy in cinterrupts

```
1  Parameters: Δ, thr
2  coalesced = 0, timeout = now + Δ;
3  while true do
4      while now < timeout do
5          while new completion arrival do
               /* burst detection, update timeout */
6              timeout = now + Δ;
7              if ++coalesced ≥ thr then
8                  fire IRQ and reset;
       /* end of quiescent period */
9      if coalesced > 0 then
10         fire IRQ and reset;
11     timeout = now + Δ;
```

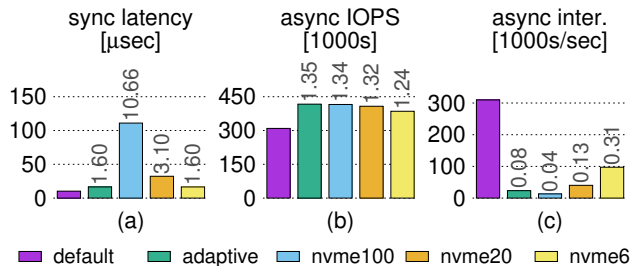


Figure 5: Adaptive strategy has better performance for both types of workloads regardless of how NVMe coalescing is configured. (a) latency of a synchronous read request. (b) throughput of an asynchronous read workload with high iodepth. (c) interrupt rate for async workload. Labels show performance relative to default.

reasonable Δ and *thr* configurations.

Comparison to NVMe Coalescing. The adaptive strategy outperforms various NVMe coalescing configurations, even those with smaller timeouts, across different workloads. We compare the adaptive strategy, configured with $thr = 32$, $\Delta = 6$, to no coalescing (default), nvme100, which uses a timeout of 100μs, the smallest possible in standard NVMe, nvme20, which uses a theoretical timeout of 20μs, and nvme6, which uses a theoretical timeout of 6μs. All NVMe coalescing configurations have threshold set to 32.

We run two single-threaded synthetic workloads with fio [4]. In the first workload, the thread submits 4 KB read requests via read, which blocks until the system call is done. In the second workload, the thread submits 4 KB read requests via libaio in batches of 16, with iodepth=512.[3]

Figure 5(a) reports the latency of the read requests for the synchronous workload. As expected, the default strategy has the lowest latency of 10μs, because it generates an interrupt for every request. All coalescing strategies add exactly their timeout to the latency of each read request (c_9 in Figure 4). Because they have the same timeout, nvme6 and adaptive have the same latency, but nvme6 pays the price for this low

[3] iodepth represents the number of in-flight requests.

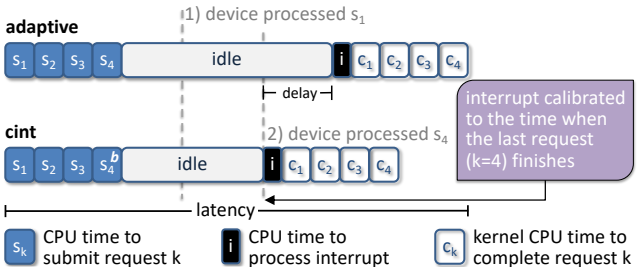


Figure 6: Completion timeline for multiple submissions. The adaptive strategy can detect them as part of a burst, but only after the delay expires. Cinterrupts explicitly marks the last request in the batch. Idle periods occur when the CPU waits for I/O to complete.

timeout in the next workload.

Figure 5(b) reports the read IOPS for the second workload and shows that if there are enough requests to hit the threshold, the timeout adds unnecessary interrupts ($c_1 - c_8$ in Figure 4). The default strategy's throughput is limited because it generates too many interrupts, as shown in Figure 5(c).

The workload has enough requests in flight that waiting for the threshold of 32 completions does not harm throughput. However, nvme20 and nvme6 *must* fire an interrupt every 20μs or 6μs, respectively: Figure 5(c) shows that nvme20 generates 1.7x more interrupts than adaptive, and nvme6 generates 4.2x more interrupts than adaptive, explaining their lower throughput.

The adaptive strategy can accurately detect bursts, although it adds Δ delay to confirm the end of a burst; without additional information, this delay is unavoidable. Figure 6 shows how cinterrupts addresses this problem by enhancing adaptive with two annotations, Urgent and Barrier, which software passes to the device. We now describe both annotations.

3.3 Urgent

Urgent is used to request an interrupt for a single request: the device will generate an immediate interrupt for any request annotated with Urgent. The primary use for Urgent is to enable the device to calibrate interrupts for latency-sensitive requests. Urgent eliminates the delay in the adaptive strategy.

To demonstrate the effectiveness of Urgent, we run a synthetic mixed workload with fio with two threads: one submitting 4 KB read requests via libaio with iodepth=16 and one submitting 4 KB read requests via read, which blocks until the system call is done. In cinterrupts, the latency-sensitive read requests are annotated with Urgent, which is embedded in the NVMe request that is sent to the device (see §4.1.1). Results are shown in Figure 7.

Without cinterrupts, the requests from either thread are *indistinguishable* to the device. The default (no coalescing) strategy addresses this problem by generating an interrupt for every request, resulting in 2.7x more interrupts than cinterrupts (Figure 7(d)).

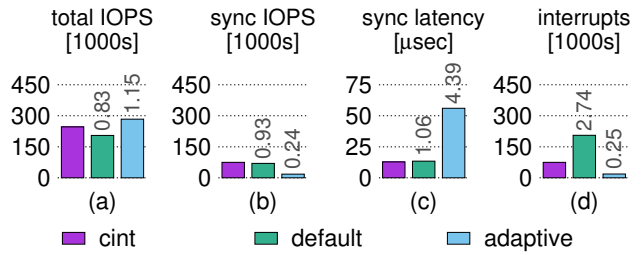


Figure 7: Effect of Urgent. Synthetic workload with two threads running a mixed workload: one thread submitting synchronous requests via read, one thread submitting asynchronous requests via libaio. Cinterrupts achieves optimal synchronous latency and better throughput over the default (no coalescing). The adaptive strategy achieves better overall throughput, at the expense of synchronous latency. Labels show performance relative to cinterrupts.

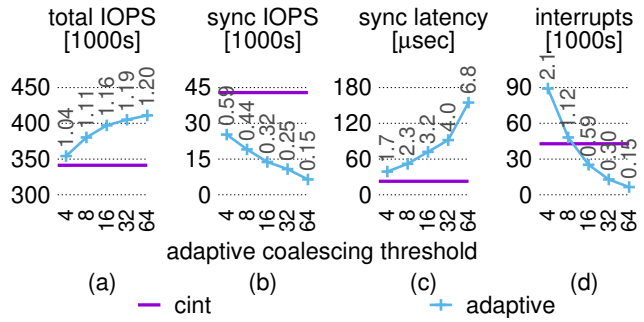


Figure 8: In a mixed workload, increasing the coalescing threshold increases the latency of synchronous requests proportionally to the coalescing rate. Labels show performance relative to cinterrupts.

On the other hand, with Urgent, cinterrupts calibrates interrupts to the latency-sensitive read requests, enabling low-latency without generating needless interrupts that hamper the throughput of the asynchronous thread. This results in both higher asynchronous throughput and lower latency for the synchronous requests. The adaptive strategy is unable to identify the latency-sensitive requests and in fact tries to minimize interrupts for all requests, resulting in higher asynchronous throughput but a corresponding increase in read request latency (Figure 7(c)).

In fact, the more aggressive the coalescing, the more unusable synchronous latencies become. Figure 8 shows the same experiment with higher iodepth. As the target coalescing rate increases, there is a corresponding linear increase in the synchronous latency. On the other hand, the purple line in Figure 8(c) shows that Urgent in cinterrupts makes synchronous latency acceptable. This latency comes at the expense of less asynchronous throughput, as shown in Figure 8(a), but we believe this is an acceptable trade-off.

3.4 Barrier

To calibrate interrupts for batches of requests, cinterrupts uses Barrier, which marks the end of a batch and instructs the device to generate an interrupt as soon as all preceding requests have finished. The semantic difference between Urgent and Barrier is that an Urgent interrupt is generated as soon as the Urgent request finishes, whereas the Barrier interrupt may have to wait if requests are completed out of order.

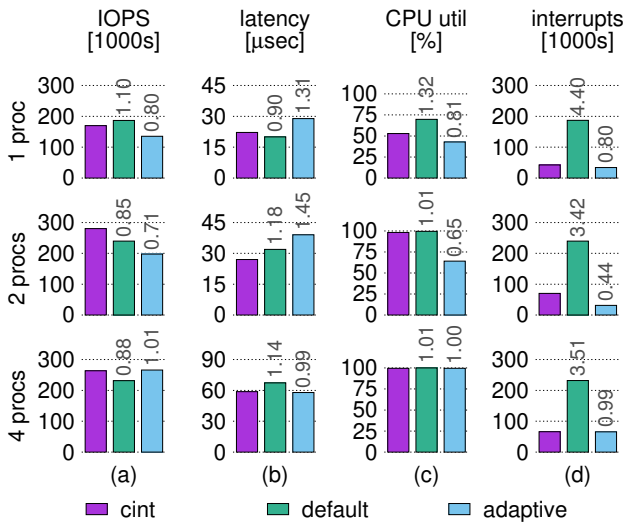


Figure 9: Effect of Barrier. Each process submits a batch of 4 requests at a time, submitting a new batch after the previous batch has finished. Cinterrupts always detects the end of a batch with Barrier. Note that when there is CPU idleness, adaptive always adds Δ delay to the latency. Labels show performance relative to cinterrupts.

Barrier minimizes the interrupt rate, which is always beneficial for CPU utilization, while enabling the device to generate enough interrupts so that the application is not blocked. For example, in the submission stream $s_1 - s_4$ in Figure 6, the last request in the batch, s_4, is marked with Barrier.

To demonstrate the effectiveness of Barrier, we run an experiment with a variable number of threads on the same core, where each thread is doing 4 KB random reads through libaio, submitting in fixed batch sizes of 4. The trick is determining the end of the batch without additional overhead, which is only possible in cinterrupts: we modify fio to mark the last request in each batch with a Barrier. Figure 9 shows the throughput, latency, CPU utilization, and interrupt rate.

Single Thread. When there is a single thread, the default (no coalescing) strategy can deliver lower latency than cinterrupts. This is because there is CPU idleness and no other thread running. However, the default strategy generates 4.4x the number of interrupts as cinterrupts, which results in 1.32x CPU utilization. The default strategy can also process some completions in parallel with device processing, whereas cinterrupts waits for all completions in the batch to arrive before processing. On the other hand, the Δ delay in the adaptive algorithm is clear: the latency of requests is 29 μs, compared to 22 μs with cinterrupts.

Two Threads. When there are two threads in the experiment, the advantage of the default strategy goes away: the 3.4x

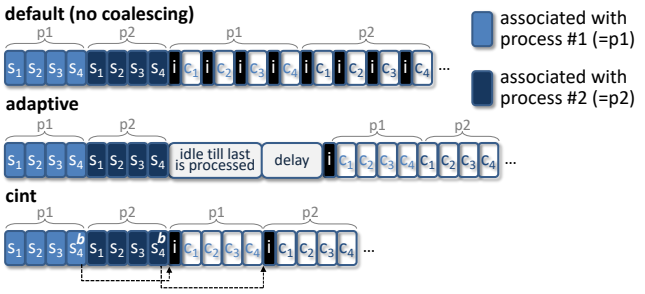


Figure 10: Completion timeline for two threads submitting request batches. The adaptive strategy experiences CPU idleness both because of the delay and because it waits to process any completions until they all arrive. On the other hand, due to Barrier, cinterrupts can process each batch as soon as it completes.

interrupts taxes a saturated CPU. On the other hand, cinterrupts has the best throughput and latency because calibrating interrupts enable better CPU usage.

Figure 10 shows that the adaptive strategy exhibits highest synchronous latency due to CPU idleness, which comes from waiting for completions and the delay used to detect the end of the batch. This idleness is eliminated in the next experiment, where there are enough threads keep the CPU busy.

Four Threads. With four threads, the comparison between cinterrupts and the default NVMe strategy remains the same. However, at four threads, the adaptive strategy matches the performance of cinterrupts because without CPU idleness, the delay is less of a factor. Although the adaptive strategy does well in this last case, we showed in §3.3 that this aggregation comes at the expense of synchronous requests.

Note that Figure 10 is a simplification of a real execution, because it conflates time spent in userspace and the kernel, and does not show completion reordering. The full cinterrupts algorithm addresses reordering by employing the adaptive strategy to ensure no requests get stuck.

3.5 Out-of-Order Urgent

The full cinterrupts interrupt generation strategy is shown in Algorithm 2. Requests are either unmarked or marked by Urgent or Barrier. Unmarked requests are handled by the underlying adaptive algorithm and can of course piggyback on interrupts generated by Urgent or Barrier.

We noticed that Urgent requests sometimes get completed with other requests, which increases their latency because the interrupt handler does not return until it reaps all requests in the interrupt context. To address this, cinterrupts implements out-of-order (OOO) processing, a driver-level optimization for Urgent requests. With OOO processing, the IRQ handler will only reap Urgent requests in the interrupt context, which enables faster return-to-userspace of the Urgent requests.

Unmarked requests will not be reaped until a completion batch consists only of those requests, as shown in Figure 11.

Algorithm 2: cinterrupts coalescing strategy

```
1  Parameters: Δ, thr
2  coalesced = 0, timeout = now + Δ;
3  while true do
4      while now < timeout do
5          while new completion arrival do
6              timeout = now + Δ;
7              if completion type == Urgent then
8                  if ooo processing is enabled then
                       /* only urgent requests */
9                      fire urgent IRQ;
10                 else
                       /* process all requests */
11                     fire IRQ and reset coalesced;
12             if completion type == Barrier then
13                 fire IRQ and reset coalesced;
14             else
15                 if ++coalesced ≥ thr then
16                     fire IRQ and reset coalesced;
       /* end of quiescent period */
17     if coalesced > 0 then
18         fire IRQ and reset coalesced;
19     timeout = now + Δ;
```

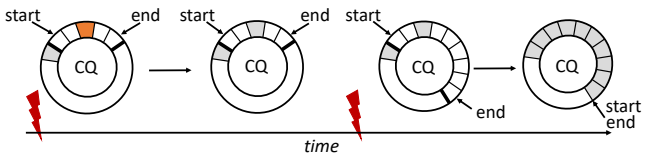


Figure 11: OOO Urgent processing. Grayed entries are reaped entries. Urgent requests in an interrupt context (first interrupt) are processed immediately, and the interrupt handler returns. The other requests are not reaped until the next interrupt, which consists only of non-Urgent requests. After the second IRQ, the driver rings the completion queue doorbell to signal that the device can reclaim the contiguous range.

The driver also does not ring the CQ doorbell until it completes a contiguous range of entries. *thr* ensures non-Urgent requests are eventually reaped. For example, suppose in Figure 11 that $thr = 9$. Then an interrupt will fire as soon as 9 entries (already reaped or otherwise) accumulate in the completion queue.

The trade-off with OOO processing is an increase in the number of interrupts generated. Figure 12 reports performance metrics from running the same mixed workload as in Figure 8. OOO processing generates 2.4x the number of interrupts in order to reduce the latency of synchronous requests by almost half. The impact of the additional interrupts is noticeable in the reduced number of asynchronous IOPS.

Incidentally, these additional interrupts, as well as the interrupts in the default strategy, act as an inadvertent tax on the asynchronous thread. If we instead limit the number of asynchronous requests, the need for these additional interrupts goes away. In the second row of Figure 12, we throttle the asynchronous thread with the blkio cgroup [10] to its

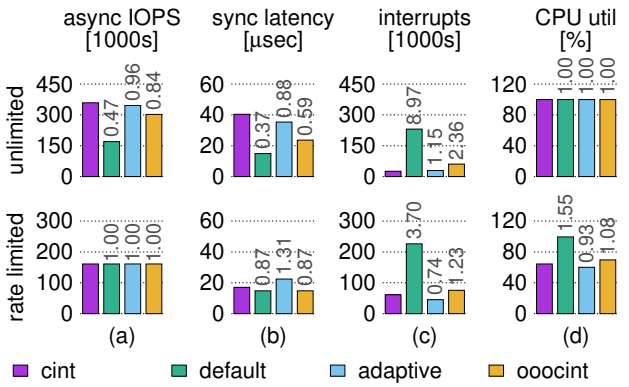


Figure 12: Mixed workload. Out-of-order (OOO) driver processing of Urgent requests enables lower latency at the expense of more interrupts. Limiting the number of async requests (bottom row) reduces this overhead. Labels above bars indicate performance ratio compared to cinterrupts.

throughput in the default scenario (green bar in the first row). In this case, OOO cinterrupts only generates 23% more interrupts, and its synchronous latency matches that of the default strategy while using 30% less CPU.

OOO processing is turned on by default in the cinterrupts NVMe driver but can be disabled with a module parameter.

4 Implementation

4.1 Software Modifications

4.1.1 Kernel Modifications

It is software's responsibility to pass request annotations to the device. To minimize programmer burden, our implementation includes a modified kernel that sets default annotations. Table 1 summarizes how the kernel sets these defaults, which naturally derive from the programming paradigm of each system call: any system call that blocks the application, such as read and write, is marked Urgent, and any system call that supports asynchronous submission, such as io_submit, is marked Barrier. System calls in the sync family are blocking, so they are marked Urgent. By following the programming paradigm of each system call, cinterrupts can be supported in the kernel with limited intrusion; the changes to the Linux software stack described in this section total around 100 LOC.

The cinterrupts kernel propagates annotations from the system call layer through the filesystem and block layers to the device. In the system call handling layer, cinterrupts embeds any bits in the iocb struct. The block layer can split or merge requests. In the case of request split – for example, a 1M write will get split into several smaller write blocks – each child request will retain the bit of the parent. In the case of merged requests, the merged request will retain a superset of the bits in its children. If this superset contains both Urgent and Barrier, we mark the merged request as Urgent for simplicity. This is not a correctness issue because the underlying adaptive algorithm will ensure that no request gets stuck.

System call	Kernel default annotations
(p)read(v), (p)write(v)	Urgent if fd is blocking or if write is `O_DIRECT`
preadv2, pwritev2	If `RWF_NOWAIT` is not set, use Urgent
io_submit	Barrier on the last request
(f)(data)sync, syncfs	Urgent
msync	With `MS_SYNC`, Barrier on the last request

Table 1: Summary of storage I/O system calls and the corresponding default bits used by the kernel.

For cases in which these defaults do not match application-level semantics, we expose a system call interface for applications to override these defaults. We leverage the preadv2/pwritev2 system call interface [63], which already exposes a parameter that accepts flags:

```
ssize_t preadv2(int fd, const struct iovec *iov,
                int iovcnt, off_t offset, int flags)
```

We create two new flag types, RWF_URGENT and RWF_BARRIER, which the application can use to pass bits as it sees fit. The application can explicitly ask for a request to be unmarked by passing both flags. We explain how applications can use this interface in the next section.

4.1.2 Application Case Studies

Ultimately the application has the best knowledge of when it requires interrupts, so cinterrupts enables the application to override kernel defaults for even better performance, using the syscall interface described previously. We modified RocksDB to use these flags.

RocksDB Background Tasks. Flushing and compaction are the two main sources of background I/O in RocksDB. We modify RocksDB to explicitly mark these I/O requests as non-Urgent. Since RocksDB already isolates the environment for interacting with files, our changes were minimal and involved replacing the calls to pread/pwrite with preadv2/pwritev2, creating a new file option to express urgency of I/O for that file, and modifying the Flush and Compaction jobs to explicitly mark I/O as non-Urgent, which totaled around 40 lines of code. We show in Section 5.4.1 that these manual annotations especially help RocksDB during write-intensive workloads.

RocksDB Dump. RocksDB includes a dump tool that dumps all the keys in a RocksDB instance to a file [1]. Typically this tool is used to debug or migrate a RocksDB instance. As it is a maintenance-level tool, dump requests do not need to be Urgent, so we manually modify the dump code to mark dump read and write I/O as non-Urgent. In this way, dump I/O requests are completed when interrupts are generated by the underlying adaptive coalescing strategy. On top of the RocksDB changes described in the previous section, marking dump requests as non-Urgent only required two lines of code. We show in Section 5.5.1 that modifying the dump tool can increase the throughput of foreground requests by 37%.

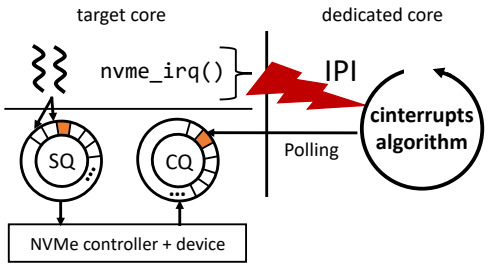


Figure 13: Cinterrupts emulation: the dedicated core polls on the NVMe completion queue of the target core and sends IPIs for any completion. IPIs emulate hardware interrupts of a real device that supports cinterrupts. The target core submits requests normally to hardware by writing to the NVMe submission queue.

4.2 Hardware Modifications

Cinterrupts modifies the hardware-software boundary to support Urgent and Barrier. The key hardware component in cinterrupts is an NVMe device that recognizes these bits and implements Algorithm 2 as its interrupt generation strategy. Device firmware, which is responsible for interrupt generation, is the only device component that must be modified in order to support cinterrupts. Device firmware is typically a blackbox, so we chose to emulate the interrupt generation portion of cinterrupts while leveraging real NVMe hardware for I/O execution.

4.2.1 Firmware Emulation

To emulate interrupt generation in cinterrupts, we explored using several existing aspects of the NVMe specification, all of which were insufficient. We considered using the urgent priority queues to implement Urgent. While this would have worked for Urgent, there is still no way to implement Barrier or Algorithm 2. Furthermore, in NVMe devices, while it is possible to have a dedicated urgent priority queue, hardware queues are still limited in NVMe devices; Azure NVMe devices have 8 queues that must be shared across many cores [26], while Intel devices have 32-128 queues [17, 70]. By labelling individual requests, cinterrupts is explicitly designed to work in a general context where queues cannot be differentiated due to resource limitations.

We also considered using special bogus commands to force the NVMe device to generate an interrupt. The specification recommends that "commands that complete in error are not coalesced" [57]. Unfortunately, neither device we inspected [22, 75] respected this aspect of the specification.

Instead, we prototype cinterrupts by emulating interrupt generation with a dedicated sidecore that uses interprocessor interrupts (IPIs) to emulate hardware interrupts. We implement this emulation on Linux 5.0.8.

Dedicated Core. Our emulation assigns a dedicated core to a target core. The target core functions normally by running applications that submit requests to the core's NVMe submission queue, which are passed normally to the NVMe device. The dedicated core runs a pinned kernel thread, created in the NVMe device driver, that polls the completion queue of the target core and generates IPIs based on Algorithm 2. Cinterrupts annotations are embedded in a request's command ID, which the polling dedicated core inspects to determine which bits are set. In a hardware-level implementation of cinterrupts, Urgent and Barrier can be communicated in any of the reserved bits in the submission queue entry of the NVMe specification [57].

To faithfully emulate the proposed hardware, we disable hardware interrupts for the NVMe queue assigned to that core; in this way, the target core only receives interrupts iff ideal cinterrupts hardware would fire an interrupt. Figure 13 shows how our dedicated core emulates the proposed interrupt generation scheme. Importantly, we still leverage real hardware to execute the I/O requests, and the driver still communicates with the NVMe device through the normal SQ/CQ pairs, but we replace the device's native interrupt generation mechanism with the dedicated core. Section 5.1 shows that this emulation has a modest 3-6% overhead.

4.3 Discussion

Other I/O Requests. We initially did not annotate requests generated by the kernel itself, for example from page cache writeback and filesystem journalling. But because filesystem journalling is on the critical path of write requests, non-Urgent journal transactions caused a slight increase in latency of application-level requests. Hence by default we mark journal commits as Urgent. Because journalling does not generate a large interrupt rate for our applications, marking these requests tightened application latency without adding overhead.

On the other hand, our applications did not see significant benefit in marking writeback requests. As such, we rely on the application to inform us when these requests are latency-sensitive, for example page cache flushes will be Urgent when they are explicitly requested through fsync.

Other Implementations. The Barrier implementation can be strict or relaxed, where a strict version only releases the interrupt if all requests in front of it in the submission queue have been completed. A relaxed Barrier is equivalent to Urgent and works well assuming that requests do not complete too far out of order; it does not require the interrupt generation algorithm to record any additional state. The cinterrupts prototype evaluated in this paper uses a relaxed Barrier, which already enjoys significant performance benefits. We have retained a separate flag because Barrier is semantically different to the application and to enable future implementations to choose to implement strict Barrier.

The strict Barrier requires more accounting overhead to keep track of which requests have completed: we explored a preliminary implementation of the strict Barrier in our emula-

tor but its overheads were larger than its benefit. We suspect firmware implementations of a strict Barrier will be more efficient. Alternatively, this strict ordering could be enforced in the kernel: the driver can withhold completions from userspace until all other requests have completed. Such an implementation might be efficient by piggybacking on the accounting mechanisms in the block layer of the Linux kernel.

Urgent Storm. If all requests in the system are marked as Urgent, this can inadvertently cause an interrupt storm. To address this, cinterrupts has a module parameter that can be configured to target a fixed interrupt rate, similar to NICs, enforced with a lightweight heuristic based on Exponential Weighted Moving Average (EWMA) of the interrupt rate.

Lines of Code. Linux modifications to support cinterrupts total around 100 LOC. The cinterrupts emulator in the NVMe driver is around 500 LOC, with an additional 200 LOC for implementations of strict Barrier and Urgent storm.

5 Evaluation

These questions drive our evaluation: What is the overhead of our cinterrupts emulation (§5.1)? How do device vendors and admins select Δ and thr (§5.2)? How does cinterrupts compare to the default and the adaptive strategies in terms of latency and throughput (§5.3)? How much does cinterrupts improve latency and throughput in a variety of applications (§5.4)?

5.1 Methodology

Experimental Setup. We use two NVMe SSD devices: Intel DC P3700, 400 GB [21] and Intel Optane DC P4800X, 375 GB [23]. We refer to them as P3700 and Optane.

Both SSDs are installed in a Dell PowerEdge R730 machine equipped with two 14-core 2.0 GHz Intel Xeon E5-2660 v4 CPUs and 128 GB of memory running Ubuntu 16.04. The server runs cinterrupts' modified version of Linux 5.0.8 and has C-states, Turbo Boost (dynamic clock rate control), and SMT disabled. We use the maximum performance governor.

Our emulation pairs one dedicated core to one target core. Each core is assigned its own NVMe submission and completion queue. The microbenchmarks are run on a single core, but we run macrobenchmarks on multiple cores. For our microbenchmarks, we use fio [4] version 3.12 to generate workloads. All of our workloads are non-buffered random access. We run the benchmarks for 60 seconds and report averages of 10 runs.

For a consistent evaluation of cinterrupts, we implemented an emulated version of the default strategy. Similar to the emulation of cinterrupts we described in §4.2.1, device interrupts are also emulated with IPIs.

Emulation Overhead. The cinterrupts emulation is lightweight and its overheads come from sending the IPI and cache contention from the dedicated core continuously polling on the CQ memory of the target core. Table 2 summarizes the overhead of emulation. We also show the overhead

	Sync latency of 4 KB, μs			
mitigations	off	default	off	off
system	baremetal interrupts	baremetal interrupts	emulation interrupts	baremetal polling
P3700	$80_{\pm 29.0}$	$81_{\pm 29.1}$	$82_{\pm 28.2}$	$78_{\pm 28.1}$
Optane	$10_{\pm 1.3}$	$11_{\pm 1.3}$	$10_{\pm 1.2}$	$8_{\pm 1.2}$

Table 2: Emulation overhead is comparable with overhead of mitigations. Cinterrupts runs with mitigations disabled to compensate for the emulation overhead.

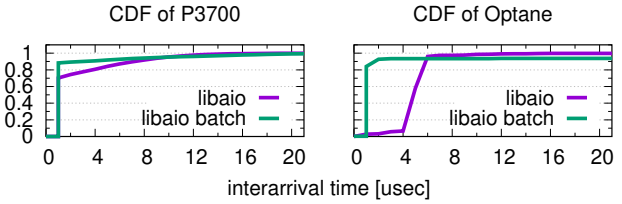


Figure 14: Using interarrival CDF to determine Δ.

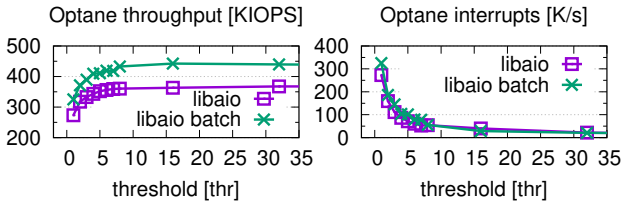


Figure 15: Determining thr under a fixed Δ (Δ=6 μs for Optane and Δ = 15 μs for P3700). thr is the smallest value where throughput plateaus, which is between 16-32, so we set thr= 32 for both devices. We omitted P3700 thr results as it shows virtually the same behavior.

of mitigations for CPU vulnerabilities [31] to show that the overhead of our emulation is comparable to the overhead of the default mitigations for CPU. Therefore, our performance numbers with mitigations disabled and emulation on mirrors results from a server with mitigations enabled and emulation off (cinterrupts implemented in real hardware).

Emulation imposes a modest 3-6% latency overhead for both devices. There is a difference in emulation overhead between the devices, which we suspect is due to each device's time lag between updating the CQ and actually sending an interrupt. As the difference between the last column and the first column shows, this lag varies between devices, and the longer the lag, the smaller the overhead of emulation.

Baselines. We compare cinterrupts to our adaptive strategy and to the default interrupt strategy, which does not coalesce. The adaptive strategy is a proxy for comparison to NVMe coalescing, which it outperforms (Section 3.2).

5.2 Selection of Δ and thr

Δ should approximate the interarrival time of requests, which depends on workload. Figure 14 shows the interarrival time for two types of workloads. The first workload is a single-threaded workload that submits read requests of size 4 KB

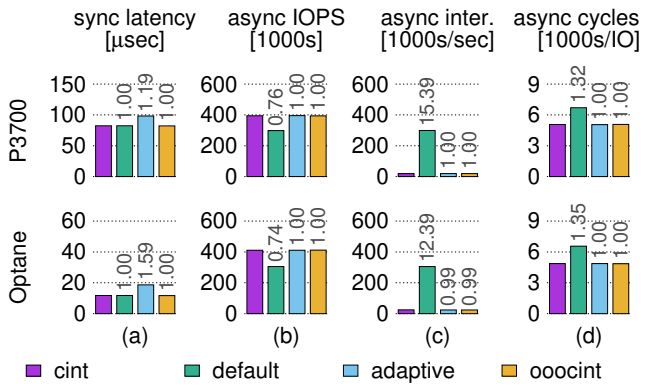


Figure 16: Workloads with only one type of request. Column (a) shows latency of synchronous requests (lower is better); (b), (c) and (d) show metrics for the asynchronous workload.

with libaio and iodepth=256. The second workload is the same workload, except with batched requests. We run the same workloads on P3700 and Optane, to show that vendors or sysadmin will pick different Δ for different devices.

When libaio submits batches, the CPU can send many more requests to the device, resulting in lower interarrival times – a 90th percentile of 1 μs in the batched case versus 6 μs in the non-batched case for Optane. For P3700, both workloads have a 99th percentile of 15 μs. We pick Δ to minimize the interrupt rate without adding unnecessary delay, so for P3700 we set Δ=15 μs and for Optane we set Δ=6 μs.

After fixing Δ, we sweep thr in the $[0, 256)$ range and select the lowest thr after which throughput plateaus; the results are shown in Figure 15. thr= 32 achieves high throughput and low interrupt rate for both devices.

In practice, hardware vendors should use this methodology to set default values to Δ and thr for their devices, and system administrators could tune these values for their workloads.

5.3 Microbenchmarks

We use fio to generate two workloads to show how cinterrupts behaves at the extremes. The synchronous-only workload submits blocking 4 KB reads via read. The asynchronous-only workload submits 4 KB reads via libaio with iodepth 256 and batches of size 16. For each device, cinterrupts and the adaptive strategy are configured with the same Δ and thr. The results are shown in Figure 16.

As in §3.3, the synchronous workload shows the drawback of the adaptive strategy, which adds exactly Δ=15 μs to request latency for P3700 and Δ=6 μs for Optane (first column of Figure 16). Cinterrupts remedies this with Urgent. The default strategy performs as well as cinterrupts in the synchronous workload, because it generates an interrupt for every request. This strategy is penalized in the asynchronous workload, where the default strategy generates 12-15x the number of interrupts as cinterrupts.

interrupt scheme	thruput [KIOPS]	norm	avg lat [ms]	norm	p99 lat [ms]	norm
cint	$388_{\pm 5.6}$	1.00	$1.3_{\pm 0.3}$	1.00	$15.0_{\pm 0.8}$	1.00
default	$391_{\pm 1.8}$	1.01	$1.3_{\pm 0.1}$	1.00	$14.4_{\pm 0.4}$	0.96
adaptive	$391_{\pm 4.6}$	1.01	$1.3_{\pm 0.4}$	1.00	$14.2_{\pm 0.9}$	0.95
app-cint	$405_{\pm 5.6}$	1.04	$1.3_{\pm 0.1}$	1.00	$12.7_{\pm 0.3}$	0.85

Table 3: Modifying RocksDB with annotations that make the flush non-urgent (app-cint). Results are for database load (fillbatch experiment in db_bench). Note that cint and default have the same performance, within error bounds, which is expected for RocksDB.

Cinterrupts matches the synchronous latency of default, while achieving up to 35% more asynchronous throughput, and matches the asynchronous throughput of adaptive while achieving up to 37% lower latency. Finally, OOO does not add overhead to cinterrupts performance when it is not triggered.

5.4 Macrobenchmarks

To evaluate the effect of cinterrupts on real applications, we run three application setups on Optane: RocksDB [64], KVell [48], and RocksDB and KVell colocated on the same cores. RocksDB is a widely used key-value store that uses pread/pwrite system calls in its storage engine, and KVell is a new key-value store employing Linux AIO in its storage engine. Both applications use direct I/O. KVell uses default kernel annotations (Barrier) while we will note when RocksDB uses default annotations or the modified annotations described in Section 4.1.2.

We run each application on two cores. In KVell, an additional four cores are allocated for clients. Cinterrupts is the only strategy that performs the best across all three setups.

5.4.1 RocksDB

Load. Using db_bench [8], we load a database with 10 M key-value pairs, with 16 byte keys and 1 KiB values. During the load phase, we compare the results under cinterrupts where RocksDB is unmodified and modified. In unmodified RocksDB, every I/O is labelled Urgent by default. In modified RocksDB, background activity is non-Urgent as described in Section 4.1.2. Table 3 shows the performance results.

We see that marking background activity as non-Urgent has a modest but significant 4% increase in throughput without affecting latency (app-cint vs cint). This is because delaying the interrupts of background I/O does not affect foreground latency. In fact, doing so actually decreases the tail latency of foreground writes by 15%. Hence reducing the CPU pressure caused by interrupts enables better p99 latency.

Steady State. After loading the database to 20GB, we run two experiments from db_bench: readrandom, where each thread reads randomly from the key space, and readwhilewriting, where one thread inserts keys and the rest of the threads read randomly. The readwhilewriting experiment runs for 30 seconds. For each experiment, we also vary the number of

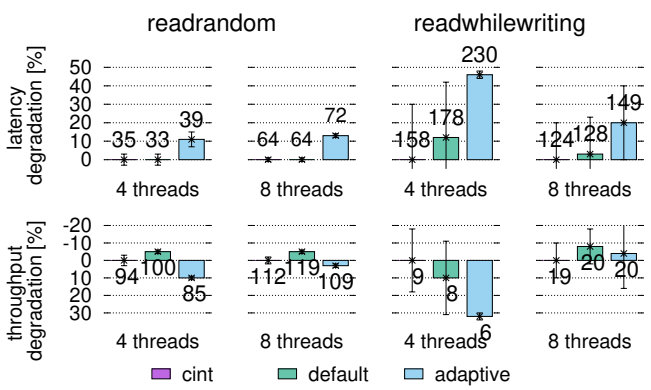


Figure 17: Latency of get operation and throughput in RocksDB for varying workloads. We show performance degradation with respect to cinterrupts. Labels show absolute values in μs and KIOPS, respectively. As expected, cinterrupts and the default strategy have nearly the same performance within error bounds, but the adaptive strategy has up to 45% worse latency and 5-32% worse throughput due to the Δ delay.

Workload	Description
A	update heavy: 50% reads, 50% writes
B	read mostly: 95% reads, 5% writes
C	read only: 100% reads
F	read latest: 95% reads, 5% updates
D	read-modify-write: 50% reads, 50% r-m-w
E	scan mostly: 95% scans, 5% updates

Table 4: Summary of YCSB workloads.

threads. The latency of the get operation and throughput for both experiments is shown in Figure 17.

As expected, for both metrics, cinterrupts and the default strategy perform nearly the same because both generate interrupts for every request; in the next two applications, the default strategy will suffer due to this behavior. On the other hand, adaptive does consistently worse because of its Δ delay; this is particularly noticeable in the latency measurements. With 8 threads, this delay penalty is amortized across more threads, which reduces the performance degradation.

Interestingly, modified RocksDB had similar performance to unmodified RocksDB during these benchmarks. This is because there is very little if any background I/O in the readrandom benchmark, and the write rate is not high enough for the background I/O interrupts to affect foreground performance in the readwhilewriting benchmark.

5.4.2 KVell

We use workloads derived from the YCSB benchmark [14], summarized in Table 4. We load 80 M key-value pairs, with 24 byte keys and 1 KB item sizes for a dataset of size 80 GB. Each workload does 20M operations. Figure 18 shows KVell throughput, average latency, and 99th percentile latency for each YCSB workload.

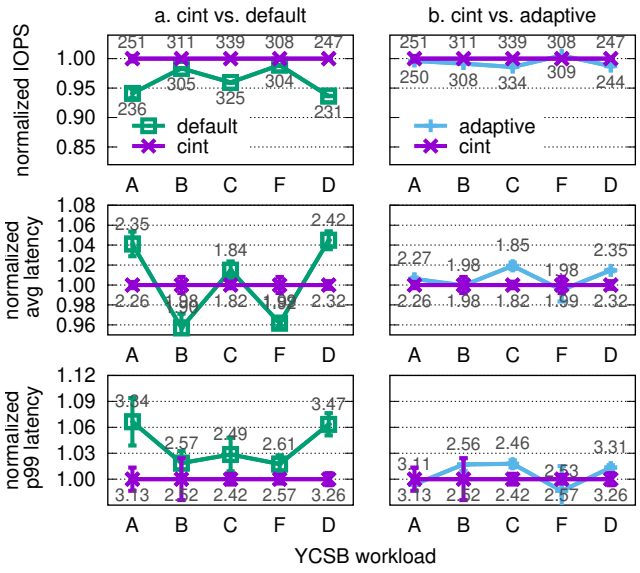


Figure 18: Throughput and latency results for YCSB on KVell. Labels show absolute throughput in KIOPS and latency in ms.

Throughput. Cinterrupts does better than default for throughput, because default generates an interrupt for *every* request. In contrast, cinterrupts uses Barrier to generate an interrupt for a single batch, which consists of 10s of requests. The difference between cinterrupts and default is more pronounced for write-heavy workloads (A, D), but less pronounced for read-heavy workloads (B, C, F) . This is because reads are efficient in KVell, so there is some CPU idleness in these workloads (3% idleness under default and 14% idleness under cinterrupts).

The adaptive strategy performs similarly to cinterrupts because it is designed to detect bursts. Its delay is more pronounced in latency measurements.

Latency. The adaptive strategy has 5-8% higher average and 99th percentile latency than cinterrupts in all workloads. Again, this is the effect of the Δ delay, which cinterrupts remedies with Barrier. Cinterrupts latency also does better than the default, where interrupt handling and context switching both add to the latency of requests and slow down the request submission rate. The high number of interrupts in the default strategy also adds to latency variability, which is noticeable in the larger 99th percentile latencies.

YCSB-E. Scans are interesting because their latency is determined by the completion of requests that can span multiple submission boundaries. Table 5 shows throughput results for YCSB-E with different scan lengths, and Figure 19 shows latency CDFs for scans of length 16 and 256.

Similar to the other YCSB workloads, the adaptive strategy again can almost match the throughput of cinterrupts, because it is designed for batching. At higher scan lengths, factors such as application-level queueing begin affecting scan throughput, reducing the benefit of cinterrupts.

	length=16		length=256	
interrupt scheme	scans [KIOPS]	normalized	scans [KIOPS]	normalized
cint	$26.2_{\pm 0.5}$	1.00	$1.6_{\pm 0.04}$	1.00
default	$23.1_{\pm 0.3}$	0.88	$1.5_{\pm 0.06}$	0.95
adaptive	$24.9_{\pm 0.6}$	0.95	$1.6_{\pm 0.02}$	0.97

Table 5: YCSB-E throughput results for KVell. Excessive interrupt generation limits default throughput to 86%-89% of cinterrupts'.

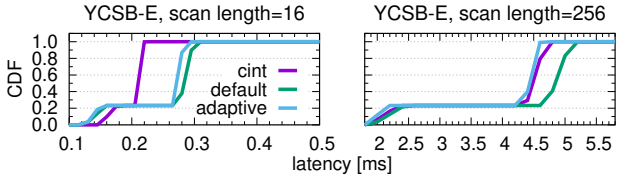


Figure 19: Latency CDF of scans of length 16 and 256 in KVell.

Figure 19 shows that there is a notable difference in scan latency between cinterrupts and the default for both scan lengths; the difference in 50th percentile latencies between default and cinterrupts is around 600 μs for both scan lengths. This difference is maintained at the 99th percentile latencies.

Notably, there is a 400μs difference between cinterrupts and adaptive 50th percentile latencies when the scan length is 16, which goes away when the scan length is 256. The adaptive strategy does well in KVell's asynchronous programming model and longer scans are able to amortize the additional delay over many requests.

5.5 Colocated Applications

We run two types of colocated applications to see the effects of cinterrupts in consolidated datacenter environments.

5.5.1 RocksDB + Dump Tool

First we run two colocated instances that both use pread/pwrite, which means by default the kernel marks all I/O as Urgent. The first is a regular RocksDB instance, and the second is a RocksDB instance running the RocksDB dump tool. As described in Section 4.1.2, we modify the RocksDB dump tool to explicitly disable the Urgent bit on its I/O requests.

We load two databases with 10 M key-value pairs, as in the previous section. Then, one database runs readrandom, as in the previous section, while we run the dump tool on the second database. We compare the performance of get requests under modified and unmodified RocksDB in Table 6. app-cint shows the results when the dump tool is modified.

By disabling Urgent in the dump tool, we increase the throughput of get requests by 37%, decrease the average latency by 32%, and decrease the 99th percentile latency by 86% compared to the kernel annotations cinterrupts. This is not only from the reduced interrupt rate generated by the dump tool, but also from the reduced I/O bandwidth generated by the dump tool. On the other hand, the throughput of the dump tool decreases by 11% under app-cint, but this is an acceptable trade-off for the foreground improvements.

interrupt scheme	thruput [KIOPS]	norm	get lat [ms]	norm	p99 lat [ms]	norm
cint	$24.8_{\pm 3.9}$	1.00	$30.4_{\pm 0.4}$	1.00	$421_{\pm 41}$	1.00
default	$24.9_{\pm 1.6}$	1.00	$30.9_{\pm 0.3}$	1.02	$420_{\pm 25}$	1.00
adaptive	$20.2_{\pm 0.7}$	1.02	$43.9_{\pm 0.5}$	1.44	$85.1_{\pm 7.0}$	0.15
app-cint	$33.1_{\pm 1.2}$	1.37	$20.7_{\pm 0.4}$	0.68	75.7 ± 0.1	0.14

Table 6: Performance of RocksDB readrandom while the RocksDB dump tool is running in the background. app-cint modifies the dump tool to mark its I/O requests as non-urgent, boosting throughput and latency of the foreground get operations. Interestingly, the adaptive strategy can tame the tail latency (p99 lat) of get requests, but does so at the expense of limited IOPS.

5.5.2 RocksDB + KVell

Finally, we run RocksDB and KVell on the same cores. RocksDB runs the readrandom benchmark from before, and KVell runs YCSB-C. We run two experiments, varying the number of threads of the RocksDB instance. The latency of RocksDB requests and the throughput of KVell is shown in Tables 7 and 8.

When there are four RocksDB threads, the default strategy matches the RocksDB latency of cinterrupts, but has 12% less KVell throughput due to the excessive interrupt rate. Conversely, the adaptive strategy can match the KVell throughput of cinterrupts, but has 11% worse RocksDB latency.

As before, when there are more RocksDB threads, the effect of cinterrupts is less pronounced, because the CPU spends less of its time handling interrupts and more of its time context-switching and in userspace. Even so, cinterrupts still achieves a modest 5-6% higher throughput and up to 6% better latency than the other two strategies.

6 Cinterrupts for Networking

Figure 3 (in §2) shows that NICs suffer from similar problems as NVMe drives with respect to interrupts. A natural future direction is applying cinterrupts to the networking stack. Accomplishing this goal, however, is more challenging, as explained next in the context of Ethernet.

Cooperation. For network cinterrupts to work, modifying a single host (as in storage) is insufficient. Multiple communicating parties should be changed to agree on how cinterrupts semantics are communicated. In particular, transmitters should be modified to send network packets that indicate whether to fire an interrupt immediately upon reaching their destination, and receivers should be modified to react accordingly. Any interrupt-driven software routers along the way should also preferably support cinterrupts.

Propagation. NVMe controllers deal with plaintext read or write requests associated with pointers to buffers; the con-

interrupt scheme	RocksDB get lat [μs]	normalized	KVell [KIOPS]	normalized
cint	$116_{\pm 0.8}$	1.00	$171_{\pm 2.8}$	1.00
default	$115_{\pm 0.8}$	0.99	$153_{\pm 2.0}$	0.89
adaptive	$129_{\pm 0.0}$	1.11	$171_{\pm 2.0}$	1.00

Table 7: Results from colocated experiment: 4 RocksDB threads and KVell. As expected, cinterrupts both has lower latency than the adaptive strategy and higher throughput than the baseline.

interrupt scheme	RocksDB get lat [μs]	normalized	KVell [KIOPS]	normalized
cint	$164_{\pm 0}$	1.00	$131_{\pm 1}$	1.00
default	$163_{\pm 0}$	0.99	$123_{\pm 0}$	0.94
adaptive	$174_{\pm 1}$	1.06	$124_{\pm 0}$	0.95

Table 8: Results from colocated experiment: 8 RocksDB threads and KVell. The performance gains of cinterrupts is reduced with respect to Table 7, because the CPU is both context-switching more and spending more time in userspace.

troller either copies bytes from the buffers to the drive or vice versa. This is true even when NVMe is encapsulated in other, higher-level protocols, as is the case with, e.g., NVMe over TCP over TLS encryption (denoted NVMeTLS [62]).

In contrast, network stacks are layered. NICs may operate at the Ethernet level, but the content of Ethernet frames frequently encapsulates higher-level protocols, like IP, TCP, UDP, and VXLAN. Crucially, the transmitted payload, which includes headers of higher-level protocols, is oftentimes encrypted. For example, the payload in tunnel-mode IPsec packets [36] encapsulates encrypted IP and TCP headers. Bits in these headers are thus unsuitable for communicating cinterrupts information, as the receiving NIC might not be able to observe them. Consequently, to support cinterrupts, each layer at the sender should be modified to explicitly propagate cinterrupts information to its encapsulating layer, until a low-enough protocol level is reached.

Within a data center, it seems reasonable to choose Ethernet as the aforementioned low-enough protocol. In practice, however, there are no free Ethernet reserved bits or flags that can be used for this purpose [68]. Cinterrupts bits can instead reside one level higher, at the least-encapsulated IP layer, as its headers are not encrypted, and its "options" field [44] can be used to add the extra bits. The downside is that other, non-IP networks—such as RDMA over Converged Ethernet (RoCE) [72] and Fiber Channel over Ethernet (FCoE) [32]—should be handled separately, in some other way.

Segmentation. TCP performance is accelerated by NIC offloading functionality, which significantly reduces CPU processing overhead. Notably, upon transmit, software may use TSO (TCP segmentation offload) to hand a sizable ($\leq$64KB) TCP segment to the NIC, relying on the NIC to split the outgoing segment into a sequence of ($\leq$MTU) Ethernet frames [51]. Likewise, with LRO (large receive offload), the NIC may reassemble multiple incoming frames into a single sizable segment before handing it to software [51, 52].

Storage cinterrupts affect only the timing and number of device interrupts. Network cinterrupts can also increase the number of I/O requests and thus the CPU usage, assuming TSO and LRO are not applied beyond cinterrupts. To illustrate, assume $\{M_i\}_{i=0}^{15}$ is a consecutive series of 1KB messages, each individually associated with a cinterrupt. To optimize latency, differently than what frequently happens on existing systems, $\{M_i\}_{i=0}^{15}$ should seemingly *not* be aggregated into a single 16KB TCP segment at the sender before it is handed to the NIC (leveraging TSO), nor should it be aggregated to a single segment by the receiver NIC (leveraging LRO); otherwise only the cinterrupt of M_{15} will survive. But such a policy might inadvertently degrade both latency and throughput if the CPUs of the sender or receiver are saturated, necessitating a more sophisticated policy that considers CPU usage.

URG and PSH. The TCP flags URG and PSH seem related to cinterrupts. But even if ignoring the aforementioned propagation problem, in practice, the semantics of these flags are sufficiently different that they cannot be repurposed for cinterrupts. Specifically, URG is used to implement socket out-of-band communication [27, 29, 54], and its usage model involves the POSIX SIGURG signal. (URG also has security implications [28, 29, 80], and middleboxes and firewalls tend to clear it by default [12, 59].) When examining how PSH is used in the Linux network stack (see calls to the tcp_push and tcp_mark_push functions in the source code), we find that it is used in many more circumstances than is appropriate for cinterrupts. For example, sending a 16KB message using a single write system call frequently results in four Ethernet frames encapsulating PSH segments, instead of one.

7 Conclusion

In this paper we show that the existing NVMe interrupt coalescing API poses a serious limitation on practical coalescing. In addition to devising an adaptive coalescing strategy for NVMe, our main insight is that software directives are the best way for a device to generate interrupts. Cinterrupts, with a combination of Urgent, Barrier, and the adaptive burst-detection strategy, generates interrupts exactly when a workload needs them, enabling workloads to experience better performance even in a dynamic environment. In doing so, cinterrupts enables the software stack to take full advantage of existing and future low-latency storage devices.

8 Acknowledgements

We are deeply indebted to our shepherd, Orran Krieger, for helping shape the final version of this paper. We are also grateful to the anonymous reviewers for their comments that have greatly improved this paper. We also thank Marcos Aguilera, Nadav Amit, and Jon Howell for discussions and comments on earlier iterations of this work.

References

[1] Administration and data access tool. https://github.com/facebook/rocksdb/wiki/Administration-and-Data-Access-Tool. Accessed: May, 2021.

[2] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[4] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio. Accessed: May, 2021.

[5] Jens Axboe. Linux kernel mailing list, blk-mq: make the polling code adaptive. https://lkml.org/lkml/2016/11/3/548, 2016. Accessed: May, 2021.

[6] Pavel Begunkov. Linux kernel mailing list, blk-mq: Adjust hybrid poll sleep time. https://lkml.org/lkml/2019/4/30/120, 2019. Accessed: May, 2021.

[7] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[8] Benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed: May, 2021.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: introducing multi-queue ssd access on multi-core systems. In *International Systems and Storage Conference (SYSTOR)*, 2013.

[10] Block IO controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt. Accessed: May, 2021.

[11] Keith Bush. Linux nvme mailing list: nvme pci interrupt handling improvements. https://lore.kernel.org/linux-nvme/20191209175622.1964-1-kbusch@kernel.org/, 2019. Accessed: May, 2021.

[12] Cisco Systems, Inc. Cisco ASA series command reference: urgent-flag. https://www.cisco.com/c/en/us/td/docs/security/asa/asa-cli-reference/T-Z/asa-command-ref-T-Z/u-commands.html#wp2606000884, 2021. Accessed: May, 2021.

[13] 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 *USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM symposium on Cloud computing (SoCC)*, 2010.

[15] Jonathan Corbet. Batch processing of network packets. https://lwn.net/Articles/763056/. Accessed: May, 2021.

[16] Jonathan Corbet. Driver porting: Network drivers. https://lwn.net/Articles/30107/. Accessed: May, 2021.

[17] Intel Corporation. Intel Optane SSD DC D4800X Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-d4800x-product-brief.pdf. Accessed: May, 2021.

[18] Intel Corporation. Intel Optane technology for data centers. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html. Accessed: May, 2021.

[19] Intel Corporation. Intel data direct I/O technology (Intel DDIO): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012. Accessed: May, 2021.

[20] Intel Corporation. Intel Ethernet Converged Network Adapter XL710. https://ark.intel.com/content/www/us/en/ark/products/83967/intel-ethernet-converged-network-adapter-xl710-qda2.html, 2014. Accessed: May, 2021.

[21] Intel Corporation. Intel SSD DC P3700 Series. https://ark.intel.com/content/www/us/en/ark/products/79624/intel-ssd-dc-p3700-series-400gb-1-2-height-pcie-3-0-20nm-mlc.html, 2014. Accessed: May, 2021.

[22] Intel Corporation. Intel Optane SSD 900P Series. https://ark.intel.com/content/www/us/en/ark/products/123623/intel-optane-ssd-900p-series-280gb-2-5in-pcie-x4-20nm-3d-xpoint.html, 2017. Accessed: May, 2021.

[23] Intel Corporation. Intel Optane SSD DC P4800X Series. https://ark.intel.com/content/www/us/en/ark/products/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint.html, 2017. Accessed: May, 2021.

[24] Intel Corporation. Intel Optane SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201861/intel-optane-ssd-dc-p5800x-series-400gb-2-5in-pcie-x4-3d-xpoint.html, 2018. Accessed: May, 2021.

[25] Intel Corporation. Intel SSD DC P4618 Series. https://ark.intel.com/content/www/us/en/ark/products/192574/intel-ssd-dc-p4618-series-6-4tb-1-2-height-pcie-3-1-x8-3d2-tlc.html, 2019. Accessed: May, 2021.

[26] Microsoft Corporation. Microsoft Documentation: Optimize performance on the Lsv2-series virtual machines. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/storage-performance, 2019. Accessed: May, 2021.

[27] Kevin R. Fall and W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 2011.

[28] Fernando Gont. Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations. Technical report, Internet Engineering Task Force, March 2012. Work in Progress.

[29] Fernando Gont and Andrew Yourtchenko. On the Implementation of the TCP Urgent Mechanism. RFC 768, Internet Engineering Task Force, 2011.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2017.

[31] Hardware vulnerabilities, The Linux kernel user's and administrator's guide. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/index.html. Accessed: May, 2021.

[32] John Hufferd. Fibre Channel over Ethernet (FCoE). https://www.snia.org/educational-library/fibre-channel-over-ethernet-fcoe-2013-2013, 2013. Accessed: May, 2021.

[33] Intel. DPDK: Data plane development kit. https://www.dpdk.org, 2014. Accessed: May, 2021.

[34] Rick A. Jones. Netperf: A network performance benchmark. https://github.com/HewlettPackard/netperf, 1995. Accessed: May, 2021.

[35] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Third ACM Symposium on Cloud Computing (SoCC)*, 2012.

[36] S. A. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Internet Engineering Task Force, November 1998.

[37] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing array of SSDs when the storage device is no longer the performance bottleneck. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[38] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[39] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O path: a holistic approach for application performance. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[40] Avi Kivity. Wasted processing time due to nvme interrupts. https://github.com/scylladb/seastar/issues/507, 2018. Accessed: May, 2021.

[41] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency SSDs. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[42] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[43] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[44] Charles M. Kozierok. The TCP/IP guide. http://www.tcpipguide.com/free/t_IPDatagramOptionsandOptionFormat.htm. Accessed: May, 2021.

[45] Damien Le Moal. I/O latency optimization with polling. *Linux Storage and Filesystems Conference (VAULT)*, 2017.

[46] 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 (USENIX ATC)*, 2019.

[47] Ming Lei. Linux-nvme mailing list: nvme-pci: check CQ after batch submission for Microsoft device. https://lore.kernel.org/linux-nvme/20191114025917.24634-3-ming.lei@redhat.com/, 2019. Accessed: May, 2021.

[48] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[49] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems (EuroSys)*, 2014.

[50] Long Li. Linux kernel mailing list: fix interrupt swamp in NVMe. https://lkml.org/lkml/2019/8/20/45, 2019. Accessed: May, 2021.

[51] Linux kernel documentation. Segmentation offloads. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html, 2021. Accessed: May, 2021.

[52] Mellanox Technologies. How to enable large receive offload (LRO). https://community.mellanox.com/s/article/how-to-enable-large-receive-offload--lro-x, 2020. Accessed: May, 2021.

[53] Merriam-Webster. "Calibrate". https://www.merriam-webster.com/dictionary/calibrate, 2020. Accessed: May, 2021.

[54] Microsoft Corporation. OOB Data in TCP. https://docs.microsoft.com/en-us/windows/win32/winsock/protocol-independent-out-of-band-data-2#oob-data-in-tcp, 2018. Accessed: May, 2021.

[55] Jeffrey C. Mogul and Kadangode K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX Annual Technical Conference (ATEC)*, 1996.

[56] Rikin J. Nayak and Jaiminkumar B. Chavda. Comparison of accelerator coherency port (ACP) and high performance port (HP) for data transfer in DDR memory using Xilinx ZYNQ SoC. In *International Conference on Information and Communication Technology for Intelligent Systems (ICTIS)*, 2017.

[57] NVM Express, Revision 1.3. https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf. Accessed: May, 2021.

[58] NVM Express, Revision 1.4, Figure 284. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf. Accessed: May, 2021.

[59] Palo Alto Networks. How to preserve the TCP URG flag and pointer. https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000ClWACA0, 2018. Accessed: May, 2021.

[60] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[61] 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. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[62] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC offloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[63] preadv2(2) — Linux manual page. https://man7.org/linux/man-pages/man2/preadv2.2.html. Accessed: May, 2021.

[64] RocksDB. https://github.com/facebook/rocksdb. Accessed: May, 2021.

[65] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O path optimizations for flash solid-state drives. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[66] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[67] SPDK: Storage performance development kit. https://spdk.io/. Accessed: May, 2021.

[68] Charles E. Spurgeon and Joann Zimmerman. Ethernet: The definitive guide, 2nd edition. https://www.oreilly.com/library/view/ethernet-the-definitive/9781449362980/ch04.html. Accessed: May, 2021.

[69] Steven Swanson and Adrian M. Caulfield. Refactor, reduce, recycle: Restructuring the IO stack for the future of storage. *Computer*, 2013.

[70] Billy Tallis. Intel Optane SSD DC P4800X 750GB hands-on review. https://www.anandtech.com/show/11930/intel-optane-ssd-dc-p4800x-750gb-handson-review/3, 2017. Accessed: May, 2021.

[71] Mellanox Technologies. Mellanox ConnectX-5 VPI Adapter. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf, 2018. Accessed: May, 2021.

[72] The RoCE Initiative. RoCE is RDMA over Converged Ethernet. https://www.roceinitiative.org. Accessed: May, 2021.

[73] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS)*, 2007.

[74] John E. Uffenbeck. *The 80x86 family: design, programming, and interfacing*. Prentice Hall PTR, 1997.

[75] Western Digital Corporation. Ultrastar DC SN200. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-nvme-series/data-sheet-ultrastar-dc-sn200.pdf. Accessed: May, 2021.

[76] 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 dtabases. In *ACM International Systems and Storage Conference (SYSTOR)*, 2015.

[77] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[78] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[79] Tom Yates. Improvements to the block layer. https://lwn.net/Articles/735275/. Accessed: May, 2021.

[80] Young Yoon, Jae Yong Oh, and Young Min Yoon. NIDS evasion method named "SeolMa". *Phrack Magazine, Volume 0x0b, Issue 0x39*, 2001.

[81] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[82] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[83] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *USENIX Conference on File and Storage Technologies (FAST)*, 2020.

ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction

Kyuhwa Han[1,2], Hyunho Gwak[1], Dongkun Shin[1*], and Joo-Young Hwang[2]

[1]Sungkyunkwan University, [2]Samsung Electronics

Abstract

The NVMe zoned namespace (ZNS) is emerging as a new storage interface, where the logical address space is divided into fixed-sized zones, and each zone must be written sequentially for flash-memory-friendly access. Owing to the sequential write-only zone scheme of the ZNS, the log-structured file system (LFS) is required to access ZNS solid-state drives (SSDs). Although SSDs can be simplified under the current ZNS interface, its counterpart LFS must bear segment compaction overhead. To resolve the problem, we propose a new LFS-aware ZNS interface, called ZNS+, and its implementation, where the host can offload data copy operations to the SSD to accelerate segment compaction. The ZNS+ also allows each zone to be overwritten with sparse sequential write requests, which enables the LFS to use threaded logging-based block reclamation instead of segment compaction. We also propose two file system techniques for ZNS+-aware LFS. The copyback-aware block allocation considers different copy costs at different copy paths within the SSD. The hybrid segment recycling chooses a proper block reclaiming policy between segment compaction and threaded logging based on their costs. We implemented the ZNS+ SSD at an SSD emulator and a real SSD. The file system performance of the proposed ZNS+ storage system was 1.33–2.91 times better than that of the normal ZNS-based storage system.

1 Introduction

In the NVMe zoned namespace (ZNS) [9] interface, the logical address space is divided into fixed-sized zones. Each zone must be written sequentially and reset explicitly for reuse. The ZNS SSD has several benefits over legacy SSDs. First, performance isolation between different IO streams can be provided by allocating separate zones to each IO stream, which is useful for multi-tenant systems. Second, if the zone size becomes a multiple of the flash erase block size (64–1,024 KB), the ZNS SSD can maintain a zone-level logical-to-physical address mapping (i.e., mapping between zone and flash blocks) because each zone is sequentially written. The coarse-grained mapping requires a small internal DRAM of SSD. Compared with legacy SSDs, which require a large DRAM equivalent to 0.1% of storage capacity (e.g., 1 GB DRAM for 1 TB SSD) for a fine-grained mapping, the DRAM usage of the ZNS SSD is significantly reduced. In particular, because the mapped flash blocks of a zone will be fully invalidated at zone reset, the SSD-internal garbage collection (GC) is not required, and thus, the log-on-log [34] problem can be solved by the *GC-less* SSD. The over-provisioned space for GC is not necessary anymore, and the unpredictable long delays by GC can be avoided. The write amplification by GC can also be eliminated, which will allow triple-level cell (TLC) or quad-level cell (QLC) SSDs with low endurance to proliferate.

IO Stack for ZNS. Generally, new storage interfaces require revamping the software stack. For the ZNS, we need to revise two major IO stack components, file system and IO scheduler. First, the in-place updating file systems such as EXT4 must be replaced with append logging file systems such as the log-structured file system (LFS) to eliminate random updates. Because a segment of LFS is written sequentially by append logging, each segment can be mapped to one or more zones. Second, the IO scheduler must guarantee the in-order write request delivery for a zone. For example, an in-order queue for each zone can be used, and the scheduler only needs to determine the order of services between different zones.

Increased Host Overhead. Under the append logging scheme of LFS, the obsolete blocks of a dirty segment must be reclaimed by segment compaction (also called segment cleaning or garbage collection), which moves all valid data in the segment to other segments to make the segment clean. The compaction invokes a large number of copy operations, especially when the file system utilization is high. The *host-side GC* must be performed in exchange for using GC-less ZNS SSD, although the duplicate GCs by the log-on-log situation can be avoided. The overhead of host-side GC is higher than that of device-side GC because the host-level block copy requires IO request handling, host-to-device data transfer, and page allocation for read data [20]. In addition, segment com-

*Corresponding author

paction needs to modify file system metadata to reflect data relocation. Moreover, the data copy operations for a segment compaction are performed in a batch, and thus, the average waiting time of many pending write requests is significant. According to the experiments of F2FS [19] — one of the widely used log-structured file systems, the performance loss by segment compaction is about 20% when the file system utilization is 90%. Therefore, it can be said that the current ZNS focuses on the SSD-side benefit without considering the increased complexity of the host. To simplify the design of SSD, all the complicated things are passed to the host. (Nevertheless, the host can benefit from the ZNS, in terms of performance isolation and predictability.)

In addition, ZNS storage systems will involve *diminishing returns* as we increase the bandwidth of SSD by embedding more flash chips. The zone size of a ZNS device will be determined to be large enough to utilize the SSD-internal flash chip parallelism. Therefore, a higher bandwidth of ZNS SSD will provide a larger zone size, and the file system must use a larger segment size accordingly. Then, the host suffers from segment compaction overhead more seriously because the overhead generally increases in proportion to the segment size [25]. To improve IO performance and overcome diminishing returns, a host-device co-design is required, which places each subtask of segment compaction in the most appropriate location without harming the benefit of the original ZNS, instead of simply moving the GC overhead from the SSD to the host.

LFS-aware ZNS. We need some device-level support to alleviate the segment compaction overhead of LFS. Two approaches can be considered: *compaction acceleration* and *compaction avoidance*. We propose a new LFS-aware ZNS interface, called **ZNS+**, and its implementation, which supports **internal zone compaction (IZC)** and **sparse sequential overwrite** via two new commands of `zone_compaction` and `TL_open`. A segment compaction requires four sub-tasks: victim segment selection, destination block allocation, valid data copy, and metadata update. Whereas all others must be performed by the host file system, it is better to offload the data copy task to the SSD, because the device-side data copy is faster than the host-side copy. For compaction acceleration, ZNS+ enables the host to offload the data copy task to the SSD via `zone_compaction`.

To avoid segment compaction, LFS can utilize an alternative reclaiming scheme, called *threaded logging* [19, 26, 28], which reclaims invalidated space in existing dirty segments by overwriting new data. It requires no cleaning operations, but generates random overwrites to segments. In the F2FS experiments using a normal SSD [19], threaded logging showed smaller write traffic and higher performance than segment compaction. However, threaded logging is incompatible with the sequential write-only ZNS interface owing to its random writes. Therefore, the current F2FS patch for the ZNS is disabling threaded logging [3]. In this paper, we will use the term *segment recycling* to cover both segment compaction and threaded logging.

The *sparse sequential overwrite* interface of ZNS+ is a relaxed version of the *dense sequential append write* constraint in ZNS. For a zone opened via `TL_open`, the sparse sequential overwrite is permitted for threaded logging. The ZNS+ SSD transforms sparse sequential write requests to dense sequential requests by plugging the holes between requests with untouched valid blocks in the same segment (**internal plugging**) and redirects the merged requests to a newly allocated flash blocks. Similar to IZC, internal plugging internally copies the valid data of a segment without involving any host-side operations. The only requirement of the write pattern for internal plugging is that the block addresses of the consecutive writes must be in the increasing order. Because the internal plugging is handled between write requests, it improves the average response time of write requests compared with the batch-style segment compaction. Although there are significant extensions in ZNS+ compared to the original ZNS, ZNS+ SSD can provide the same merits of the original ZNS SSD such as small mapping table, no duplicate GC, no over-provisioned space, and performance isolation/predictability.

ZNS+-aware File System. The file system also needs to be adapted to utilize the new features of ZNS+. First, an SSD-internal data copy operation will use different copy paths depending on the source and destination logical block addresses (LBAs). For example, when the two LBAs are mapped to the same flash chip, the *copyback* operation of flash memory can be utilized, which moves data within a flash chip without off-chip data transfers, thus reducing the data migration latency. The copyback operation is currently in the standard NAND interface [6], and its usefulness at SSD-internal garbage collection has been demonstrated by many studies [15, 30, 33]. To fully utilize the copyback operations, we propose the **copyback-aware block allocation** for segment compaction, which attempts to allocate the destination LBA of a data copy such that both the source LBA and destination LBA of the target data are mapped to the same flash chip. The technique can be extended to target other fast copy paths of SSDs.

Second, because ZNS+ supports both segment compaction acceleration and threaded logging, the host file system needs to choose one of those segment recycling policies. Although threaded logging can avoid the segment compaction overhead, it has several drawbacks, as will be explained at §3.3.2. Considering both the merits and demerits of threaded logging, we propose the **hybrid segment recycling** technique for ZNS+, which selects either threaded logging or segment compaction based on their reclaiming costs and benefits.

We implemented the ZNS+ SSD at the SSD emulator of FEMU [22] and an OpenSSD device [29]. In the experiments, the file system performance of the storage system composed of our modified F2FS and ZNS+ SSD was 1.33–2.91 times better than that of the storage system based on the original F2FS and ZNS SSD. The source code of ZNS+ is publicly available at `https://github.com/eslab-skku/ZNSplus`.

2 Background

2.1 SSD Architecture

Modern SSDs consist of multiple flash chips and adopt a multi-channel and multi-way architecture for parallelism. There are multiple parallel flash controllers (channels), and each controller can access multiple flash chips (ways) in an interleaved manner. Each flash chip has multiple flash erase blocks, and each block is composed of multiple flash pages. Flash pages cannot be overwritten until the corresponding flash block is erased. Therefore, SSDs adopt an out-of-place update scheme and use a special firmware, called the flash translation layer (FTL), to manage the logical-to-physical mappings that translate logical addresses used by the host into physical addresses indicating the location within flash memory chips. Typically, flash memory manufacturers recommend that the pages inside a flash block be programmed sequentially in the page number order owing to inter-cell interference. Because the flash page size in recent flash products is usually larger than the logical block size (i.e., 4 KB), multiple logically consecutive blocks will be written at a physical flash page in a cluster. In this study, we refer to the logical consecutive blocks mapped to a flash page as a *chunk*.

Flash memory chip generally supports read, program, erase, and copyback commands. The *copyback* command is used to copy data between flash pages within a flash chip without off-chip data transfer[1]. The chunk is the basic unit of the copyback operation. Because the chip-internal data transfer cannot check the error correction code (ECC) of the target page, the bit error propagation problem exists. To cope with the issue, the error checking can be performed by the flash controller at the same time while doing the copyback operation. If an error is detected, the copied page is invalidated and the corrected data is programmed by the flash controller. Another solution is to allow only a limited number of consecutive copybacks using a threshold copyback counts, which is determined based on copyback error characteristics of flash chip [15, 33].

2.2 Zone Mapping in ZNS SSD

The current NVMe standard interface defines ZNS commands and zone types [5]. There are several zone management commands, such as open, close, finish, and reset for a zone, as well as read and write commands. A notable command under discussion is `simple copy`, through which the host can order the SSD to copy data internally from one or more source logical block ranges to a single consecutive destination logical block range. Although it is apparently similar to our `zone_compaction` command, no studies regarding the issues of the copy command are currently available, and `simple copy` does not fit for our ZNS+, as will be explained at §3.

[1] A flash chip consists of multiple flash dies, each of which has multiple flash planes. Specifically, the copyback can be used for a data copy within a flash plane. In this study, we assume that a flash chip has one die and one plane structure for simplicity. Therefore, we use the term "flash chip" instead of "flash plane" to denote the target device for the copyback command.

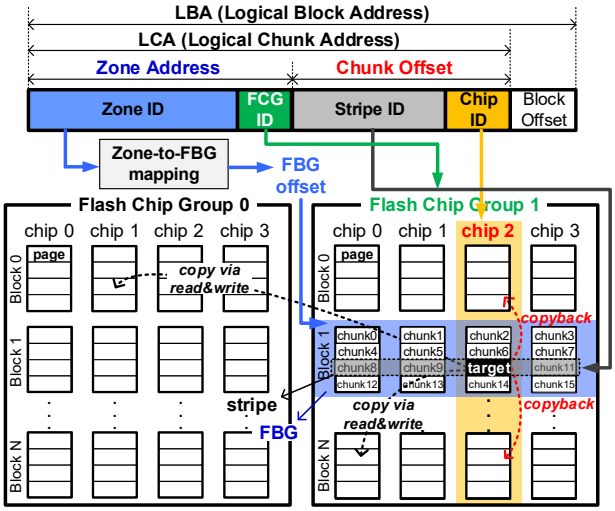


Figure 1: An example of zone and chunk mapping. With the zone address, the second FBG in the FCG 1 is selected. With the chunk offset, the third stripe in the selected FBG and the third flash page (chip 2) in the stripe are targeted.

There are no zone mapping constraints in the ZNS specification. The physical locations of a zone and the chunks in the zone within the storage device are transparent to the host. Device vendors can choose different mapping policies that consider internal design issues. We introduce a general and efficient SSD-internal zone mapping policy, which can minimize the size of required mapping information and maximize the flash chip-level parallelism. Depending on the zone size, one zone can be mapped to one or more physical flash blocks, which are called the **flash block group** (**FBG**) mapped to the zone. The zone size needs to be aligned to the size of the flash block to prevent the creation of partially valid flash blocks. To maximize the flash operation parallelism, the flash blocks from a set of flash chips accessible in parallel will compose the FBG of a zone, and the chunks of a zone need to be interleavingly placed on the parallel flash chips. The number of parallel flash chips for the chunk interleaving is referred to as the *zone interleaving degree* D_{zone}, and the set of logically consecutive chunks across the parallel flash chips is referred to as a **stripe**. For a coarse-grained zone-to-FBG mapping, an FBG has flash blocks at the same block offset in the parallel flash chips, and the chunks of a stripe are located at the same page offset in different flash blocks.

D_{zone} can be smaller than the maximum number of parallel flash chips, D_{max}. Then, D_{zone} needs to be a divisor of D_{max} to partition all the parallel flash chips of an SSD into the same size of **flash chip groups** (**FCGs**). When N_{FCG} denotes the number of FCGs, $N_{FCG} = D_{max}/D_{zone}$. For a given logical chunk address, SSD must determine the mapped flash page of the chunk. Figure 1 presents an example of the zone and chunk mapping, where D_{zone} is 4, and D_{max} is 8. A logical chunk address is divided into the zone address and chunk offset. First, the mapped FBG of the target zone is determined

with its zone address. We assume that the FCG of a zone is statically determined by the least significant $\log_2 N_{FCG}$ bits of the zone address (i.e., FCG ID in Figure 1). Therefore, the i-th zone is mapped to the (i mod N_{FCG})-th FCG. Such a static FCG mapping can reduce the size of zone mapping entry and support our copyback-aware block allocation. The remaining bits of zone address (i.e., zone ID) is used to determine the FBG within the selected FCG. Because the FBG of a zone is dynamically allocated whenever the zone is opened, SSD must maintain the zone-to-FBG mapping table.

The chunk offset is composed of a stripe ID and a chip ID. The former locates a stripe within the target FBG, and the latter determines the target chip index within the target FCG. The bit size of chip ID field is same to $\log_2 D_{zone}$, and the j-th logical chunk of a zone is mapped to the (j mod D_{zone})-th flash chip of the FCG allocated to the zone. Such a direct mapping from chunk offset value can avoid chunk-level fine-grained address mapping. The ZNS SSD only needs to manage the zone-to-FBG mapping. The host can easily calculate the mapped flash chip of a logical chunk by using the FCG ID and chip ID in the logical chunk address without knowing the SSD-internal zone-to-FBG mapping. When a logical chunk needs to be copied to other logical address, the corresponding flash page can be copied within ZNS+ SSD. If the destination logical address is mapped to the same flash chip, the copyback operation can be utilized. Otherwise, a normal copy operation via read and write commands is used.

A zone can be reset via a reset command, which changes the **write pointer (WP)** of the zone to the first block location to overwrite the zone. The WP locates the block address where new data can be written. Owing to the sequential write-only scheme of the ZNS, the WP of a zone is always incremented while the zone space is consumed. Because flash blocks cannot be overwritten, a new FBG is allocated to the zone to be reset, and the zone-to-FBG mapping is modified accordingly. The old FBG can be reused for other zones after it is erased.

2.3 F2FS Segment Management

In this study, we target F2FS [19] as a ZNS-aware file system, which is an actively maintained LFS. As shown in Figure 2, F2FS maintains six types of segments (i.e., hot, warm, and cold segments for each node and data) and uses the multi-head logging policy. Only one segment can be open for each type at a time. Separating hot and cold data into different segments can reduce segment compaction cost. A node block contains an inode or indices of data blocks, whereas a data block contains either directory or user file data. Cold blocks in the hot and warm segments are moved into the cold segments during segment compaction. F2FS supports both append logging and threaded logging. In the append logging, blocks are written to clean segments, yielding strictly sequential writes. On the other hand, threaded logging writes blocks to obsolete space in existing dirty segments without cleaning operations. F2FS uses an adaptive logging policy. When the number of free segments is sufficient, append logging is used first. However, if free segments become insufficient, threaded logging is enabled not to consume further free segments, instead of invoking segment compaction. However, threaded logging is disabled in the current F2FS patch for ZNS, and thus segment compactions will be frequently triggered at the F2FS for ZNS.

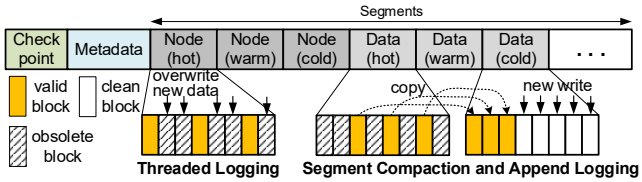


Figure 2: F2FS disk layout and logging schemes.

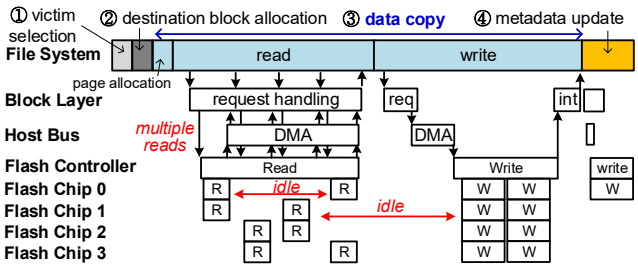
Regarding to segment compaction, F2FS supports both foreground and background operations. The foreground compaction is invoked when there are no sufficient free segments to process incoming write requests. Thus, write requests are delayed during the compaction. The background compaction is triggered only when the file system is idle and the number of free segments is below a threshold. Therefore, the IO delay due to the background compaction is insignificant. Because the threshold is configured small enough so that the compaction does not occur frequently and thus does not harm the lifespan of SSD, the invalidated space cannot be reclaimed in time with only the background compaction, especially when the file system utilization is high and there are a burst of write requests. Therefore, it is important to optimize foreground compaction to improve the overall IO performance. In this paper, we focus on the foreground compaction performance.

3 ZNS+ Interface and File System Support

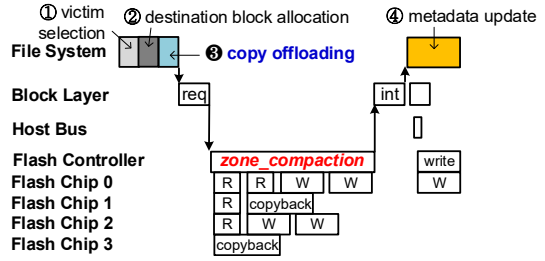
3.1 Motivation

Normal Segment Compaction. The overall process of normal LFS segment compaction consists of four tasks: victim segment selection, destination block allocation, valid data copy, and metadata update, as shown in Figure 3(a). The victim selection finds a segment with the lowest compaction cost (①). The block allocation allocates contiguous free space from destination segments (②). The data copy task moves all valid data in the victim segment to the destination segments via host-initiated read and write requests, which generate significant data transfer traffic between the host and the storage (③). The data copy task has read and write phases.

Read Phase. The host sends read requests for the valid blocks of the victim segment to the SSD if they are not cached at the page cache. Prior to sending the read requests, the corresponding memory pages must be allocated in the page cache, which may invoke write requests to the storage for page frame reclamation. Because the target blocks to be copied are generally scattered at the logical address space, multiple read requests are sent one by one. The SSD reads the target data for a read request with several flash read operations, whose



(a) Normal segment compaction via host-level copy



(b) In-storage zone compaction

Figure 3: Segmentation compaction on (a) ZNS vs. (b) ZNS+ interface (Time goes to the right.)

process can be overlapped at different flash chips. If the size of read requests is small compared to the flash chip parallelism, there will be many idle intervals of flash chips. The data read from the flash chips are transferred to the host via a storage interface such as NVMe. At the experiments using an SSD emulator, we observed that the request submission/completion handling and the device-to-host data transfer accounted for about 7% and 44% of the total read latency, respectively. (Detailed experimental environments are presented in §4.)

Write Phase. This phase can start after all the read requests issued at the read phase are completed. Because the LFS sequentially allocates new blocks for write operations at the destination segment in the append logging scheme, the file system will make one large write request to reduce the request handling overhead. Therefore, the file system waits until all the target blocks are transferred to the page cache, instead of immediately issuing a write request for each block when its read operation is completed. As a result, there is a large idle interval of the SSD, as shown in Figure 3(a).

Metadata Update. F2FS can maintain the consistency of the file system by rolling data back to the most recent checkpoint state when a sudden crash occurs. The file system writes several modified metadata blocks and node blocks to the storage to reflect the change in the data locations and then writes a checkpoint block (④). The metadata must be modified persistently to reclaim the storage space that was occupied by the valid blocks of the victim segment prior to segment compaction. Otherwise, data loss can occur when new data are overwritten in the reclaimed space.

IZC-based Segment Compaction. Figure 3(b) presents the compaction process under our IZC scheme. The data copy task of the normal segment compaction is replaced by the copy offloading (❸), which sends `zone_compaction` commands to transfer the block copy information (i.e., the source and destination LBAs). Because the target data are not loaded into the host page cache, the corresponding page cache allocation is not required. The SSD-internal controller can schedule several read and write operations efficiently while maximizing flash chip utilization. Therefore, the segment compaction latency can be significantly reduced. In addition, the in-storage block copy can utilize copyback operations.

Table 1: Comparison between ZNS and ZNS+

	ZNS	ZNS+
Copy Command	consecutive dest. range (`simple copy`)	dest. LBAs (`zone_compaction`)
Write Constraint	dense seq. write can reuse only after reset	spare seq. overwrite (`TL_open`)
Mapping Transparency	invisible	visible chunk mapping (`identify_mapping`)

3.2 LFS-aware ZNS+ Interface

Table 1 compares the original ZNS and the proposed ZNS+ interface. The ZNS+ supports three new commands, `zone_compaction`, `TL_open`, and `identify_mapping`. `zone_compaction` is used to request an IZC operation. For comparison, the `simple copy` command of the current ZNS standard delivers a single consecutive destination LBA range. Under our ZNS+ interface, the destination range can be non-contiguous; thus, our `zone_compaction` command is designed to specify the destination LBAs rather than a consecutive block range. `TL_open` is used to open zones for threaded logging. The *TL_opened* zones can be overwritten without reset, and the overwrite requests can be sparse sequential. The host can use the `identify_mapping` command to know the address bit fields which determine the mapped flash chip of each chunk.

3.2.1 Internal Zone Compaction

The process of segment compaction in the ZNS+ storage system is as follows.

(1) Cached Page Handling. The first step is to check whether the corresponding page is being cached on the host DRAM for each valid block in the victim segment. If the cached page is dirty, it must be written to the destination segment and must be excluded from IZC operations. If the cached page is clean, it can either be written via a write request or internally copied via `zone_compaction`. If it is transferred from the host via a write request, the corresponding flash read operation can be skipped. Because it is already cached, page allocation is also not needed. Instead, data transfer and write request handling overheads are involved. By comparing the flash read cost and data transfer cost, we can choose a proper scheme to relocate the cached block. In general, high-density flash memories based on TLC or QLC technologies have a relatively higher access latency than the host-to-storage data

transfer cost. However, the recent ZNAND [11] has an extremely short read latency; thus, it may be better to use the in-storage copy for the cached blocks in the ZNAND SSD.

(2) Copy Offloading. Second, to offload the data copy operations to the ZNS+ SSD, `zone_compaction(source LBAs, destination LBAs)` commands are generated. The data in the *i*-th source LBA is copied into the *i*-th destination LBA by the ZNS+ SSD. When threaded logging is enabled at F2FS, the segment compaction can select a TL_opened segment as destination, similarly to the hole-plugging [25,31]. Then, the destination LBAs can be non-contiguous.

(3) Processing IZC. Finally, ZNS+ SSD processes `zone_compaction` commands. It identifies *copybackable* chunks that can be copied with copyback operations by checking the mapped flash chips of their source and destination LBAs. A chunk is copybackable only when all the blocks in it must be copied. The SSD firmware issues flash read and write operations for *non-copybackable* chunks.

Async Interface and Request Scheduling. The handling of `zone_compaction` command is *asynchronous*. The compaction command issued by the host will be enqueued into the command queue, and the host will not wait for the completion of the command. Therefore, the following IO requests can be immediately issued by the host before the completion of the pre-issued zone compaction. The asynchronous handling can improve the performance by eliminating the waiting time of the following requests. Owing to the checkpoint scheme of LFS, the asynchronous command handling does not harm the file system consistency.

Under the asynchronous interface, there will be multiple pending normal requests while handling zone compaction. ZNS+ SSD can reorder normal requests to avoid the convoy effect by the long latency of zone compaction. If the target zone of a normal request is irrelevant to the zone compaction request arrived in advance, it can be processed before the completion of zone compaction. Even for a read request to the destination zone of an on-going zone compaction, the read request can be handled if the WP of the zone has passed through the target block address of the read request.

3.2.2 Sparse Sequential Overwrite

Internal Plugging. To support threaded logging, ZNS+ supports sparse sequential overwrite. Although the write pattern of threaded logging is incompatible with ZNS, there is one consolation; threaded logging accesses the free space of a dirty segment in an increasing order of block address because it consumes the lower address of blocks first. Therefore, its access pattern is sparse sequential (i.e., the WP of a zone will not be decremented). While threaded logging overwrites a segment, if the SSD firmware reads the *skipped blocks* between requests and merges them to the host-sent data blocks, it can make dense sequential write requests to the target zone; this technique is referred to as *internal plugging*. Because the plugging operation is SSD-internal, the latency is shorter than

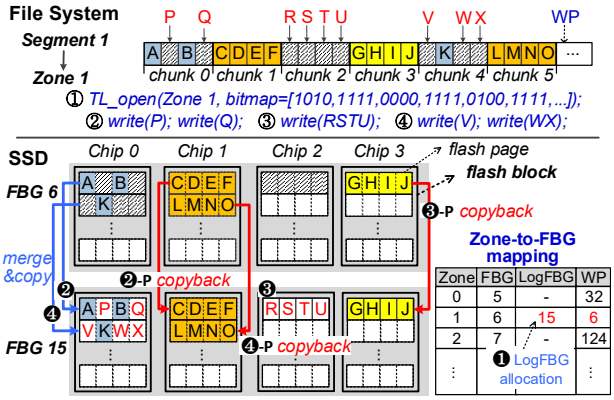


Figure 4: Skipped block plugging for threaded logging

the host-level copy latency. In addition, the SSD can schedule plugging operations efficiently across parallel flash chips to hide their latency.

Figure 4 illustrates an example of internal plugging. Segment 1 is now mapped to Zone 1 in the file system, and FBG 6 is allocated to Zone 1, as shown in the zone-to-FBG mapping. An FBG is composed of four flash blocks, each of which is in different flash chips. The host file system allocates Segment 1 for threaded logging, and sends write requests to invalid blocks to reclaim them while skipping valid blocks. For example, the blocks of A and B in chunk 0 are skipped blocks.

Opening Zone for Threaded Logging. For internal plugging, the SSD must perceive the skipped blocks in the target segment of threaded logging. Owing to the in-order request delivery of ZNS IO stack, the SSD can identify the skipped blocks by comparing the start LBA of an incoming write request with the current WP of the corresponding zone. However, only after a write request arrives, the preceding skipped blocks can be recognized. Therefore, the plugging operation will delay the handling of the write request. To solve this problem, we added a special command, called `TL_open(open zones, valid bitmap)`, that delivers the valid bitmap of the target zones selected for threaded logging (①). Once an allocated segment is informed via `TL_open`, the threaded logging reclaims only the invalid blocks marked at the transferred bitmap. Therefore, the SSD can identify the blocks to be skipped by threaded logging in advance and perform the plugging before the following write request arrives.

LogFBG Allocation. Because a TL_opened zone will be overwritten, the ZNS+ SSD resets the WP of the zone and allocates a new FBG, called **LogFBG**, where new data to the zone are written. For example, in Figure 4, the SSD allocates a LogFBG, FBG 15, for Zone 1 (❶). For the TL_opened zone, the data blocks of the zone are distributed into two FBGs, i.e., the original FBG (FBG 6) and the LogFBG (FBG 15). Therefore, both of them must be maintained as mapped FBGs for the zone. To handle a read request, the SSD identifies the block location of up-to-date data by comparing the target LBA with the WP. If the target LBA is behind the WP (target LBA < WP), the LogFBG is accessed. Otherwise, the original

FBG is accessed for the read request. While the invalid blocks of a TL_opened zone are overwritten by threaded logging, all the valid blocks are copied into the LogFBG. Under a static zone mapping policy, each logical chunk of a zone is always mapped to the same flash chip whenever a new FBG is allocated to the zone. Therefore, fully valid chunks can be copied from the original FBG to the LogFBG via copyback during internal plugging. The number of allocated LogFBGs is determined by the number of TL_opened segments, which is six at maximum in F2FS. Therefore, the space overhead due to LogFBGs is negligible. When the TL_opened zone is finally closed, the LogFBG replaces the original FBG, which is deallocated for future reuse.

LBA-ordered Plugging. When the SSD receives two write requests to chunk 0 in Figure 4 (②), it reads the skipped blocks of A and B from FBG 6, merges them with the host-sent blocks of P and Q, and writes a full chunk at the LogFBG (❷). After handling the write requests to chunk 0, the SSD can perceive that chunk 1 will be skipped by checking the valid bitmap of the zone. To prepare the WP in advance for the write request to chunk 2, the skipped chunk must be copied to the LogFBG. Therefore, after processing a write request, if the following logical chunks are marked as valid in the valid bitmap, the ZNS+ SSD copies them to the LogFBG while adjusting the WP (❷-P, ❸-P, and ❹-P). This type of plugging is called *LBA-ordered plugging* (LP), where each plugging is performed at the current WP of the zone to follow the LBA-ordered write constraint.

PPA-ordered Plugging. Although incoming data must be written at the WP of zone, there is no need to perform the internal plugging operation only at the WP. The internal plugging can be done in advance at the block addresses in ahead of the WP. We only need to consider that the flash pages in a flash block must be programmed sequentially. Therefore, the plugging operation of a fully valid chunk can be scheduled in advance even when the current WP is behind the location where the chunk must be copied to. A fully valid chunk can be copied to a physical page address (PPA) if all the flash pages at lower PPAs within the target flash block have been programmed. For the example in Figure 4, chunk 3 can be copied before the write requests to chunk 0 and chunk 2 arrive because they use different flash chips. If chunk 1 has been copied, chunk 5 can be copied even when the write requests to chunk 2 and chunk 4 have not yet arrived. We call this technique *PPA-ordered plugging* (PP), which considers only the PPA-ordered write constraint. Whenever a flash page is programmed at a LogFBG, the PPA-ordered plugging checks the validity of the chunks mapped to the following flash pages in the corresponding flash block and issues all possible plugging operations for the flash block in advance. However, if excessive plugging operations are issued, they may interfere with the user IO request handling. To resolve this problem, the plugging operations are processed in the background when the target flash chip is idle. If there is no sufficient idle time,

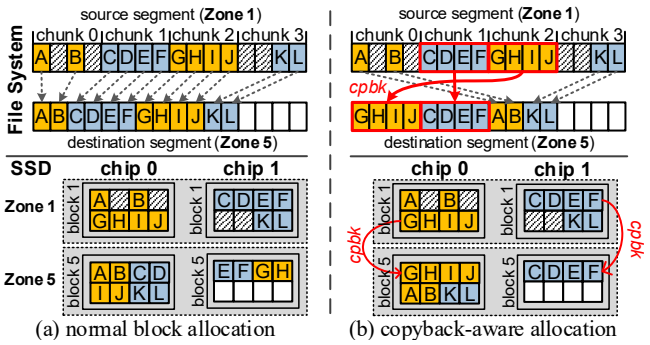


Figure 5: Copyback-aware block allocation

they are handled when the WP of the zone must pass the skipped block locations.

Why Threaded Logging Improves Performance. We can consider that the segment compaction cost restrained by threaded logging is revived in the form of internal plugging in the SSD. This is because the number of blocks to be copied by internal plugging at threaded logging is equal to the number of blocks to be copied at segment compaction for the same segment. However, the metadata modifications to reflect data relocation can be avoided by threaded logging. In addition, the internal plugging cost can be hidden by utilizing idle flash chips. Moreover, the plugging operations are distributed between normal write requests while minimizing the average delay of write requests. On the contrary, the segment compaction is a batch operation. No write requests can be processed until the segment compaction is completely finished. Thus, ZNS+ system can show a better performance when threaded logging is enabled. Because the host knows the amount of skipped blocks within a segment, the internal plugging does not harm the predictability of ZNS. Although the internal plugging amplifies flash write operations, there is no additional endurance degradation compared to segment compaction, which also generates the same amount of flash write operations to reclaim invalidated space.

3.3 ZNS+-aware LFS Optimization

3.3.1 Copyback-aware Block Allocation

Low Utilization of Copyback at LFS. For the valid blocks to be relocated at segment compaction, new block locations are sequentially allocated from a destination segment in the order of their source LBAs in the original LFS. Therefore, the scattered valid blocks in the victim segment are simply compacted without holes between them at the destination segment; in this type of scheme, most of the chunks will not be copybacked. Figure 5(a) illustrates an example of segment compaction under normal block allocation, in which two logically consecutive chunks comprise a stripe over two flash chips. Under normal block allocation, no chunks can be copybacked in this example.

Chunk Mapping Identification. To maximize the usage of copyback, we propose the *copyback-aware block allocation*.

Figure 5(b) illustrates an example of segment compaction under the copyback-aware block allocation. First, the file system reserves contiguous free blocks at the destination segment as the number of valid blocks to be copied by segment compaction. Second, the file system allocates the destination chunk location from the reserved region for each fully valid chunk in the source segment (e.g., chunk 1 and chunk 2 in the example) such that both the source and destination chunks are mapped to the same flash chip. Under a static chunk mapping scheme, the host can easily calculate the mapped flash chip number of a chunk if it knows which bit ranges of logical chunk address determine the chip offset. For the `identify_mapping` command, the ZNS+ SSD returns the bit ranges of FCG ID and chip ID, shown in Figure 1. If two chunk addresses have the same values of FCG ID and chip ID, they are mapped to the same flash chip. The host queries the chunk mapping information only while booting, and no additional inquiry is required at run time. After handling all the fully valid chunks, the destination block locations of the remaining valid blocks are determined to fill all the free space of the reserved region. For example, the new block locations of A, B, K, and L in Figure 5(b) are allocated to the remaining space and can be copied via flash read and write operations.

Maximizing Copyback Usage. If the fully valid chunks in the source segment are not evenly distributed among multiple flash chips, some chunks cannot find the copyback-able chunk locations from the reserved region and must be copied to non-copybackable chunks. One solution is to allocate additional chunks in the destination segment to maximize the usage of copyback, while leaving some unused blocks in the destination segment, which is possible because the `zone_compaction` command can specify non-contiguous destination ranges. Another issue is to use copyback for partially invalid chunks (e.g., chunk 0 and chunk 3 in Figure 5). By copying an entire chunk including invalid blocks via copyback, we can reduce the segment compaction time. Although allocating more space in the destination segment can maximize the usage of copyback, the segment reclaiming efficiency can be degraded. Considering this trade-off, a threshold for the allowed additional space needs to be determined. A more detailed consideration is beyond the scope of this study.

Extensions. The proposed copyback-aware block allocation can be extended for recent multi-core SSDs, where multiple embedded processors exist with each processor running an FTL instance to manage its own partitioned address space and flash chips while utilizing the processor-level parallelism [18, 35]. In the multi-core SSD, a zone can be mapped across multiple partitions, and the inter-partition copy latency will be longer than that of intra-partition copy because communication overhead will be imposed for the inter-partition operation. Therefore, a *partition-aware block allocation* will be beneficial for the multi-partitioned ZNS+ SSDs.

Instead of the file system-level copyback-aware block allocation, we can consider a device-level approach, where the target block location is not specified by the host, and the SSD determines the logical block locations to maximize the usage of copyback and informs the file system of the allocated block locations, similarly to the ideas of the nameless write [36] and the `zone append` command defined in the standard ZNS interface. This approach will enable copyback-aware block allocation even when the host has no knowledge of the SSD-internal chunk mapping.

3.3.2 Hybrid Segment Recycling

Although threaded logging can reduce the block reclamation overhead, its reclaiming efficiency can be lower than that of segment compaction in the ZNS+ owing to two reasons.

Reclaiming Cost Imbalance. First, threaded logging may suffer from unbalanced reclaiming costs among different types of segments. Whereas segment compaction selects a victim segment with the lowest compaction cost (i.e., the smallest number of valid blocks) among all dirty segments, threaded logging can select the target segment for a type of write request only from the same type of dirty segments to prevent different types of data from being mixed in a segment. Even when there are multiple segments whose blocks are mostly invalidated, threaded logging cannot utilize them for a write request if the types of those segments are different from the data type of the write request. Instead, the same type of dirty segment must be selected despite a high internal plugging cost. In addition, unlike segment compaction, threaded logging cannot move cold data into cold segments. The cold data trapped in a segment must be copied by the internal plugging each time the segment is opened for threaded logging.

Pre-Invalid Block Problem. Second, the reclaiming efficiency of threaded logging will be further degraded if threaded logging is used for a long period without checkpointing, which is not mandatory for threaded logging. This is due to *pre-invalid blocks* that are invalid but still referenced by in-storage metadata and thus non-reclaimable. When a logical block is invalidated by a file system operation but a new checkpoint is still not recorded, the logical block becomes pre-invalid and must not be overwritten because a crash recovery will need to restore it. They must be copied by the internal plugging. The pre-invalid blocks accumulate as threaded logging continues without checkpointing, whereas they can be reclaimed by segment compaction because segment compaction accompanies checkpointing. The original F2FS prefers threaded logging because the performance gain by avoiding segment compaction is more significant than the drawback of reclaiming inefficiency in threaded logging, which may be true in legacy SSDs. However, the reclaiming inefficiency results in a high internal plugging cost at the ZNS+ SSD.

Periodic Checkpointing. To solve the pre-invalid block problem, we use the *periodic checkpointing*, which triggers checkpointing whenever the number of accumulated pre-invalid blocks exceeds θ_{PI}. For periodic checkpointing, the file system must monitor the number of pre-invalid blocks. If

checkpointing is invoked too frequently, the write traffic on metadata blocks increases, and the flash endurance of SSD will be harmed. Therefore, an appropriate value of θ_{PI} needs to be determined considering the trade-off. From experiments using several benchmarks, we identified that the overall performance was maximized when θ_{PI} was around 128 MB. Thus, θ_{PI} was configured to the value in the experiments in §4.

Reclaiming Cost Modeling. We propose the *hybrid segment recycling* (HSR) technique, which chooses a reclaiming policy by comparing the reclaiming costs of threaded logging and segment compaction. The reclaiming cost of a segment under threaded logging, C_{TL}, can be formulated as follows:

$$C_{TL} = f_{plugging}(N_{pre\text{-}inv} + N_{valid}) \quad (1)$$

$N_{pre\text{-}inv}$ and N_{valid} indicate the number of pre-invalid blocks and the number of valid blocks, respectively. $f_{plugging}(N)$ is the in-storage plugging cost for N blocks. Because the plugging operation can be performed in the background, $f_{plugging}(N)$ is less than on-demand internal copy cost. We set $f_{plugging}(N)$ to be 90% of the on-demand copy cost based on the experimental results on performance improvement by the internal plugging.

Segment compaction only moves the valid blocks of the victim segment to the free segment. During segment compaction, cold blocks are moved to cold segments, which will decrease the future reclaiming cost of the target segment. However, it must modify node blocks and metadata blocks to reflect the change in block locations at the checkpointing. Therefore, the reclaiming cost of segment compaction, C_{SC}, can be expressed as follows:

$$C_{SC} = f_{copy}(N_{valid}) + f_{write}(N_{node} + N_{meta}) - B_{cold} \quad (2)$$

N_{node} and N_{meta} denote the numbers of the modified node blocks and metadata blocks, respectively. $f_{copy}(N)$ and $f_{write}(N)$ are the copy cost and the write cost of N blocks, respectively. B_{cold} represents the predicted future benefit from cold block migration. When the victim segment of segment compaction is a node segment, no additional node update occurs; thus, N_{node} is 0. To estimate $f_{copy}(N_{valid})$, we can assume that all the chunks are copied via in-storage copy operations, and some portion of the copy operations can be handled by copyback commands.

Using Approximate Cost. Whereas $N_{pre\text{-}inv}$ and N_{valid} can be easily calculated by examining the valid bitmap of each segment, the calculations of N_{node} and N_{meta} are not simple. They are determined by the number of node blocks associated with the data blocks that are relocated at segment compaction. It is highly expensive to precisely calculate each number during the selection process of the reclaiming policy. Therefore, we use approximate values for N_{node} and N_{meta}. Assuming that they become larger in proportion to N_{valid}, $\alpha \times N_{valid}$ can be used instead of the real value of $(N_{node} + N_{meta})$. The value of α depends on workloads, and thus, its average value can be profiled at run time. For our target benchmarks, we observed that α has an average value of 20%.

It is also difficult to predict the exact value of B_{cold}, and thus, it is approximated to $f_{plugging}(\beta \times N_{cold})$, where N_{cold} is the number of blocks unchanged across two consecutive TL-based segment recyclings of the target segment, assuming that β% of N_{cold} in the segment will be still valid when the segment is TL_opened again next time. Therefore, $f_{plugging}(\beta \times N_{cold})$ amount of internal plugging cost at the future threaded logging can be avoided by moving the cold blocks at the current segment recycling. The value of β also depends on workloads, and its appropriate value can be profiled at run time.

By comparing C_{TL} of the threaded logging and C_{SC} of the segment compaction, the HSR chooses one of the recycling policies. Note that threaded logging and segment compaction will select different victims because they examine different candidates. When the reclaiming cost imbalance among different segment types is serious, segment compaction will be chosen because it can find better victim segments.

4 Experiments

We evaluated the performance of ZNS+ SSD using an SSD emulator, which was implemented based on FEMU [22]. In the emulation environment, the host computer system used the Linux 4.10 kernel and was equipped with an Intel Xeon E5-2630 2.4 GHz CPU and 64 GB of DRAM. We allocated four CPU cores, 2 GB of DRAM, 16 GB of NVMe SSD for user workloads, and a 128 GB disk for the OS image to the guest virtual machine. The emulated NVMe SSD consists of 16 parallel flash chips by default, which can be accessed in parallel via eight channels and two ways per channel. The type of flash chip was configured to either TLC, multi-level cell (MLC), or ZNAND, as shown in Table 2. The default flash medium was the MLC in our experiments. The data transmission link between the host and the SSD was configured to a two-lane PCIe Gen2 with the maximum bandwidth of 600 MB/s per lane. The zone interleaving degree, D_{zone}, was set to be the same as the maximum number of parallel flash chips (i.e., $D_{zone} = 16$). Therefore, the default zone size of ZNS+ SSD is 32 MB, which is the total size of 16 flash blocks distributed in 16 flash chips. When the number of parallel flash chips was configured to a different value, the zone size was also changed according to the size of the parallel FBG. To determine the effect of copyback, we modified FEMU to support the copyback operation. Table 2 shows the copyback latency normalized by the latency of the copy operation using the read-and-program commands. The copyback operation is approximately 6–10% faster than the normal copy operation.

We modified F2FS version 4.10 to exploit the ZNS+ interface. The complexity of F2FS was increased by 5.4% to implement ZNS+-aware F2FS (from 20,160 LoC to 21,239 LoC). The F2FS segment size was configured to be equal to the zone size; therefore, its default size is 32 MB. The *copyback-aware block allocation* was applied, which was enabled by default in the experiments. The filebench [4] (fileserver and varmail)

Table 2: Flash memory configurations

	TLC [24]	MLC [23]	ZNAND [11]
Flash page read latency	60 μs	35 μs	3 μs
Flash page program latency	700 μs	390 μs	100 μs
DMA from/to flash controller	24 μs	24 μs	3.4 μs
Normalized copyback latency	0.94	0.90	0.94
Flash page: 16 KB, Pages per flash block: 128, 16 flash chips (default)			
Host interface: PCIe Gen2 2x lanes (max B/W: 1.2 GB/s)			

Table 3: Benchmark configurations

fileserver	112,500 files, file size: 128KB, 14GB fileset, 50 threads
varmail	475,000 files, file size: 28KB, 12.7GB fileset, 16 threads
tpcc	DB size: 12GB, 1GB buffer pool, 16 connections
YCSB	DB size: 12GB, 1GB buffer pool, 32 connections

and OLTP benchmarks (tpcc [8] and YCSB [12] on MySQL) were used for the evaluation. We set the file system size to 16 GB and determined the dataset size of each benchmark such that the file system utilization was 90% by default. The configuration of each benchmark is detailed in Table 3.

In the experiments, the following two different versions of ZNS+ were used: `IZC` and `ZNS+`. While threaded logging is disabled in `IZC`, it is enabled and the hybrid segment recycling is used in `ZNS+`. The PPA-ordered plugging was used by default in the experiments. The ZNS+ schemes were compared with `ZNS`, which uses the original F2FS (ZNS patch version) and ZNS SSD. `ZNS` uses the host-level copy to perform segment compaction and does not utilize threaded logging. Because each workload generates write requests without idle intervals, no background compactions were invoked by F2FS.

4.1 Segment Compaction Performance

Figure 6 presents the average segment compaction latencies of `ZNS` and `IZC` at various benchmarks. The SSD emulator was used for the experiments. The compaction time is divided into four phases (init, read, write, and checkpoint) and three phases (init, IZC, and checkpoint) for `ZNS` and `IZC`, respectively. The init phase reads several metadata blocks of all the files related to the victim segment. Because these metadata are generally being cached in the page cache, the init phase is short.

`IZC` reduced the zone compaction time by about 28.2–51.7%, compared to `ZNS`, by removing the host-level copy overhead and utilizing copyback operations. The ratios of the copyback operations among all the in-storage copy operations were 87%, 74%, 81%, 83%, and 83% during the workloads of the fileserver, varmail, tpcc, YCSB-a, and YCSB-f, respectively. Because the checkpoint phase must wait for the persistent completion of the previous phases, the checkpoint latency includes the waiting time for the completion of the write operations or the IZC operations. Therefore, the checkpoint latency was increased during the OLTP workloads by the `IZC` technique. The segment compaction by host-level copy operations can be disturbed by user IO requests. Whereas the IO traffic of the filebench workloads is intensive, those of the OLTP workloads are small. Therefore, the performance

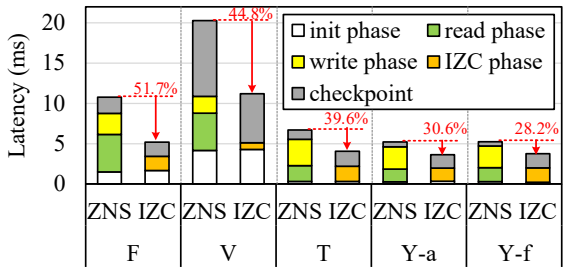


Figure 6: Average compaction time (F: fileserver, V: varmail, T: tpcc, Y-a: YCSB workloada, and Y-f: YCSB workloadf)

Table 4: The bandwidth (MB/s) at fileserver workload in different NAND flash media

	TLC	MLC	ZNAND
`ZNS`	79.5 (1.00x)	84.5 (1.00x)	104.0 (1.00x)
`IZC-H`	113.4 (1.43x)	154.6 (1.83x)	218.9 (2.10x)
`IZC-D`	96.5 (1.12x)	148.0 (1.75x)	242.4 (2.33x)

improvement by IZC is more significant during filebench workloads because the in-storage copy operations mitigate the interference with user IO requests for using the host resource and the host-to-device DMA bus.

We also compared two different policies on fully cached logical chunks, presented in §3.2.1, using different types of NAND media in Table 2. Whereas the fully cached chunks are directly written by the host in `IZC-H`, all the chunks are copied by device in `IZC-D`. Exceptionally, if the latest version of a block only exists in the host DRAM with a dirty flag, the host directly writes the block to the storage.

Table 4 compares the bandwidths of different copy schemes in different flash media. The performance gain by IZC is more significant for a faster flash media because the host IO stack contributes a larger portion of the IO latency for faster flash media. `IZC-H` and `IZC-D` offloaded 89.6% and 99.4% of block copy operations to the ZNS+ SSD, respectively. This indicates that about 10.4% of the total chunks to be copied were cached in the host DRAM (clean: 9.8%, dirty: 0.6%). When the TLC flash memory was used, `IZC-H` outperformed `IZC-D` because the TLC flash read latency is longer than the host-level write request handling overhead. However, the performance difference between `IZC-H` and `IZC-D` is reduced when the MLC flash is used. If the flash access time is further reduced by using ZNAND, `IZC-D` delivers a better performance (i.e., offloading all copy requests to the storage, regardless of whether the target blocks have been cached, achieves better performance). In the following experiments using MLC flash memory, we used `IZC-H` by default.

4.2 Threaded Logging Performance

We compared the overall performances of the benchmarks under `ZNS`, `IZC`, and `ZNS+` to evaluate the effects of in-storage zone compaction and threaded logging support, as shown in

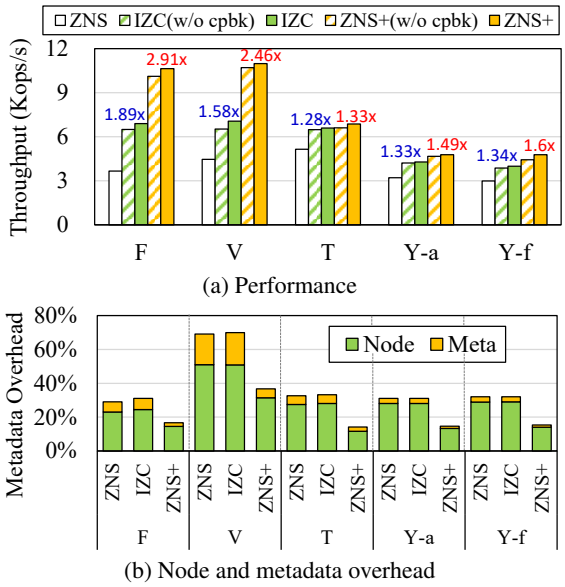


Figure 7: Effect of the threaded logging support

Table 5: Ratios of threaded logging (TL) and background plugging (BP) at ZNS+ (%)

	Fileserver	Varmail	TPCC	YCSB-a	YCSB-f
TL	94.8	92.1	85.8	89.8	89.7
BP	11.3	31.7	24.7	34.1	35.1

Figure 7(a). The copyback-disabled versions of IZC and ZNS+, IZC(w/o cpbk) and ZNS+(w/o cpbk), were also examined. Figure 7(b) presents the file system metadata overhead, which indicates the write traffic on the node blocks and the file system metadata blocks. The overhead values are normalized to the user data traffic. IZC presents about 1.21–1.77 times higher throughputs than that of ZNS because IZC can significantly reduce segment compaction time. ZNS and IZC present similar metadata overheads because they only use segment compaction for invalidated space reclamation. In the OLTP workloads, the segment compaction cost occupies a smaller portion of the total IO latency compared to filebench workloads. Therefore, the overall performance improvements at the OLTP workloads are less than those of the filebench workloads.

ZNS+ outperforms both ZNS and IZC for all benchmarks. ZNS+ presents approximately 1.33–2.91 times higher throughputs than that of ZNS. As shown in Figure 7(b), ZNS+ modifies fewer node blocks and metadata blocks because threaded logging does not invoke checkpointing. ZNS+ reduced the node and metadata write traffic by about 48%, compared to IZC, for the varmail workload. Table 5 presents the ratio of the segments reclaimed by threaded logging in ZNS+. In all the workloads, more than 85.8% of the reclaimed segments are handled by threaded logging in ZNS+, because our periodic checkpoint scheme limits the number of pre-invalid blocks. For the fileserver and varmail workloads, which update file system metadata frequently, ZNS+ presents significant

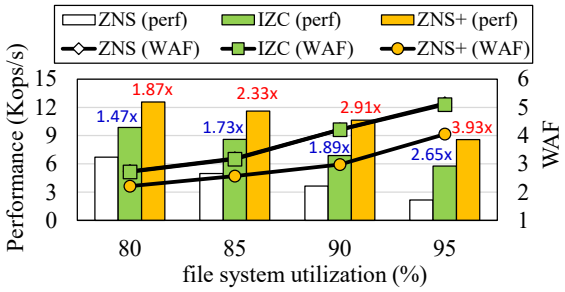


Figure 8: Performance at various file system utilizations (fileserver workload)

improvements over IZC because threaded logging achieved a high reclaiming efficiency during node segment reclaiming. Table 5 also shows the ratio of the blocks copied in the background by the PPA-ordered plugging among all the copied blocks. The ratio of background plugging is high at the fsync-intensive workloads (varmail, tpcc, and YCSB) because the small-sized fsync requests cause frequent idle intervals of the flash chips. Since the average background plugging ratio is about 27%, a significant portion of internal plugging overhead was hidden in the ZNS+ SSD.

We also compared the performances of ZNS schemes while varying the file system utilization. By changing the file-set size of the target workload, we controlled the file system utilization. Figure 8 presents the workload throughput and the write amplification factor (WAF) for each technique for the fileserver workload. The WAF is the total write traffic invoked by the file system (including data block writes, node block writes, metadata updates, segment compaction, and internal plugging) divided by the write traffic generated by the user workload. The WAF values of IZC and ZNS are similar. As the file system utilization increases, the WAF increases because segment compaction must copy a larger number of valid blocks, and segment compaction is invoked more frequently. Because threaded logging reduces the number of node and metadata updates, ZNS+ shows lower WAF values than those of IZC. The performance gain by IZC or ZNS+ over ZNS increases as the file system utilization increases because the segment recycling cost is more significant at a higher file system utilization.

4.3 SSD-internal Chip Utilization

We measured the effects of the proposed techniques on flash chip utilization, as shown in Figure 9. The IZC technique can increase the chip utilization by reducing the idle intervals invoked during the host-level copy operations, as shown in Figure 3. A higher flash chip utilization generally results in a higher IO performance. To measure the effect of the PPA-ordered plugging technique, which utilizes idle flash chips, we observed chip utilization for the following two different plugging schemes of ZNS+: LBA-ordered plugging (LP) and PPA-ordered plugging (PP). Whereas LP copies the following

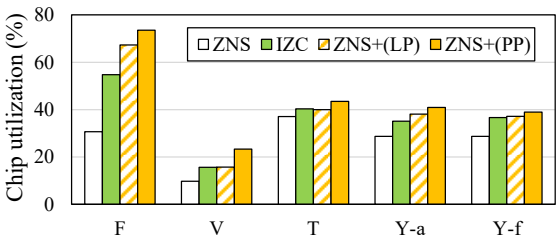


Figure 9: Flash chip utilization in different ZNS schemes

skipped blocks just after handling a write request, PP processes all the possible plugging operations whenever each flash chip is idle.

In Figure 9, IZC and ZNS+ present higher chip utilizations than that of ZNS for all workloads. Whereas ZNS+(LP) can utilize the idle interval between two consecutive write requests, ZNS+(PP) can overlap the plugging operations with normal write request handling by utilizing idle flash chips. Therefore, ZNS+(PP) showed higher chip utilizations compared to ZNS+(LP). The performance improvements achieved by the different plugging techniques were similar to the chip utilization improvements by them.

Figure 10 presents the change in chip utilization for the fileserver workload. We started to measure the utilization after the file system enters a steady state, where segment reclaiming occurs consistently. While a black dot represents the average utilization of a 100-ms interval, the red line indicates the change in the average utilization of a 5-s moving interval. Under the ZNS technique, many low utilization intervals were observed at less than 20% owing to host-level copy operations during segment compaction. IZC eliminated most of the low utilization intervals via in-storage copy operations. ZNS+ increased chip utilization significantly owing to the background plugging operation. A few low utilization intervals at ZNS+ were generated when segment compaction was chosen by the hybrid segment recycling, or the checkpoint was recorded by the periodic checkpointing. However, the periodic checkpointing is indispensable for improving reclaiming efficiency by controlling the maximum number of pre-invalid blocks.

4.4 Copyback-aware Block Allocation

Figure 11 compares the performances under the copyback-*aware* block allocation (CAB) and the copyback-*unaware* block allocation (CUB) for the fileserver workload while varying the number of parallel flash chips in the SSD. Generally, the IO performance is improved as the number of flash chips increases because of the increased IO parallelism. In our experiments, the number of flash chips determined the zone size, and the file system segment size was configured to be equal to the zone size. Therefore, as the number of flash chips increased, the segment size also increased.

Figure 11(a) presents the bandwidth of each technique. When there are only a few parallel flash chips, the performance difference between ZNS and our techniques is insignificant because the maximum internal bandwidth of SSD is extremely low, causing ZNS to fully utilize the parallel flash chips using the host-level copy. In contrast, as the number of flash chips increases, the proposed ZNS+ techniques significantly outperform ZNS.

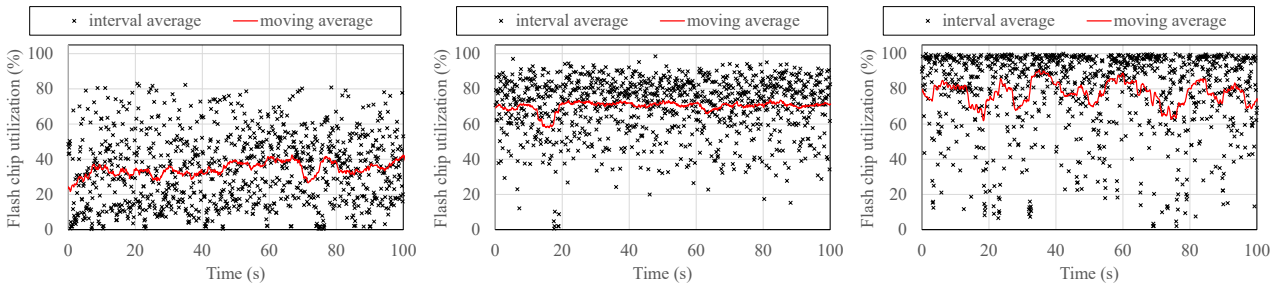
As a larger segment is used, it takes a longer time to reclaim a segment because it will have more valid blocks. The cost of checkpoint operations also increases with a large segment configuration. Therefore, ZNS and IZC present slow increase rates in bandwidth as the chip-level parallelism increases; in contrast, ZNS+ shows a faster increase rate in bandwidth because the increased block copy operations for large segments can be performed in the background by the PPA-ordered plugging. In addition, because threaded logging invokes fewer metadata updates compared to segment compaction, the checkpoint overhead does not increase significantly when the segment size is large. Consequently, the performance gap between IZC and ZNS+ increases as the number of flash chips increases.

Figure 11(b) presents the distribution of two different SSD-internal copy operations — copyback (cpbk) and read-and-program (R/P) — used to copy valid blocks during segment recycling. As the number of flash chips increases, the ratio of cpbk decreases linearly under CUB, because chunks are distributed into a larger number of chips. However, CAB causes more than 80% of the copy requests to be processed by copyback operations. Therefore, the performance of IZC-CAB is about 1.13 times better than that of IZC-CUB when the number of flash chips is 32. In the experiments, the copyback operation was configured to reduce the latency of copy operation by about 10% compared to the read-and-program operation, as shown in Table 2. Thus, a 13% performance improvement by CAB is reasonable.

As shown in Figure 11(b), the copyback ratio of ZNS+ is high despite CAB being disabled, and the number of flash chips is significant. This is because ZNS+ processes approximately 95% of the reclaimed segments with threaded logging. The fully valid chunks to be copied in the internal plugging can be copied using flash copyback operations, as shown in Figure 4. Therefore, the performance difference between ZNS+-CUB and ZNS+-CAB is insignificant.

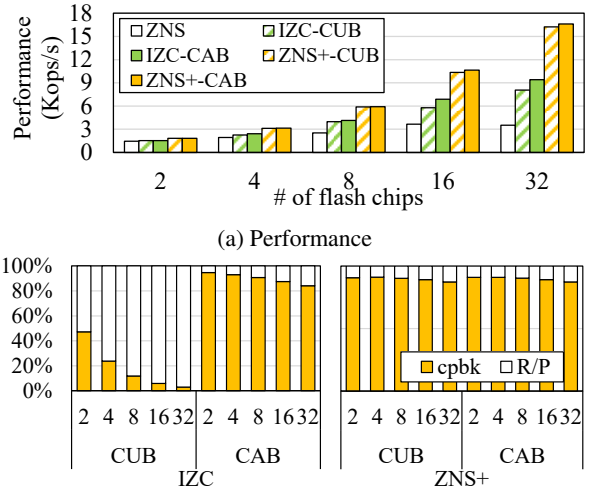
4.5 Performance at High H/W Parallelism

Figure 12(a) shows the performance change while varying the parallelism of host-to-device PCIe communication and flash chips. The PCIe communication parallelism was adjusted by changing the number of lanes, each of which was assumed to provide 600 MB/s of bandwidth. As the IO bandwidth increased, the number of flash chips was also configured to a larger value, because the total flash chip bandwidth must be high enough to utilize the increased IO bandwidth. The zone interleaving degree was configured to the total number of flash chips. Therefore, the zone size and the segment size increased as the number of flash chips increased. Although the data



(a) ZNS (b) IZC (c) ZNS+

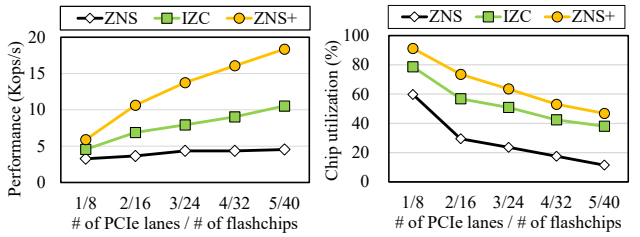
Figure 10: Flash chip utilization for fileserver workload



(a) Performance

(b) Internal copy operation breakdown

Figure 11: Performance comparison for varying chip-level parallelisms (fileserver workload)



(a) Performance (b) Chip Utilization

Figure 12: Performance at different communication and flash chip parallelisms (fileserver workload)

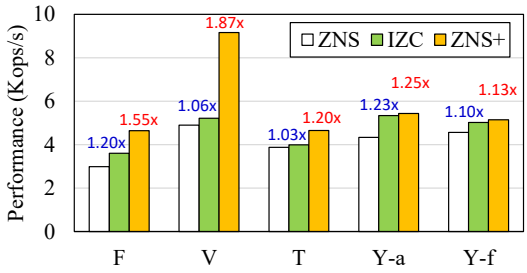


Figure 13: Performance result at a real SSD device.

transfer time between the host and the SSD can be reduced at a higher PCIe bandwidth, the performance of ZNS shows little improvements by the increased parallelism. This is due to the low chip utilization at a high chip parallelism, as shown in Figure 12(b), which is caused by idle intervals in SSD during segment compaction. Consequently, the performance gain by IZC or ZNS+ increases as the H/W parallelism increases.

4.6 Real SSD Performance

We also implemented a prototype of the ZNS+ SSD by modifying the firmware of the OpenSSD Cosmos+ platform [29] to evaluate the effects of the proposed techniques on a real system. The prototype ZNS+ SSD has two limitations compared to the SSD implemented in the FEMU emulator. First, the flash memory controller on Cosmos+ OpenSSD does not support the flash memory copyback operation; therefore, the ZNS+ SSD firmware cannot utilize it. Second, the flash memory controller does not support the partial page read operation; the entire 16 KB of the flash page must be read. Thus, SSD must use several *memcopy* operations to copy partially valid logical chunks, causing significant performance degradation, which can be easily fixed if a new flash controller is designed to support the internal copy. However, because we cannot modify the flash controller of the Cosmos+ platform, the copy requests of only fully valid logical chunks were offloaded to the SSD, and the copyback operations were replaced by the read-and-program operations in the experiments. To implement ZNS+ SSD, the number of firmware code lines was increased only by 6.3% (from 17,242 LoC to 18,334 LoC) compared to the ZNS implementation.

Figure 13 shows the performance improvement achieved by ZNS+ over ZNS for our prototype ZNS+ SSD. Owing to the limitations of Cosmos+ OpenSSD, the performance improvement in the real SSD is less significant compared to the emulation-based experiments. Nevertheless, ZNS+ improves the performance by about 13–87% compared to ZNS by minimizing IO request handling overhead and increasing flash chip utilization. The performance improvements by IZC are about 3–23%. If the flash controller is upgraded considering the internal copy operation, we will achieve higher performance improvements.

5 Related Work

Currently, SSD-based storage systems use a black-box model, where the host has no knowledge of the internal structure and management of the SSD. Such a black box model can decouple the host system and the SSD device while enabling them to communicate with each other via a simple IO interface; however, such a design causes several problems. First, duplicate logging operations and GCs can be performed in both the host and the SSD; this is called the log-on-log [34] problem. Second, it is difficult for the host to predict the IO latency owing to the complex internal operations of SSDs.

To overcome the limitations of the black box model, several studies proposed white- or grey-box models for the SSD, which expose essential knobs to control data placement, IO predictability, and IO isolation. AMF [21] proposed a new block interface that supports append-only IO via read/write/trim commands and an LFS for the interface. It solved the log-on-log problem by a direct mapping between the file system's segment and SSD's parallel flash erase blocks. Then, the SSD-internal GC becomes unnecessary because the segment is written only via the append-only scheme. It can also reduce the SSD-internal resource by using coarse-grained mapping instead of 4KB-level fine-grained mapping.

The open-channel SSD (OC-SSD) [10] exposes its hardware geometry to the host. Therefore, the host can manage flash page allocation and logical-to-physical mapping. Because the host initiates GC to reclaim invalid flash pages, the IO latency can be controlled by the host. Recently, the OC-SSD 2.0 specification [7] defined the `vector chunk copy` command, which is an interface for copying data inside the SSD. Our `zone_compaction` command is similar to the vector chunk copy command of OC-SSD. However, we are targeting the ZNS SSD, which maintains a zone-level address mapping in the device, and we propose a new block allocation technique to utilize the copyback operation of flash memory.

Multi-streamed SSDs (MS-SSDs) [17] can be considered to use a grey-box model. The host can specify the stream ID of each write request, and the MS-SSD guarantees that different streams of data are written into different flash erase blocks. If the stream ID of each write request is assigned based on the lifetime of its data, each flash erase block will have data with similar lifetimes, and thus, the GC cost can be reduced. Whereas the MS-SSD is based on the legacy SSD managed by a fine-grained address mapping, our ZNS+ SSD uses a coarse-grained mapping, and it is GC-less owing to the sequential write-only constraint of the ZNS. Instead, our ZNS+-aware LFS writes different lifetimes of data at different segments to reduce the host-level segment compaction cost.

The ZNS is an industry standardization for the open-channel interface. The ZNS interface is beneficial for flash memory-based SSDs or shingled magnetic recording drives [14, 32] owing to the sequential write-only constraint of the storage media. Compared to the open-channel interface, the ZNS provides a higher level of abstraction. Instead of directly managing the physical flash chips of an SSD, the host accesses the sequential writable zones and uses special commands to change the write pointer and the state of each zone. F2FS [19] and btrfs [27] have been patched to support zoned block devices in Linux kernel 4.10 and 4.14, respectively. In the patched F2FS [3], the file system's segment size is set to be equal to the zone size. It also disables the in-place-update and threaded logging features that can cause non-sequential writes. The patched btrfs [2] modifies the block allocation algorithm and creates a new IO path to ensure sequential access within the zone. In ZoneFS [1], each zone is shown as an append-only file. Thus, user applications can access the zoned block device via a file interface. These ZNS-aware file systems will require zone compaction operations in any form. However, they do not have any optimization techniques to reduce the host-level zone compaction overhead.

One of the recent popular research issues is in-storage computing. By offloading the host-side operations to the SSD, we can reduce the computing load of the host system. When a *reduce* operation, such as filtering and counting, is offloaded, the data traffic between the host and the storage can be reduced significantly [13, 16]. Recently, the SSD-internal bandwidth has exceeded the host IO interface bandwidth as more flash chips are embedded for a large SSD capacity. Therefore, data traffic reduction by in-storage computing is highly beneficial for recent SSDs. Our ZNS+ is also a solution that can reduce data traffic of large-capacity SSDs.

6 Conclusion and Future Work

The current ZNS interface imposes a high storage reclaiming overhead on the host to simplify SSDs. To optimize the overall IO performance, it is important to place each storage management task in the most appropriate location and make the host and the SSD cooperate. To offload block copy operations to the SSD, we designed ZNS+, which supports in-storage zone compaction and sparse sequential overwrite. To utilize the new features of ZNS+, we also proposed ZNS+-aware file system techniques, i.e., the copyback-aware block allocation and the hybrid segment recycling. In future work, we plan to optimize various ZNS-aware file systems and applications to utilize the ZNS+. We will also study the in-storage copyback-aware block allocation and the partition-aware block allocation for multi-core SSDs to minimize the number of block copy operations between different partitions.

Acknowledgements

We thank our shepherd Haryadi S. Gunawi and the anonymous reviewers for their valuable feedback. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. IITP-2017-0-00914, Software Star Lab) and Samsung Electronics.

References

[1] Accessing zoned block devices with ZoneFS. https://lwn.net/Articles/794364/.

[2] Btrfs zoned block device support. https://lwn.net/ml/linux-btrfs/20180809180450.5091-1-naota@elisp.net/.

[3] F2FS-tools: zoned block device support. https://sourceforge.net/p/linux-f2fs/mailman/message/35456357/.

[4] Filebench. https://github.com/filebench/filebench/wiki.

[5] NVMe 1.4a - TP 4053 zoned namespaces. https://nvmexpress.org/wp-content/uploads/NVM-Express-1.4-Ratified-TPs-1.zip.

[6] ONFi: Open NAND Flash Interface. http://www.onfi.org/specifications.

[7] Open-Channel Solid State Drives Specification Revision 2.0. http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

[8] Percona-Lab/tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.

[9] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/.

[10] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 359–374, 2017.

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, et al. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*, pages 338–340, 2018.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM Symp. Cloud Computing*, pages 143–154, 2010.

[13] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *43rd International Symposium on Computer Architecture (ISCA'16)*, page 153–165, 2016.

[14] Weiping He and David H. C. Du. SMaRT: An Approach to Shingled Magnetic Recording Translation. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, page 121–133, 2017.

[15] Duwon Hong, Myungsuk Kim, Jisung Park, Myoungsoo Jung, and Jihong Kim. Improving SSD Performance Using Adaptive Restricted-Copyback Operations. In *IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA '19)*, 2019.

[16] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, August 2016.

[17] 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.

[18] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. DC-Store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation. In *18th USENIX Conference on File and Storage Technology (FAST'20)*, page 183–191, 2020.

[19] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 273–286, 2015.

[20] 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 *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 603–616, 2019.

[21] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 339–353, 2016.

[22] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 83–90, 2018.

[23] Yoohyuk Lim, Jaemin Lee, Cassiano Campes, and Euiseong Seo. Parity-stream separation and SLC/MLC convertible programming for life span and performance improvement of SSD RAIDs. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*, 2017.

[24] Hiroshi Maejima, Kazushige Kanda, Susumu Fujimura, Teruo Takagiwa, Susumu Ozawa, Jumpei Sato, Yoshihiko Shindo, Manabu Sato, Naoaki Kanagawa, Junji Musha, et al. A 512GB 3b/cell 3D flash memory on a 96-word-line-layer technology. In *2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*, pages 336–338, 2018.

[25] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM SIGOPS Operating Systems Review*, 31(5):238–251, 1997.

[26] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *3rd Annual Haifa Experimental Systems Conference (SYSTOR'10)*, 2010.

[27] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[29] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.

[30] Wei Wang and Tao Xie. PCFTL: A Plane-Centric Flash Translation Layer Utilizing Copy-Back Operations. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3420–3432, 2015.

[31] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[32] F. Wu, Z. Fan, M. Yang, B. Zhang, X. Ge, and D. H. C. Du. Performance Evaluation of Host Aware Shingled Magnetic Recording (HA-SMR) Drives. *IEEE Transactions on Computers*, 66(11):1932–1945, 2017.

[33] Fei Wu, Jiaona Zhou, Shunzhuo Wang, Yajuan Du, Chengmo Yang, and Changsheng Xie. FastGC: Accelerate garbage collection via an efficient copyback-based data migration in SSDs. In *55th Annual Design Automation Conference (DAC'18)*, 2018.

[34] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, 2014.

[35] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technology (FAST'20)*, page 121–136, 2020.

[36] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-Indirection for Flash-Based SSDs with Nameless Writes. In *10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling

Tanvir Ahmed Khan
University of Michigan

Ian Neal
University of Michigan

Gilles Pokam
Intel Corporation

Barzan Mozafari
University of Michigan

Baris Kasikci
University of Michigan

Abstract

Poor data locality hurts an application's performance. While compiler-based techniques have been proposed to improve data locality, they depend on heuristics, which can sometimes hurt performance. Therefore, developers typically find data locality issues via dynamic profiling and repair them manually. Alas, existing profiling techniques incur high overhead when used to identify data locality problems and cannot be deployed in production, where programs may exhibit previously-unseen performance problems.

We present selective profiling, a technique that locates data locality problems with low-enough overhead that is suitable for production use. To achieve low overhead, selective profiling gathers runtime execution information selectively and incrementally. Using selective profiling, we build DMon, a system that can automatically locate data locality problems in production, identify access patterns that hurt locality, and repair such patterns using targeted optimizations.

Thanks to selective profiling, DMon's profiling overhead is 1.36% on average, making it feasible for production use. DMon's targeted optimizations provide 16.83% speedup on average (up to 53.14%), compared to a baseline that uses the highest level of compiler optimization. DMon speeds up PostgreSQL, one of the most popular database systems, by 6.64% on average (up to 17.48%).

1 Introduction

Poor data locality is the root cause of many performance problems [6, 34, 48]. Rapidly increasing data footprints of modern applications due to heavily data-driven use cases (*e.g.*, analytics [109], machine learning [1], etc.) make matters worse, precipitating data locality problems further [6]. Recent work shows that up to 64% of all CPU cycles are lost due to poor data locality for widely used data center applications [90].

Although many compiler optimizations aim to eliminate data locality problems statically [3, 22, 23, 70, 71], such optimizations rely on compile-time heuristics, which may not accurately identify and repair problems that manifest dynamically at run time. In fact, as we (§6.2) and others [2, 15, 20, 27] demonstrate, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice.

To overcome the limitations of static optimizations, the systems community has invested substantial effort in developing dynamic profiling tools [28, 38, 57, 97, 102]. Dynamic profilers are capable of gathering detailed and more accurate execution information, which a developer can use to identify and resolve data locality problems.

Traditionally, existing dynamic profiling tools have been used offline, namely during testing and development, where test cases are designed to adequately represent real-world program behavior. However, due to the proliferation of cloud computing and mobile devices, programs exhibit vast variations in terms of how they execute and consume data in production [48, 84]. Consequently, it has become increasingly difficult for offline profiling to be representative of how programs behave in production settings.

Unfortunately, existing dynamic profilers incur considerable overheads when used to detect data locality issues, and therefore they are not suitable for production environments [13, 57, 60–62, 77, 78].

In this paper, we present *selective profiling*, a data locality profiling technique that not only accurately detects data locality problems, but also incurs low overhead, making it suitable for production deployment. Using selective profiling, we design DMon, a system that can automatically detect and eliminate data locality problems in production systems.

Selective profiling is a lightweight technique to continuously monitor production executions for symptoms of poor data locality (*e.g.*, frequent memory stalls, increased cache misses, etc.). As these high-level indicators of data locality problems are identified, selective profiling automatically transitions to incrementally monitoring more precise information about the source location and exact cause of the data locality problem—this is done by traversing a hierarchical abstraction we introduce, called the *data locality tree* (§3), which allows

DMon to monitor hardware events in a selective way to create an accurate profile at low run-time overhead.

After gathering the profile, DMon performs an offline analysis to identify common patterns of memory accesses. DMon then matches these patterns to a set of existing data locality optimizations (§4.1), which it primarily applies automatically, in a targeted manner (unlike static techniques). For cases where DMon cannot automatically apply an optimization, it provides detailed information about the locality problem to the developer, who can fix the problem manually; in our evaluation, this case occurs only once and the developer can apply DMon-suggested optimization with minimal effort (<10 LOC). We provide four optimization passes (§4.2) which DMon can use to automatically fix data locality problems and are sufficient for DMon to fix major data locality problems we identify across the systems we test in our evaluation (§6).

Selective profiling incurs 1.36% monitoring overhead on average, making it an ideal profiling technique for detecting data locality issues in production. The run-time overhead of selective profiling is significantly (*i.e.*, 9×) lower than that of the state-of-the-art data locality profiler [17, 68]. Overall, targeted optimizations performed by DMon for 13 applications deliver on average 16.83% (up to 53.14%) speedup. To show the effectiveness of DMon for large real-world systems, we applied DMon to `PostgreSQL` [92], a popular open-source database system, where DMon-guided optimizations provided on average 6.64% and up to 17.48% speedup across all 22 TPC-H [26] queries. Furthermore, the optimizations enabled by DMon provides 20% more speedup, on average, than optimizations provided by the same state-of-the-art profiler.

Overall, we make the following contributions:

- Selective profiling, a data locality profiling technique that automatically and incrementally monitors fine-grained execution information to accurately detect data locality problems with low overhead.
- DMon, a system that implements selective profiling to detect data locality problems in production systems. DMon automatically selects specific optimizations based on memory access patterns, and applies these well-known optimization techniques automatically in most cases.
- By evaluating DMon in the context of widely-used applications, we show that selective profiling can detect data locality issues in production with low overhead (1.36% on average). Moreover, we show that selective profile-guided targeted data locality optimizations provide significant performance speedup (16.83% on average, up to 53.14%).

We explain the key design challenge for accurately and efficiently detecting data locality problems in §2. We describe selective profiling in §3, DMon's design in §4, and DMon's implementation in §5. We evaluate DMon in §6, compare DMon to related work in §7, and conclude in §8.

2 Challenges

It is challenging to accurately pinpoint data locality problems, while incurring low run-time performance overhead.

Compiler-based static data locality optimizations [14, 70, 71, 82, 91] are appealing because they incur no run-time overhead. However, static techniques apply optimizations based on compile-time heuristics, which may not accurately identify program locations that suffer from poor locality at run time. In fact, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice [2, 15, 20, 27].

To demonstrate how compile-time heuristics can hurt performance, we use a compiler-based data prefetching technique [71] to improve data locality in two matrix decomposition benchmarks [104], `lu_cb` and `lu_ncb` from the PARSEC suite [12]. This optimization combines loop splitting and explicit data prefetching to increase data locality. Using the benchmarks' standard inputs, we determine that 50% of all the cache misses in `lu_cb` and `lu_ncb` stem from a single function, which we optimized using compiler-guided data prefetching [71]. The optimization provides a 19.4% speedup for `lu_ncb`, but yields a 19.85% slowdown for `lu_cb`. This occurs because, for `lu_ncb`, prefetching reduces all cache misses; however, for `lu_cb`, there was a dramatic increase in L2 cache misses despite a reduction in L1 and L3 cache misses.

Dynamic profilers can accurately pinpoint data locality problems [13, 57, 60–62, 77, 78], however, they impose considerable overhead (*i.e.*, >10% on average), as they track too much information: memory accesses, timestamps, cache events, etc. Consequently, existing data locality profilers are not deployed in production.

A potential remedy to the high overhead of existing profilers is statistical sampling, which can collect information with reasonable overhead [9]. For instance, the state-of-the-art Intel VTune profiler [85] samples information such as hardware and software performance counters, timestamps, program locations, and accessed memory addresses to gather the necessary information for detecting data locality issues.

Alas, even sampling is not enough to reduce the overhead incurred by popularly available profilers (*e.g.*, Intel VTune) to detect data locality problems to levels acceptable for production use. To assess the impact of sampling, we use the state-of-the-art profiler VTune to detect the data locality issues in our evaluation targets. Despite sampling-based data collection, VTune still incurs 26% overhead on average (and up to 60%), which is unacceptable for production settings.

We argue that not only the monitored execution information must be deliberately chosen to only pertain to data locality problems, but monitoring must occur incrementally, only when there are increasingly clear signs of poor data locality. Next, we explain how selective profiling achieves this.

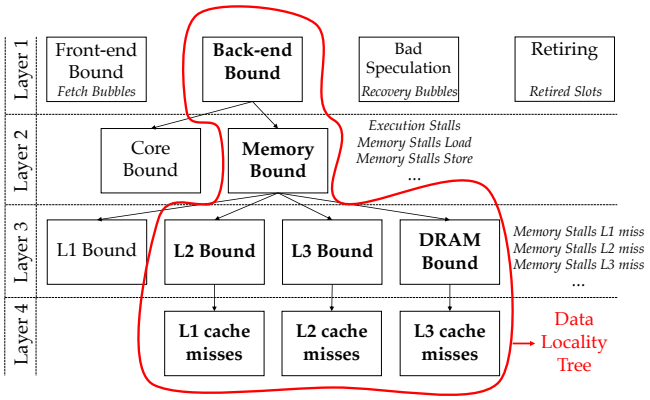


Figure 1: The locality tree abstraction. Performance events that pertain to each tree node are in *italic*. There are no dedicated events to determine if a program is back-end bound. Instead, selective profiling subtracts from total stalls the sum of the stalls that cause other bottlenecks at layer 1 to determine if an execution is back-end bound.

3 Selective Profiling

Selective profiling is a monitoring technique that incrementally monitors more detailed, yet more targeted, run-time information to identify data locality problems. Next, we discuss the three key components of selective profiling: (1) Targeted Monitoring, (2) Incremental Monitoring, and (3) Sampling.

3.1 Targeted Monitoring

Unlike existing offline profilers [57, 97, 98, 102, 106] that monitor many hardware events and information such as program locations, selective profiling needs to carefully choose which information to monitor in order to accurately and efficiently detect data locality problems. A straw-man approach is to only monitor events such as data cache misses, which are directly related to data locality problems. However, simply monitoring data cache misses in isolation can be misleading. For instance, a seemingly large number of data cache misses may have no impact on the performance of an application that spends a lot of time fetching instructions to execute (a common theme in modern Web services [8, 48]).

Selective profiling monitors a select group of hardware events that allow it to determine if the execution of a program is bounded by a subset of those events that we call the *data locality tree*. As shown in Fig. 1, the data locality tree is a hierarchical abstraction of data locality-related performance events from Intel's Top-Down methodology [106]. The Top-Down methodology provides a breakdown of performance events in Intel CPUs, which a developer can use as a guideline to navigate their manual profiling efforts. However, unlike Top-Down, selective profiling automatically transitions from one layer to another, incrementally monitoring more events at each layer of the tree, as increasing evidence of data locality issues is observed at run time.

At layer 1, selective profiling determines whether the execution is back-end bound—*i.e.*, spends a large portion of the time either in CPU execution (CPU bound) or accessing memory (memory bound). At layer 1, a program can also be front-end bound (*i.e.*, fetching instructions), incurring misspeculations, or retiring instructions. For executions that are back-end bound, selective profiling determines whether they are processor-core bound or memory bound in layer 2.

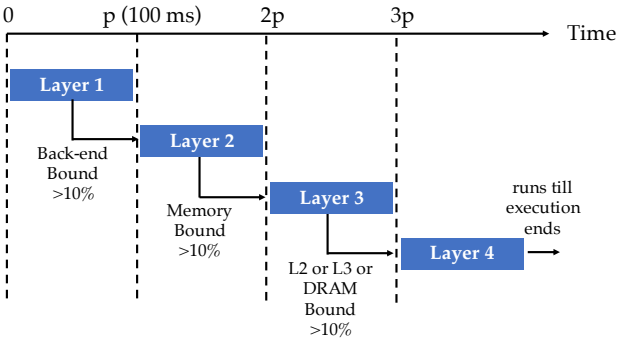


Figure 2: Incremental monitoring

If an execution is memory bound in layer 2, selective profiling monitors events that provide a breakdown of the execution into 4 categories in layer 3. Of those 4 categories, only 3 are related to data locality problems: L2 bound and L3 bound represent the time spent accessing the L2 and the L3 cache, respectively; "DRAM bound" represents the time spent accessing the DRAM. If a program is L1 bound, the data or instructions that the program uses are already as close to the processor as possible and it is hard to improve data locality further. In such cases, the program may have other performance problems, such as false sharing [93] or lock contention [87].

Selective profiling also tracks information to map performance problems back to code. In layer 4, selective profiling records program location information along with hardware events. For example, if a program is L2 bound, selective profiling records L1 cache misses and the location of the instruction causing the miss. By locating and reducing L1 cache misses, the execution time will potentially not be L2 bound, and the locality problem will likely be fixed. Similarly, if a program is L3 or DRAM bound, selective profiling records L2 and L3 cache misses and associated program locations, respectively.

3.2 Incremental Monitoring

Unfortunately, merely restricting the scope of monitored performance events to the data locality tree is not sufficient for low overhead monitoring of data locality issues. Thus, selective profiling instead adopts an incremental monitoring approach. This approach increases the amount of information gathered at run time to efficiently identify program locations that may have a locality problem.

Fig. 2 shows the details of incremental monitoring. By default, selective profiling monitors the hardware events that provide the layer 1 breakdown. Selective profiling only transitions to monitoring layer 2 events if the execution is back-end bound for at least 10% of a time-slice p (100ms by default).

We use 10% as the default threshold, which we empirically determine to be a reasonable threshold (§6.4). We also choose 100ms as a reasonable time-slice for our programs, since the shortest execution across our benchmarks was 1 second and the longest was 2867 seconds. Nonetheless, the percentage and monitoring periods are both configurable. We explore the sensitivity of our results to all these parameters in §6.4.

If selective profiling determines that the execution is also memory bound for at least 10% of the same interval p, it starts monitoring layer 3 events. If selective profiling determines that the execution is L2, L3, or DRAM bound for at least 10% of the same interval p, it transitions to layer 4. Selective profiling then gathers L1, L2, and L3 cache miss events and program locations where the misses occur.

Incremental monitoring is key to ensuring selective profiling's low performance overhead. Successive layers are more costly to monitor as they must count more events—for example, layer 2 requires counting 3× more hardware performance events than layer 1. However, unless selective profiling determines that an execution is back-end bound, it only needs to monitor events at layer 1. As shown in §6.1, only monitoring layer 1 events incurs a negligible overhead (0.7% on average).

Programs can go through phases of different locality issues (*e.g.*, L2 cache misses in one phase and L3 cache misses in another phase). Selective profiling can pinpoint the root cause of the locality problem for each phase, provided the duration of a given phase is at least $4p$ (where p is the duration of selective profiling's time-slice, per layer). If this time-slice is too long, selective profiling may miss some short-running phases. The time slice is configurable. We empirically determine that a time slice of 100ms is effective in practice (§6.4).

3.3 Sampling

In addition to targeted and incremental monitoring, selective profiling also employs sampling at layer 4 for recording L1, L2, and L3 cache misses to further reduce the overhead. Although sampling can reduce run-time overhead, it can also reduce the coverage of data locality issues that selective profiling detects if the sampling period is too high. We define coverage as the ratio of the number of locality issues detected with a given sampling rate to the number of locality issues detected with the highest possible sampling rate.

By default, selective profiling uses a conservative sampling period of 1000 (1 sample per 1000 events), which we have empirically found to yield high coverage (97%, discussed in §6.4) in detecting locality problems across the 13 benchmarks we evaluated. The developer, however, can use a lower sampling period (up to 1 sample per 100 events, as allowed per Linux's `perf` interface). We analyze the coverage versus overhead trade-off of different sampling periods in §6.4.

Selective profiling does not apply sampling in layers 1–3 since sampling reduces coverage. Moreover, in layers 1–3, selective profiling's incremental monitoring reduces the overhead to a negligible amount in all tested applications (on average 1.36%). Therefore, selective profiling does not need to apply sampling at those layers. However, if the overhead of the first three layers is high, selective profiling can optionally enable sampling at those layers as well.

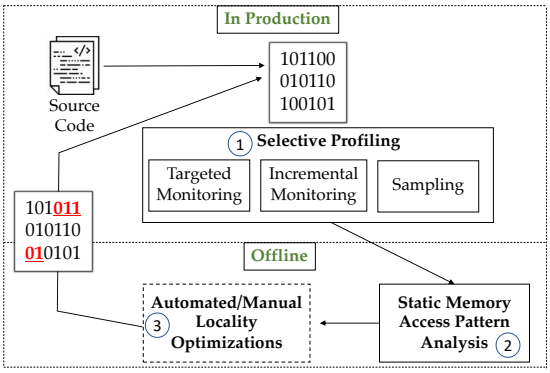


Figure 3: How DMon leverages selective profiling to detect and repair data locality problems.

Now, we describe how data locality information collected via selective profiling can be used to guide automated and manual profile-guided optimizations using DMon.

4 DMon

Selective profiling detects program locations with poor data locality in production. DMon analyzes these locations offline to identify the data access patterns causing data locality issues. Based on the recognized access patterns, DMon applies existing compiler optimizations only to these program locations in a targeted manner to improve data locality. We offer four such optimizations which we describe in §4.2. These optimizations can be automatically applied in most cases for C/C++ applications; for applications written in other programming languages, selective profiling results can still enable manual optimizations (§6.3).

Fig. 3 shows how DMon employs selective profiling to identify and eliminate data locality issues. In step ①, DMon monitors programs in production to determine whether they suffer from poor locality using selective profiling.

Steps ②–③ happen offline, during recompilation. In step ②, DMon determines the memory access patterns that are causing poor data locality (§4.1). In step ③, based on the identified access patterns, either profile-guided automatic optimizations or manual optimizations can be performed to improve data locality (§4.2). The optimized program is then rebuilt and redeployed in production.

4.1 Static Memory Access Pattern Analysis

Once selective profiling identifies memory access instructions that suffer from poor locality in production, DMon analyzes the corresponding program locations offline to determine the cause of the problems. DMon only analyzes memory access instructions that incur more than 10% of the total cache miss events sampled in layer 4 of selective profiling.

Table 1: Four common memory access patterns that cause data locality problems in many applications. Here, we show their examples from the PARSEC [12] benchmark suite.

Benchmark	Code snippet	Access pattern
lu_ncb	`a[i] += alpha*b[i];`	Direct Addressing

radix	`this_key = key_from[i] & bb;`
	`this_key = this_key >> shiftnum;`
	`tmp = rank_ff_mynum[this_key];`
		Indirect Addressing

radiosity	`while(int_list)`
	`{`
	`  if(int_list->dst==inter->dst)return(1);`
	`  int_list = int_list->next ;`
	`}`
		Unbalanced Access

dedup	`if(LstElmnt->seq.l2num > H->Elmnts[Child]->seq.l2num){`	Pointer Chasing

To determine the patterns of data locality issues, we initially analyze the results of selective profiling manually for the benchmarks from the popular PARSEC [12] benchmark suite. Based on our manual analysis of program statements causing data locality issues, we identify four key memory access patterns that can lead to poor data locality. Table 1 shows one example of each of these memory access patterns that cause poor data locality. Perhaps unsurprisingly, all the accesses that contribute significantly to poor data locality are in loops that execute many times and access a relatively large amount of data compared to other memory access operations in the application. These four memory access patterns also cause data locality problems in a diverse set of real-world applications (as we show in §6.3).

For `lu_ncb`, most cache misses that hurt program performance happen while accessing arrays in a loop. Since the loop induction variable (`i`) is directly used to index those arrays, we call this pattern *direct addressing*. For `radix`, the loop induction variable (`i`) is used to index an auxiliary array to load an intermediate value (`this_key`). The loaded intermediate value is used as index while accessing another array, and the last access suffers from poor data locality. We categorize this pattern as *indirect addressing*.

For `radiosity`, most cache misses occur in a while loop, where two member variables (`dst` and `next`) of a structure (`int_list`) are accessed repeatedly. We determine that this structure also contains four other member variables not accessed in this loop. Since only accessing a subset of all member variables causes cache misses, we call this access pattern as *unbalanced access*. Finally, for `dedup`, locality suffers while accessing a chain of structure pointers (pointers `H`, `Elmnts[Child]`, and `seq`, and finally a member variable `l2num`) in a loop. We denote this pattern as *pointer chasing*.

Based on findings of these manual observations, we design the static memory access pattern analysis component of DMon, as shown in Fig. 4. Although DMon's pattern detection is inspired by the manual analysis of locality issues in PARSEC, we show in our evaluation that the patterns DMon identifies generalize to a broad set of systems (§6.2 and §6.3). In particular, the four patterns of poor locality constitute the root causes of all the data locality problems we discover in nine other benchmarks that we had not studied previously.

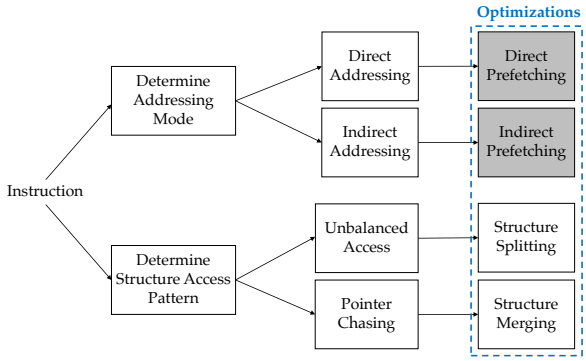


Figure 4: Static memory access pattern analysis in DMon and their corresponding optimizations. Shaded optimizations are mutually exclusive.

As shown in Fig. 4, DMon determines the addressing mode of the memory instruction (*i.e.*, direct or indirect addressing). If the access is made to a structure instance, DMon also determines the type of the access (*i.e.*, unbalanced access and pointer chasing). We discuss each analysis next.

Addressing mode. DMon's static analysis checks if the instruction uses direct or indirect addressing. Here, direct addressing occurs if the computation of the accessed location does not involve another memory address (*e.g.*,`for(i=...) a[i]`). Conversely, indirect memory addressing occurs if the computation of the accessed location involves computing another memory address (*e.g.*,`for(i=...) a[b[i]]`).

Structure access pattern. In addition to determining the addressing mode, DMon's static analysis checks to see if the instruction accesses a member of a structure. DMon does this by mapping the instruction to the compiler intermediate representation and checking if it accesses a structure field. DMon searches for two patterns when a structure member is accessed, namely *unbalanced access* and *pointer chasing*.

DMon concludes that there is an unbalanced access pattern, when accesses to only a *subset* of member variables incur a large fraction of cache misses. Pointer chasing occurs when the accessed memory location belongs to a hierarchy of nested structures (*e.g.*, `A->B->C`).

4.2 Optimizations Implemented in DMon

To show the usefulness of selective profiling, we implement four profile-guided data locality optimization passes using LLVM [56] for C/C++ programs. Our passes optimize the four patterns of poor data locality that DMon identifies. For applications written in other languages, selective profiling results can be used to apply manual optimizations (§6.3).

As shown in Fig. 4, DMon recommends applying a specific optimization technique based on the addressing mode and the structure access patterns of the memory access instruction. While these optimizations are well-known and usually applied statically, selective profiling information enables the targeted

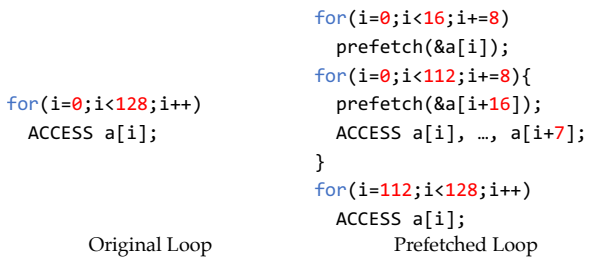


Figure 5: Software prefetching for direct memory access, adapted from [71]. The induction variable is of type `int`. The `prefetch` instruction prefetches one cache line (64 bytes).

application of these optimizations to where they are absolutely needed in a program. As we show in §6.2, DMon-enabled targeted profile-guided optimizations outperform purely static optimizations by 10% on average.

Direct prefetching. The first optimization we implement uses direct prefetching [71] to fix locality problems that stem from memory accesses that use direct addressing. Direct prefetching fetches the cache lines that a program will access in the near future into the cache to improve data locality.

At a high level, direct prefetching works by splitting each loop suffering from poor data locality into three loops, as shown in Fig. 5. The first loop is responsible for prefetching the initial cache line that contains the data accessed by the loop. The second loop starts prefetching the next cache line(s). It also simultaneously performs the original computation that was carried out in the original loop, starting with the first prefetched cache line. The third and last loop completes the computation using the last prefetched cache line.

Direct prefetching can be applied based on compile-time heuristics only. However, this can cause significant performance degradation [29], as we also show in our evaluation (§6.2). This happens because these heuristics might (1) bloat the code footprint by adding unnecessary prefetching instructions (*e.g.*, for lines that would anyways be prefetched by the hardware prefetcher), and (2) cause cache pollution by prefetching data that is not frequently-accessed.

Direct prefetching can also be applied in hardware with popular hardware prefetchers including next-line and stride prefetchers that most modern processors supposedly employ [42, 94]. However, DMon finds that many directly addressed memory accesses suffer from poor data locality, because the underlying hardware prefetchers can not prefetch the cache lines in a timely manner. This is because prefetchers work in a reactive manner, *i.e.*, it takes several iterations for the hardware prefetcher to detect the pattern and start prefetching, but if prefetching is done with explicit instructions, the performance benefits are immediate.

Instead of applying direct prefetching based on compile-time heuristics, our pass only applies it to program locations where DMon identifies that direct addressing access pattern is causing poor data locality.

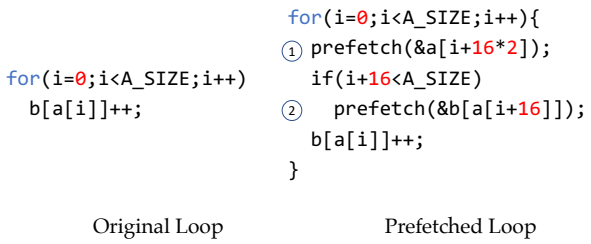


Figure 6: Software prefetching for indirect memory access, adapted from [3].

Indirect prefetching. Our second optimization uses indirect prefetching [3], which is similar to direct prefetching in that it brings data that will soon be used into the cache. Unlike direct prefetching, indirect prefetching also has to prefetch one additional cache line per each level of indirection.

Fig. 6 shows an example of indirect prefetching. Here, the original loop increments elements in an array, `b`. However, the index of the array `b` is computed using another array, `a`. The loop on the right side prefetches the cache line containing the elements of `b` that will be accessed in the near future (prefetch ②). Prefetching the elements of `b` requires accessing the elements of `a`. Thus, to prefetch the elements of `b`, we need to (1) have an array boundary check, and (2) also prefetch the cache line containing the elements of `a` (prefetch ①).

Structure splitting. The third optimization, structure splitting, moves infrequently-accessed members of a structure with a pointer to a new structure that only contains those members. Structure splitting is beneficial only when the total size of infrequently-accessed member(s) is larger than the pointer size. Thus, the size of the original structure is reduced, fitting into fewer cache lines. During memory access pattern analysis, if DMon detects that an unbalanced access pattern (*i.e.*, a subset of structure members are accessed more frequently than others) to members of a structure is causing poor locality, structure splitting is an appropriate optimization.

Fig. 7 shows an example of structure splitting. Here, before structure splitting, the structure `S` has three members (`a`, `b`, `c`) of types `A`, `B`, `C`, respectively. In the original program, an instance of `S` spans two cache lines. Both cache lines need to be accessed each time the program accesses an instance of `S`. For example, if neither of these cache lines is present in the L1 cache, the program will incur two L1 cache misses.

After structure splitting, the new structure `S'` fits in a single cache line (Cache Line 1) because the infrequently-accessed member `b` is moved into a new structure `S2`, residing in its own cache line (Cache Line 2). Consequently, when the program accesses an instance of `S`, it will usually only need to access the cache line (Cache Line 1) containing the frequently-accessed members (`a`, `c`), which would incur a single L1 cache miss (rather than two).

Structure splitting has been previously explored [22] in type-safe languages (*e.g.*, Java). However, implementing structure splitting in a type-unsafe language (we target C/C++) is more challenging. This is because structure splitting needs to

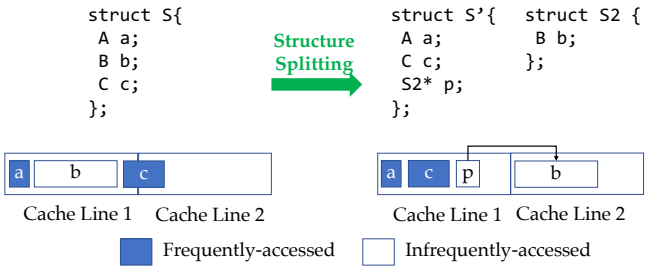


Figure 7: Structure splitting, example adapted from [22].

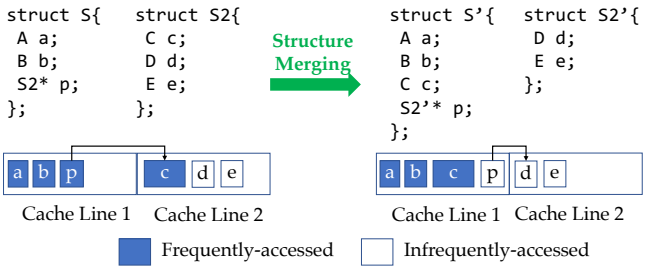


Figure 8: Structure merging example.

ensure that the program continues operating correctly when the layout of the structure is modified. More specifically, all the instructions that used to refer to the old layout need to be updated to refer to the new layout.

In our optimization pass, we addresses this challenge using a complete, interprocedural, inclusion-based pointer analysis [5] that can determine all instructions that could possibly access the split structures. As shown in §6.2, this optimization can automatically be applied in all but one of the benchmarks.

Structure merging. The final optimization, structure merging, is the inverse of structure splitting as it replaces a frequently-accessed pointer member of a structure with the data that the pointer references. The key idea is to eliminate the pointer chasing pattern that DMon identifies by removing a level of indirection for frequently-accessed elements.

Fig. 8 shows an example of structure merging. Before merging, the structure `S` has three members (`a, b, p`) of types `A, B, S2*`, respectively. The instance of `S` resides in the first cache line, and the pointer `p` points to an instance of structure `S2` that resides in the second cache line. The size of `a`, `b`, and `c` is such that they can all fit in one cache line. If `c` is accessed as frequently as `a` and `b`, then data locality can be improved by merging these two structures into one. This structure merge will also bypass one memory access (`S'->C` instead of `S->S2->C`). Structure merging only combines member variables across different structure types and hence does not perform exhaustive data structure conversions (*e.g.*, transforming a linked list into an array) [22, 23].

DMon employs structure merging conservatively so that it will only be applied if soundness can be guaranteed. In other words, DMon applies this optimization only if all updates via the structure pointer can be safely redirected (*e.g.*, in Fig. 8, all changes to `S->S2->C` could be replaced by `S'->C`). To ensure this, structure merging also uses the same pointer analysis [5] that structure splitting uses.

Other optimizations.. DMon can be easily extended to accommodate additional optimizations if needed to fix different patterns of memory accesses which cause data locality problems. For example, DMon can work as a framework to apply optimizations like loop reordering, blocking, tiling, and strip mining in a profile-guided manner. However, many of these optimizations require expensive memory access trace collection which can not be deployed in production due to high overheads [64]. In the future, we intend to explore how these optimizations can be applied based on more efficient profiling.

5 Implementation

DMon's selective profiling prototype is implemented for Intel processors. In particular, selective profiling relies on the Linux `perf` [97] interface for profiling hardware events in layers 1–4 (§3). We initially build the benchmarks using debug information and the highest level of compiler optimization (`-O3`), and then use the `strip` utility [101] to remove the debug information. During in-production monitoring, selective profiling records the program counter for each sampled cache miss event in layer 4. To efficiently deal with multi-threaded applications, selective profiling maintains a per-thread buffer (2MB per thread) to record the program counters. When the buffer gets full, the previous samples get overwritten. Offline, DMon uses the program counter, the stripped debug information, and the program binary to find the source code location where a cache miss occurred in production.

We implement DMon's optimizations in the LLVM [56] compiler framework. We use `clang` [99] to generate the LLVM intermediate representation (IR) that the optimization passes of DMon can operate on. The optimizations rely on the program's debug information to map the source code location to LLVM IR, because a 1-to-1 mapping between machine code and LLVM IR does not exist.

Similar to other state-of-the-art profile-guided optimization techniques [17, 68], DMon's use of debug information for mapping machine code to LLVM and locating code locations to optimize can introduce inaccuracies. This happens due to optimizations such as inlining. Although it is possible to improve the accuracy of such mapping using more invasive instrumentation and tracing [7], this would be prohibitively costly for production usage [48]. In our evaluation (§6), we show that the accuracy provided by debug information can lead to substantial speedup.

The optimizations for structure splitting and structure merging use a whole-program pointer analysis [19].

6 Evaluation

In this section, we first evaluate the efficiency of selective profiling by measuring its run-time monitoring overhead. Then, we evaluate the effectiveness of DMon by showing the extent to which fixing the locality problems detected by

DMon improves performance of popular benchmarks. Next, we evaluate selective profiling's generality by applying it to widely-used real-world applications. Finally, we perform sensitivity studies to evaluate how DMon's overhead and detection results vary in response to changes of the different system parameters of DMon.

Software. All experiments are conducted in Ubuntu 18.04 (kernel version 4.15.0-46-generic). The static compiler analyses are implemented in LLVM (7.0.0) on bitcode emitted by clang. Therefore, we use `clang 7` as the baseline compiler.

Hardware. We use a 20-core 2.2 GHz Intel Xeon NUMA (with 2 sockets) machine, with 64 KB of L1-cache (32 KB instruction and 32 KB data), 1024 KB of L2-cache, 14 MB of L3-cache (shared across the same NUMA node), and 96 GB of RAM. Like most Intel processors, each core in the machine uses two hardware prefetchers (next-line and sequential load history driven prefetchers) in the L1 data cache and two hardware prefetchers (adjacent cache line and streaming prefetchers) in the L2 cache [42, 94]. We configure multi-threaded applications and benchmarks to run with 8 threads.

Benchmarks. We use a combination of benchmarks and real-world programs that have been widely used in prior performance profiling and optimization work. In particular, we use all 12 benchmarks from the PARSEC [12] suite, all 11 benchmarks from the SPLASH-2X [103] suite, and all 3 benchmarks written in C from the NPB [10] suite, as well as `HashJoin`, `RandomAccess`, `kcstashtest`, and `DIS`, which are programs with poor data locality from other popular benchmark suites [11, 24, 63, 73]. We also study one of the most popular and heavily-optimized open-source databases, `PostgreSQL` [81], running the TPC-H analytical workload [26]. Finally, we study real-world applications from the Renaissance benchmark suite [83].

Metrics. In all our plots, we report speedup numbers as the ratio between the execution time of the original application compiled with the highest level of optimization (`-O3`) and its run time after applying DMon-guided optimizations. Negative speedup denotes slow-down. Similarly, we report selective profiling overhead as the percentage increase in benchmark execution time while enabling selective profiling. We report performance data as the average of 25 runs in all experiments.

6.1 Selective Profiling Efficiency

We evaluate the selective profiling efficiency by studying the overhead selective profiling incurs during dynamic detection of locality problems. Fig. 9 shows this overhead. We present results for all the benchmarks we evaluated, including the ones for which selective profiling did not find locality optimization opportunities. For each benchmark, we present the overhead of each layer of monitoring (1–4) that selective profiling employs. Since, selective profiling monitors only one layer at a time, the effective overhead for a given program is less than the maximum overhead across four layers.

Across all layers and benchmarks, selective profiling incurs up to 4.92% overhead, and on average only 1.36% overhead. On average, selective profiling incurs an overhead of 0.7% in layer 1, an overhead of 1.5% in layer 2, an overhead of 2.5% in layer 3, and an overhead of 2% in layer 4. For benchmarks that do not have locality problems, layers 2–4 are never triggered.

In only 3 out of all 28 benchmarks, selective profiling incurs more than 3% overhead: `IS` (4.6%), `kcstashtest` (4.2%), and `HashJoin` (4.9%). However, as we detail in §6.2, optimizations suggested by DMon also provide greater speedups for these benchmarks than for others (`IS` 30.3%, `kcstashtest` 32.4%, and `HashJoin` 53.1%—compared to 16.83% average speedup enabled by DMon). These benchmarks suffer the most from poor locality, and consequently, selective profiling incurs more overhead to pinpoint the root cause of those problems.

6.2 Effectiveness

We evaluate the effectiveness of DMon by studying (1) data locality problems detected by DMon, (2) speedups provided by DMon-guided optimizations, (3) comparison of the speedups provided by DMon-guided optimizations to the speedups provided by Google's AutoFDO [17]—the state-of-the-art profile-guided locality optimization approach, (4) whether DMon-guided optimizations generalize across different program inputs, and (5) the overhead on compilation times due to DMon-guided optimizations.

Locality issues detected by DMon. Table 2 summarizes the data locality problems that DMon detects. For brevity, Table 2 omits benchmarks where less than 10% of the execution time is bounded by locality problems, as these benchmarks could not benefit from eliminating locality improvements. We also omit these benchmarks in our average performance numbers.

Additionally, Table 2 shows the most prominent level of the memory hierarchy for the locality issues detected by selective profiling. Note that, in many cases, DRAM accesses constitute the locality bottlenecks. This is expected, since the highest-latency memory access instructions are served from DRAM. Finally, Table 2 also reports the program locations (as "file": "line number") that suffer the most from poor locality, along with the optimizations DMon recommends in each case.

As shown, DMon successfully identifies locality problems and suggests appropriate optimizations in each case. In all cases but one (`fmm`), DMon applies optimizations automatically. For `fmm`, while the direct prefetching is applied automatically, structure splitting cannot be applied automatically. This is because, due to excessive type casts, the compile-time optimization cannot exactly determine which program statements may access the modified structure, and therefore cannot automatically update such statements. Nonetheless, since DMon points the developer to the exact source of the locality issue in `fmm`, the fix can easily be applied manually with an 8 LOC update. Moreover, structure splitting and merging can be applied automatically for other applications (`dedup` and `radiosity`)

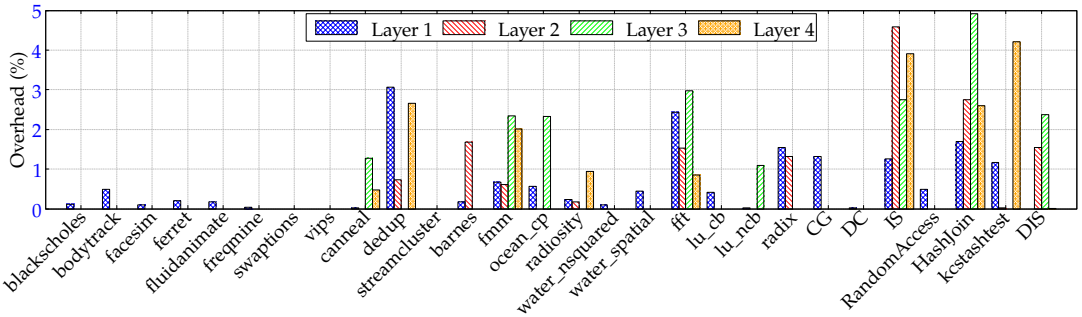


Figure 9: Monitoring overhead of selective profiling (All $\sigma < 0.02\mu$).

Table 2: DMon's detection results of locality problems.

Benchmark	Execution time (seconds)	Memory hierarchy bottleneck	Program location	Optimization	Automated fix?
canneal	71.8	L3, DRAM	netlist_elem.cpp: 80	Direct Prefetching	Yes
dedup	5.1	DRAM	binheap.c: 93	Structure Merging	Yes
fmm	18.8	DRAM	interactions.C: 169	Structure Splitting	No
				Direct Prefetching	Yes
ocean_cp	36.2	L2, L3, DRAM	multi.C: 273	Direct Prefetching	Yes
radiosity	95.8	L2, L3	rad_tools.C: 399	Structure Splitting	Yes
fft	1.2	DRAM	fft.C: 765	Direct Prefetching	Yes
lu_ncb	47.8	L3, DRAM	lu.C: 466	Direct Prefetching	Yes
radix	6.1	L2, L3, DRAM	radix.C: 624	Indirect Prefetching	Yes
IS	1	L3, DRAM	is.c: 392	Indirect Prefetching	Yes
RandomAccess	607.1	DRAM	randacc.c: 125	Indirect Prefetching	Yes
HashJoin	2867.3	L3, DRAM	npj2epb.c: 300	Indirect Prefetching	Yes
kcstashtest	3.20	L2, L3, DRAM	kcstashdb.h: 146	Direct Prefetching	Yes
DIS	165.3	L2, L3, DRAM	transitive.c: 107	Direct Prefetching	Yes

Table 3: Speedup comparison between DMon and compile-time optimizations.

Benchmark	Speedup provided by compile-time optimizations (%)	Speedup provided by DMon (%)
canneal	-7.90	1.07
dedup	-18.90	3.65
fmm	2.83	2.68
ocean_cp	-1.06	2.90
radiosity	-7.14	11.21
fft	1.11	4.57
lu_ncb	3.49	19.40
radix	0.96	1.85
IS	30.52	30.29
RandomAccess	38.83	47.67
HashJoin	9.74	53.14
kcstashtest	37.41	32.39
DIS	-0.28	7.93

where the automatic transformation can identify and update all statements pointing to the split and merged structures.

Speedup. Table 3 compares the speedup provided by the DMon-guided optimizations. Optimizations guided by DMon provide up to 53.14% and on average 16.83% (8% median) speedup. To study the impact of the targeted optimizations guided by selective profiling results, we also report the speedup achieved by the same optimizations if they are applied indiscriminately (*i.e.*, in a non-targeted way), through purely-static compiler passes [3, 71].

As shown in Table 3, DMon-guided optimizations outperform compile-time optimizations in 10/13 benchmarks. Crucially, static optimizations hurt performance in 5/13 cases due to being applied too broadly (with no runtime information), and therefore causing outcomes such as cache pollution and code bloat. DMon-guided optimizations always improve the performance. In 3/13 benchmarks where static optimizations outperform DMon-guided optimizations, the margin is $\leq 5\%$ which can be reduced by reducing the incremental monitoring threshold (default, 10%) of selective profiling.

Comparison against Google AutoFDO. We compare the speedup provided by DMon-guided optimizations to that of Google's AutoFDO [17], the state-of-the-art profile guided

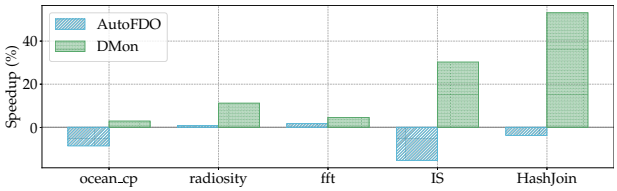


Figure 10: Speedup comparison to AutoFDO (All $\sigma < 0.09\mu$)

optimization technique. AutoFDO has limited data locality optimization capabilities [68]; our comparison is thus limited to five benchmarks for which AutoFDO can optimize locality.

We compare the speedup provided by DMon-guided optimizations to the speedup provided by AutoFDO in Fig. 10. As shown, DMon-guided optimizations provide better speedup than AutoFDO for all five benchmarks. This is because AutoFDO could only identify data locality problems that can be solved by performing direct prefetching optimizations. By contrast, DMon can identify other data locality issues that can be addressed by additional locality optimizations (*i.e.*, indirect prefetching, structure splitting, and structure merging).

For example, AutoFDO's direct prefetching slows down the execution of `IS` by 15%, while DMon-guided indirect prefetching provides a 30% speedup. Even for cases where both DMon

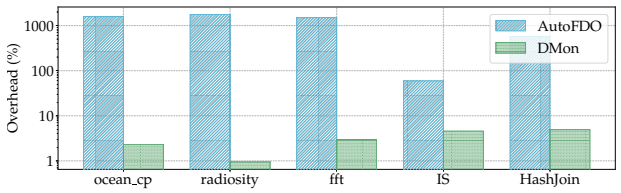


Figure 11: Overhead comparison to AutoFDO (All $\sigma < 0.07\mu$)

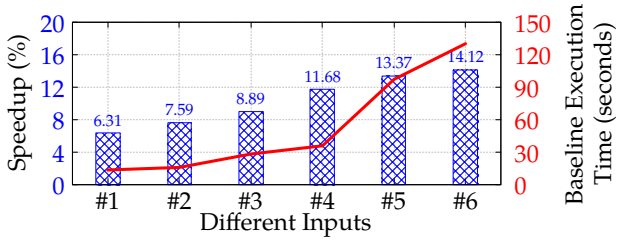


Figure 12: DMon-generated optimization after observing input #4 generalizes to unseen inputs (All $\sigma < 0.01\mu$).

and AutoFDO suggest direct prefetching (*e.g.*, `ocean_cp`), DMon-guided optimizations outperform AutoFDO, because, unlike AutoFDO, DMon provides hints as to where (*e.g.*, L1, L2, or L3) the cache line should be prefetched.

We compare selective profiling overhead against AutoFDO's profiling overheads in Fig. 11. For the 5 benchmarks in this study, selective profiling incurs 3.3% mean overhead, whereas AutoFDO incurs 978% mean overhead, making the latter unsuitable for production use.

Generalization across program inputs. Profile-guided optimizations perform best when the application is optimized with a profile that is representative of the application's common behavior [17, 79, 95]. DMon-guided fixes also generalize if the program shows similar data locality behavior across different inputs. Therefore, we evaluate DMon's generality across different program inputs for 9 benchmarks. These program inputs vary widely both in terms of input size (from megabytes to gigabytes) as well as execution times needed to process the input (from seconds to minutes).

We report a detailed case study using the `radiosity` benchmark to determine how well the locality optimizations suggested by DMon generalize to different inputs. We choose this benchmark because the fix suggested by DMon is structure splitting—an optimization that modifies the data layout, and hence has the potential to be affected by changing program inputs. Fig. 12 shows the speedup provided by DMon-guided optimizations for `radiosity` for various input sizes.

Here, for brevity, we refer to different input sizes using "#1" through "#6". DMon only observes the execution for the randomly selected input #4. After observing input #4, DMon-guided optimizations are applied. Then, all inputs are rerun with the newly-optimized program, with the results of this run reported in Fig. 12. As shown, the optimization suggested by DMon generalizes well to other inputs, providing considerable

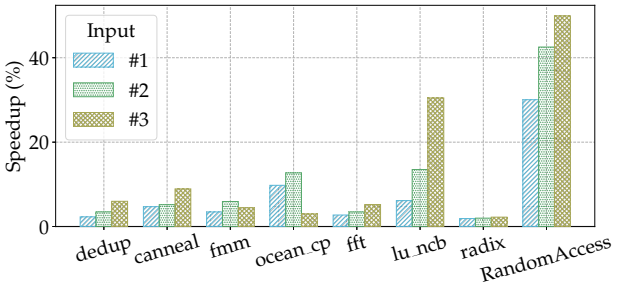


Figure 13: Input generalization (All $\sigma < 0.04\mu$)

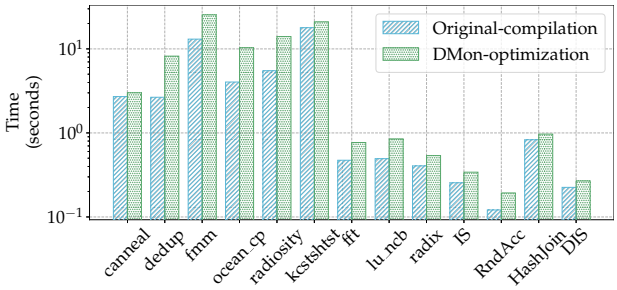


Figure 14: Overhead of DMon-guided optimizations compared to baseline compilation time ($\sigma < 0.1\mu$, log-scaled y).

speedups in each case. Longer executions that use *larger* inputs benefit more from optimizations.

Fig. 13 shows how DMon-guided optimizations improve data locality for unobserved inputs of several other benchmarks. Here, we include all benchmarks with at least 3 inputs. Across all evaluation targets, we find that data locality behavior follows a similar trend for different inputs. Hence, DMon's fixes generalize to different inputs for these benchmarks.

Recompilation overhead. We evaluate the offline recompilation overhead while applying DMon-guided optimizations, though this does not impact the production overhead. We perform this experiment, because automated structure splitting and merging require pointer analysis, which is known to be expensive [55]. However, the specific pointer analysis we employ is flow- and context- insensitive and scales well [40].

Fig. 14 shows the offline compilation overhead incurred by our DMon-guided optimizations on top of the baseline compilation overhead (`clang`). On average, DMon-guided optimizations incur 72% more overhead. However, the optimization takes on average less than 7 seconds and is no longer than 26 seconds. Even for large applications (*e.g.*, PostgreSQL [92] code base has over 1M LOC), the analysis takes 307 seconds. For an offline process, we believe these durations are reasonable and on par with standard compiler transformations that use whole-program pointer analysis. Moreover, this is a one-time compile-time overhead and will be amortized for long-running applications (*e.g.*, data-center applications that are compiled once but run on thousands of servers for days). Finally, structure splitting and merging can be applied manually if the cost of pointer analysis is deemed prohibitive.

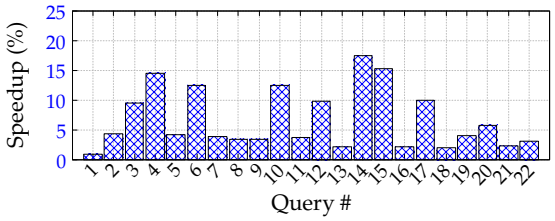


Figure 15: Speedup due to DMon-guided optimizations for 22 TPC-Hqueries on `PostgreSQL` (All $\sigma < 4.53\%$ of μ).

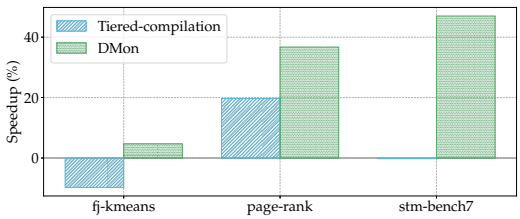


Figure 16: Speedup provided by selective profile-guided optimizations for just-in-time (JIT) compiled applications against tiered compilation (All $\sigma < 7.68\%$ of μ).

6.3 Real-World Case Studies

We evaluate the applicability of selective profiling and DMon to large systems by studying (1) speedups provided by DMon-guided optimizations on PostgreSQL [81]—one of the most popular database systems, and (2) speedups achieved after manual repair of data locality problems detected by selective profiling for just-in-time (JIT) compiled real-world applications from the Renaissance benchmark suite [83].

PostgreSQL case study. We evaluate DMon's ability to improve the locality (and thereby performance) of `PostgreSQL` v11.2 [81], one of the most popular open-source database management systems. For this study, we run the popular TPC-H [26] queries on a 1GB database stored in PostgreSQL. We intentionally select the database size to fit in memory to ensure a memory-bound workload (instead of disk-bound one), as the vast majority of real-world databases fit in memory [67, 80].

To evaluate DMon, we profile PostgreSQL with DMon while serving all 22 TPC-H queries. For these queries, selective profiling incurs 1.2% average and 2.7% maximum overhead. For `PostgreSQL`, DMon identifies a locality problem in a function (`ExecParallelHashNextTuple`) that accesses the members `area` and `parallel_state` of structure `hashtable` [39]. DMon identifies that this memory access is the primary reason for poor data locality in 6 out of 22 TPC-H queries. Moreover, this memory access causes L2 and L3 cache misses for all 22 TPC-H queries. The cause of the locality problem in this case is pointer chasing. Structure merging automatically repairs this problem and speeds up all 22 TPC-H queries, as shown in Fig. 15. The L3 cache misses in `PostgreSQL` are reduced by up to 22.11% (3.05% on average) and the latency of the 22 TPC-H queries are improved by up to 17.48% (6.64% on average). We also test optimized `PostgreSQL` based on DMon-profile on larger databases (10 and 100GB), where DMon improves the latency of the 22 TPC-H queries by 4.68% on average. For larger databases (10 and 100GB), the overall performance gain due to DMon's optimizations are comparatively less than (2% on average) that of smaller databases (1GB). That is because the performance of `PostgreSQL` for larger databases are primarily bottlenecked by storage I/O costs.

These results are particularly encouraging, considering that `PostgreSQL` is one of the most heavily-optimized codebases, having been improved by developers over the past 20 years. Most database developers hand-tune their code using the TPC benchmarks as regression tests (*i.e.*, their performance is best on TPC). This fact makes it even more promising that DMon-guided optimizations are able to improve the performance of these benchmark queries on a mature database system. We reported this data locality issue to the developers of `PostgreSQL` (for the version 11.2), which they have fixed since then.

Renaissance case study. A key advantage of just-in-time (JIT) compilation over ahead-of-time compilation (*e.g.*, Java vs. C++) is that JIT can apply dynamic optimizations—including limited data locality optimizations—using tiered compilation [65]. We compare selective profile-guided data locality optimizations to tiered compilation from OpenJDK [100] on real-world applications from the Renaissance suite [83]. For these applications, selective profiling incurs 2.2% average and 2.6% maximum overhead.

We use selective profiling to detect data locality issues in three Renaissance applications (jdk-concurrent `fj-kmeans`, apache-spark `page-rank`, and Scala `stm-bench7`). We omit other Renaissance benchmarks for which selective profiling does not find any data locality problems. Most of the data locality issues found here corresponds to Java/Scala source code (we map binary instruction information back to Java code using perf-map-agent [45]) of Renaissance applications. Since currently DMon's optimizations only support C/C++ applications, we manually apply data locality optimizations to these applications. In all cases, we modify <10 LOC.

As shown in Fig. 16, selective profile-guided optimizations provide on average 26% and up to 47% more speedup than tiered compilation. This demonstrates that selective profiling is effective even for JIT-compiled applications.

Apart from these real-world case studies, we have also tested DMon on `Memcached` [35] and `RocksDB` [33] with YCSB benchmarks [25]. For these two applications, the individual pieces that make up the locality issues are relatively minor. Compiler-based data locality optimizations typically add extra instructions and logic in the code, which only helps when there are many cache misses causing slowdowns. For program statements responsible for a relatively small percentage of all cache misses (less than 5%), applying these optimizations do not provide any speedup, as the extra code and logic outweighs the benefits.

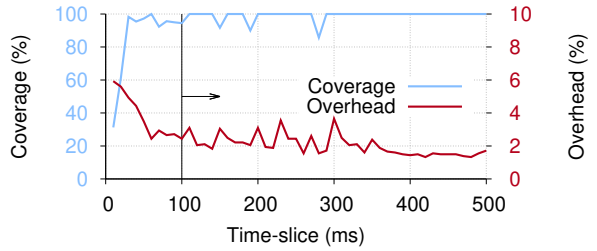


Figure 17: Effect of granularity of in-production time-slice on detection coverage and overhead (All $\sigma < 3.03\%$ of μ).

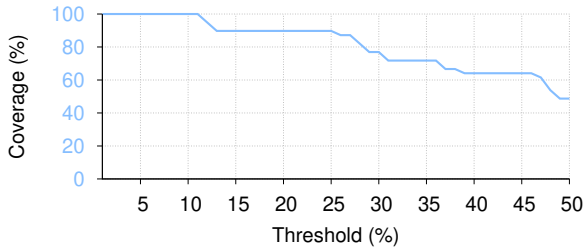


Figure 18: Effect of incremental monitoring threshold on the coverage of locality problems selective profiling detects across all benchmarks.

6.4 Sensitivity Analysis

We evaluate the impact of selective profiling's different parameters on effectiveness (coverage) and efficiency.

In-Production Monitoring Time-Slice. The granularity of the monitoring time-slice is a key design decision for selective profiling's incremental monitoring scheme (§3). Small time-slices allow selective profiling to identify locality problems for shorter-running applications, but also trigger frequent transitions during incremental monitoring and result in higher monitoring overhead. On the other hand, larger time-slices lower overhead but may fail to detect locality problems for shorter-running programs.

Fig. 17 shows the impact of the time-slice granularity on selective profiling's detection coverage (left y-axis) and overhead (right y-axis) for the benchmark, (`lu_ncb`). We vary the time-slice granularity from 10ms to 500ms (with 10ms increments) and measure selective profiling's coverage in detecting data locality issues and the associated performance overhead.

As shown in Fig. 17, selective profiling has lower coverage and higher overhead for smaller time-slices. As the time-slice granularity increases, selective profiling achieves greater coverage with lower overhead. Selective profiling's coverage is lower for smaller time-slices because selective profiling cannot monitor sufficient performance events in a small time slice. Beyond 100ms, both the coverage (99.07% on average with standard deviation of 3%) and the overhead (2.04% on average with standard deviation of 0.6%) lines flatten. Ergo, we set selective profiling's default time-slice as 100ms.

Incremental Monitoring Threshold. We vary the threshold of incremental monitoring (§3) from 1% to 50% and measure the coverage of data locality issues selective profiling detects for all 13 benchmarks in Table 2. 100% coverage is achieved when there is no incremental monitoring (*i.e.*, DMon continuously monitors events at the all levels of the locality tree). As shown in Fig. 18, selective profiling achieves greater than 80% coverage if the incremental monitoring scheme uses a threshold of <29%. Nevertheless, we set the default-threshold as 10%, as this threshold achieves 100% coverage.

In-Production Sampling Period. As described in §3, sampling period is a key design decision for selective profiling. Fig. 19 shows the impact of the sampling period on the coverage of locality issues selective profiling detects and its runtime overhead. We compute coverage with respect to the baseline coverage of 100%, achievable via the lowest possible sampling period offered by Linux `perf` (sampling every 100th event). A sampling period k on the x-axis means selective profiling will record one out of each k events. The left y-axis represents the runtime overhead and the right y-axis represents the coverage of locality issues selective profiling detects.

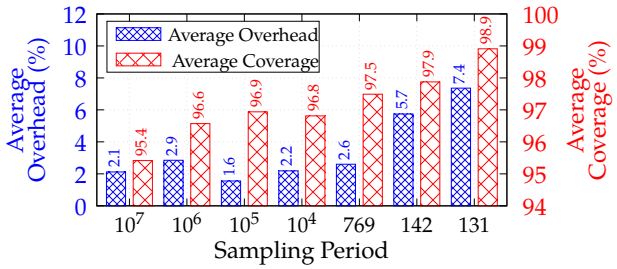


Figure 19: Effect of sampling period on the coverage of locality problems selective profiling detects and the average overhead across all benchmarks ($\sigma < 0.01\mu$).

The overhead and coverage reported in Fig. 19 are arithmetic averages over all benchmarks. A smaller sampling period increases the overhead of selective profiling, but also increases coverage. In our experiments, we chose a sampling period of 1000, which yields a high coverage of 97% with 2.6% overhead on average in layer 4 of selective profiling.

7 Related Work

DMon finds data locality problems with low overhead using selective profiling, identifies the root cause behind the problem, and guides optimizations to eliminate the problem. Existing profilers are not able to determine the root causes of data locality problems without incurring a high overhead.

Profilers. General-purpose profilers [57, 97, 102] report program hotspots without identifying the root cause behind performance problem. Consequently, recent studies propose specialized profilers to locate root cause for specific performance issues. Parallel profilers [36, 41, 44, 46] focus on critical path profiling to estimate potential performance gain [28, 107]. Synchronization profilers [4, 30, 108, 110] identify lock contention. Similarly, we design selective profiling as a special-

ized profiling technique for data locality. Selective profiling uses the APIs of a state-of-the-art profiler, Linux `perf`, and targets a subset of the events explored as part of the Top-Down [106]. Our main contributions over `perf` and Top-Down are: (1) full automation in profiling, (2) low-enough overhead for production deployment, (3) ability to automatically identify targeted optimizations based on the underlying performance problem.

Profile-guided data locality optimizations. Profile-guided approaches collect execution traces to identify where optimizations can be applied [21, 49, 51, 52, 59, 60, 69, 78]. State-of-the-art techniques [17, 37, 74–76] primarily address instruction locality. While prior work [50, 53, 86] also optimizes data locality, these solutions incur >10% profiling overhead. Selective profiling, however, incurs only 1.36% overhead on average (§6.1).

Static locality optimizations. Static approaches use complex analysis techniques to find opportunities to apply locality-improving transformations [14, 16, 18, 31, 47, 58, 66, 88, 105]. Alas, these techniques use compile-time heuristics to apply transformations, which can lead to sub-optimal speedups or even reductions in performance. To avoid these issues, we use application profiles collected by selective profiling to apply optimizations in a *targeted* manner, leading to better speedups and avoiding transformations which hurt performance.

Dynamic locality optimizations. There are several proposals for monitoring program execution and modifying program binaries to improve locality on the fly [32, 72, 89, 96]. These techniques require non-existent hardware support and incur high overhead (up to 6× [96]). Just-in-time (JIT) compilation techniques [21, 43] provide limited data locality optimizations. On the other hand, DMon works with existing hardware, incurs negligible overhead, and guides optimizations that provide better speedup (16.83% on average).

8 Conclusion

Poor data locality is a major performance problem that hurt applications in production. Unfortunately, existing data locality profilers are not efficient enough to be deployed in production. This is limiting, since production profiles are difficult to replicate offline. We address this problem by selective profiling, a technique capable of discovering data locality problems with negligible overhead (on average 1.36%) in production. We also design DMon, which guides automatic and manual data locality optimizations based on profiles generated using selective profiling. For an extensive set of real-world applications and widely-used benchmarks, DMon provides up to 53.14% and on average 16.83% speedup for the cases where DMon applies targeted optimizations after detecting significant data locality problems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Michael Stumm, for their insightful feedback and suggestions. This work was supported by the Intel Corporation, the NSF grants #1553169, #1629397, #2010810, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Yifan Zhao for running several `PostgreSQL` experiments. We also thank Xiaohe Cheng, Zhiqi Chen, and Shariq Hafeez for testing DMon on various applications. Finally, we thank Kevin Loughlin for his feedback on this paper's earlier versions.

A Artifact Appendix

Abstract

We provide the open-source public repository as an artifact for DMon.

Scope

This artifact allows to validate the effectiveness and efficiency of the selective profiling technique.

Contents

This artifact includes one end-to-end example of how to apply selective profiling to monitor in-production data locality issues and one example of data locality optimization applied in a targeted manner based on the output of selective profiling.

Hosting

We host the artifact on Github. Our open-source artifact repository can be obtained from `https://github.com/efeslab/DMon-AE`. The branch name for the artifact is `main`. The commit hash for the artifact is `d9a0f31`.

Requirements

Intel processor, Linux `perf`, `pmu-tools` [54] that implement the Top-Down methodology [106], and LLVM [56].

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] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 305–317, Piscataway, NJ, USA, 2017. IEEE Press.

[4] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313, 2017.

[5] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[6] 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.

[7] 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.

[8] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 462–473. ACM, 2019.

[9] Reza Azimi, Michael Stumm, and Robert W Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, 2005.

[10] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[12] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. IEEE, 2008.

[13] Michael D Bond and Kathryn S McKinley. Continuous path and edge profiling. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140. IEEE Computer Society, 2005.

[14] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM, 2008.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[16] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 252–262, New York, NY, USA, 1994. ACM.

[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. ACM, 2016.

[18] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 557–570, 2018.

[19] Jia Chen. Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen, 2018. [Online; accessed 16-Nov-2018].

[20] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.*, 42(9):1045–1057, September 1993.

[21] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization.

In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 332–340, New York, NY, USA, 2006. ACM.

[22] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[23] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[24] cloudfare. kyotocabinet/kcstashtest.cc at master - cloudflare/kyotocabinet. https://github.com/cloudflare/kyotocabinet/blob/master/kcstashtest.cc, 2013. [Online; accessed 4-April-2019].

[25] 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*, pages 143–154, 2010.

[26] Transaction Processing Performance Council. Tpc-h. [Online; accessed 23-April-2019].

[27] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 219–228, New York, NY, USA, 2013. Association for Computing Machinery.

[28] 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.

[29] Daniel Lemire. Is software prefetching (__builtin_prefetch) useful for performance?, 2018. [Online; accessed 24-April-2019].

[30] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.

[31] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Acm Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[32] Tyler Dwyer and Alexandra Fedorova. On instruction organization. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[33] Facebook. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb/, 2021.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[35] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[36] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices*, 46(6):458–469, 2011.

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

[38] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[39] The PostgreSQL Global Development Group. Line number 3225. https://github.com/postgres/postgres/blob/master/src/backend/executor/nodeHash.c.

[40] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[41] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, 2010.

[42] Ravi Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. *Intel Software Network*, 2008. [Online; accessed 5-December-2020].

[43] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented*

Programming, Systems, Languages, and Applications, OOPSLA '04, pages 69–80, New York, NY, USA, 2004. ACM.

[44] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGARCH Computer Architecture News*, 40(1):223–234, 2012.

[45] jvm-profiling-tools. perf-map-agent, 2018. [Online; accessed 6-December-2020].

[46] Melanie Kambadur, Kui Tang, and Martha A Kim. Harmony: Collection and analysis of parallel block vectors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463. IEEE, 2012.

[47] Mahmut Taylan Kandemir. A compiler technique for improving whole-program locality. In *ACM SIGPLAN Notices*, volume 36, pages 179–192. ACM, 2001.

[48] 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*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[49] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient tracing of cold code via bias-free sampling. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 243–254, 2014.

[50] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, pages 101–110. IEEE, 2014.

[51] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[52] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, ISCA 2021, June 2021.

[53] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Huron: hybrid false sharing detection and repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 453–468, 2019.

[54] Andi Kleen. Github - andikleen/pmu-tools: Intel pmu profiling tools. https://github.com/andikleen/pmu-tools.

[55] William Landi and Barbara G Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, 1991.

[56] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[57] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[58] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *ACM SIGPLAN Notices*, volume 52, pages 1–16. ACM, 2017.

[59] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 171–180. IEEE Computer Society, 2011.

[60] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[61] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: a lightweight profiler to guide array regrouping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 405–415. IEEE, 2014.

[62] Xu Liu and Bo Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 47:1–47:12, New York, NY, USA, 2015. ACM.

[63] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213. Citeseer, 2006.

[64] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 185–195, 2018.

[65] Markus Weninger. What exactly does -xx:-tieredcompilation do?, 2016. [Online; accessed 11-November-2019].

[66] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996.*, pages 94–104, 1996.

[67] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[68] Mircea Trofin. Support for cache prefetching profiles. by mtrofin · pull request #75 · google/autofdo, 2018. [Online; accessed 17-November-2019].

[69] Svetozar Miucin and Alexandra Fedorova. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems*, pages 243–253. ACM, 2018.

[70] Todd C Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, to the Department of Electrical Engineering.Stanford University, 1994.

[71] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, New York, NY, USA, 1992. ACM.

[72] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2018.

[73] Joseph Musmanno. Data intensive systems (dis) benchmark performance summary. Technical report, TITAN SYSTEMS CORP WALTHAM MA, 2003.

[74] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165. ACM, 2018.

[75] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14. IEEE Press, 2019.

[76] 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.

[77] Paratools. Threadspotter. http://threadspotter.paratools.com/, 2019. [Online; accessed 22-Oct-2019].

[78] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348. ACM, 2010.

[79] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM.

[80] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[81] PostgreSQL. Postgresql: The world's most advanced open source relational database. [Online; accessed 23-April-2019].

[82] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[83] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47. ACM, 2019.

[84] Manman Ren and Shane Nay. Improving iOS Startup Performance with Binary Layout Optimizations, 2019. [Online; accessed 25-Oct-2019].

[85] Roman Oderov. Sampling and vtune's disadvantages, 2012. [Online; accessed 23-April-2019].

[86] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[87] J Sedlacek and H Thomas. Visualvm all-in-one java troubleshooting tool, 2018.

[88] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, 1999.

[89] Jithendra Srinivas, Wei Ding, and Mahmut Kandemir. Reactive tiling. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 91–102. IEEE, 2015.

[90] 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.

[91] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[92] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM.

[93] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[94] Vish Viswanathan. Disclosure of hardware prefetcher control on some intel processors. *Intel SW Developer Zone*, 2014.

[95] David W Wall. Predicting program behavior using real or estimated profiles. *ACM SIGPLAN Notices*, 26(6):59–70, 1991.

[96] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. On-the-fly structure splitting for heap objects. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2012.

[97] Wikipedia contributors. Perf (linux) — Wikipedia, the free encyclopedia, 2018. [Online; accessed 24-April-2019].

[98] Wikipedia contributors. Vtune — Wikipedia, the free encyclopedia, 2018. [Online; accessed 23-April-2019].

[99] Wikipedia contributors. Clang — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[100] Wikipedia contributors. Openjdk — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=OpenJDK&oldid=927329117, 2019. [Online; accessed 23-November-2019].

[101] Wikipedia contributors. Strip (unix) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[102] Wikipedia contributors. Dtrace — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DTrace&oldid=950798652, 2020. [Online; accessed 25-April-2020].

[103] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[104] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 219–229, New York, NY, USA, 1994. ACM.

[105] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 343–356, 2013.

[106] 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.

[107] 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. ACM, 2019.

[108] 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.

[109] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[110] 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.

CLP: Efficient and Scalable Search on Compressed Text Logs

Kirk Rodrigues, Yu Luo, Ding Yuan
University of Toronto & YScope Inc.

Abstract

This paper presents the design and implementation of CLP, a tool capable of losslessly compressing unstructured text logs while enabling fast searches directly on the compressed data. Log search and log archiving, despite being critical problems, are mutually exclusive. Widely used log-search tools like Elasticsearch and Splunk Enterprise index the logs to provide fast search performance, yet the size of the index is within the same order of magnitude as the raw log size. Commonly used log archival and compression tools like Gzip provide high compression ratio, yet searching archived logs is a slow and painful process as it first requires decompressing the logs. In contrast, CLP achieves significantly higher compression ratio than all commonly used compressors, yet delivers fast search performance that is comparable or even better than Elasticsearch and Splunk Enterprise. In addition, CLP outperforms Elasticsearch and Splunk Enterprise's log ingestion performance by over 13x, and we show CLP scales to petabytes of logs. CLP's gains come from using a tuned, domain-specific compression and search algorithm that exploits the significant amount of repetition in text logs. Hence, CLP enables efficient search and analytics on archived logs, something that was impossible without it.

1 Introduction

Today, technology companies easily generate petabytes of log data per day. For example, eBay reported generating 1.2 PB of logs per day in 2018 [46]. This data can be used for a variety of important use cases including security forensics, business insights, trend analysis, resource optimization, and so on. Since many of these cases benefit from large amounts of data, companies strive to retain their logs for as long as possible. Moreover, some industries (e.g., health services) are required by law to store their logs for up to six years [5].

However, storing and analyzing a large amount of log data impose significant costs. Although it is difficult to obtain transparent, publicly available information about companies' storage costs, studies have estimated that a lower bound for capital depreciation and operational costs could be on the order of two cents per gigabyte, per month [7]. For a company like eBay, this translates to over $50 million to store the logs generated in a year, and nearly $500 million to store the logs generated over three years. As a result, the log management industry has grown incredibly large.

Currently, Elastic [2] and Splunk [4] are two of the largest companies in the industry. In just their last fiscal year, Elastic reported revenue of $428 million with a total of 11,300 customers [14] while Splunk reported revenue of $2.359 billion with 19,400 customers [36]. Moreover, their offerings, Elasticsearch [15] and Splunk Enterprise [37], are used by several large companies like eBay, Verizon, and Netflix.

Tools like Splunk Enterprise and Elasticsearch operate by generating external indexes on the log messages during ingestion. Then in response to a query, these tools can quickly search the indexes corresponding to the logs, decompressing only the chunks of data that may contain logs matching the search phrase. Elasticsearch, for example, is built around a general-purpose search engine Lucene [42]. However, this approach comes at the cost of a large amount of storage space and memory usage. Although these tools apply light compression to the logs, the indexes often consume an amount of space that is the same order of magnitude as the raw logs' size; furthermore, these indexes must be kept mostly in memory or on fast random access storage in order to be fully effective. Thus, Splunk Enterprise and Elasticsearch users with large amounts of data can only afford to retain their indexed logs for a short period, typically a few weeks [8].

To avoid discarding logs at the end of their retention period, companies can use industry-standard compression tools like Gzip [21] to archive them, potentially gaining a 95% reduction in storage costs. In addition, recent advancements in compression algorithms like Zstandard [16] bring significantly improved compression and decompression speeds. However, these general-purpose compressors are not designed with search (on compressed data) in mind. They typically encode duplicates in length-distance pairs [40, 49], i.e., starting from the current position, if the next L (length) characters are the same as the ones starting at D (distance) behind, we can encode the next L characters with (D,L), and directly embed this pair at the current position, an approach known as an internal macro scheme [40]. Performing searches on this archived data, however, is painful and slow—the tool needs to sequentially scan the entire data set, essentially decompressing the data. This leads to the unfortunate reality that *log analysis and log archiving are generally mutually exclusive*.

To bridge this gap, we have created a method for lossless log compression that still allows efficient searches on the compressed data, without the need to decompress every log message. It works by combining a domain-specific compression and search algorithm with a lightweight general-purpose compression algorithm. The former allows us to achieve a modest amount of compression without sacrificing search performance. The latter increases the overall compression ratio significantly with a minor impact on compression and decompression performance.

The domain-specific compression algorithm uses an external macro scheme, i.e., it extracts the duplicated patterns into a dictionary that is stored separately from the encoded log messages [40]. A search query will be processed by first searching in the dictionary, and then searching those encoded messages for which the dictionary search suggests possible matches. This method relies on the simple observation that today's software logs contain a large amount of repetitive static text. By systematically separating the static text from the variable values, the algorithm can deduplicate the static text into a dictionary. Applying a similar process to the variable values, the algorithm converts an entire log message into a series of integers and dictionary entries that are easily compressible using a general-purpose compressor.

The search process similarly encodes the query string as a compressed message and searches for a match; but supporting queries with wildcards makes this process significantly more involved. For example, a wildcard can make it ambiguous whether a token is part of the message's static text or whether it is part of a variable. As a result, the algorithm must consider the effect of wildcards at every stage of the encoding and search process.

Using this method of compression and search, we have built an end-to-end log management tool, CLP[1], that enables real-time data ingestion, search, analytics, and alerting on top of an entire history of logs. CLP is agnostic to the format of logs and can ingest heterogeneous and unstructured logs. As a result, CLP is capable of reducing the size of currently archived logs while simultaneously enabling search and analytics on the compressed data.

Our evaluation shows that CLP's compression ratio is significantly higher compared to all tested compressors (e.g., 2x of Gzip), while enabling efficient search on compressed data. This comparison even includes industry-standard tools like Zstandard at their highest (and slowest) compression level. Furthermore, CLP's search speed outperforms commonly used sequential search tools on compressed data by 8x in a wide range of queries. Even compared with index-based log-search tools Splunk Enterprise and Elasticsearch, CLP outperforms them by 4.2x and 1.3x respectively. CLP's distributed architecture further allows it to scale to petabytes of logs. CLP is open-sourced and can be found at `https://yscope.com`. It is also hosted in the cloud so users can use it as a service.

CLP's main limitation is that its algorithm is designed primarily for text logs. This is not a problem in the vast majority of software logs that we have seen, but we acknowledge that there are projects that log primarily binary or structured data. However, if converted to text with a verbose schema, these logs can be compressed and searched using CLP without additional overhead.

The rest of this paper is organized as follows. §2 describes the core elements of CLP's design for compression and search. §3 details how CLP handles the various intricacies of handling wildcards and patterns of variables. §4 describes our syntax for variable patterns. §5 explains how CLP can cache queries in reusable manner for performance. §6 describes a characteristic of CLP's compression format that can be used for privacy control. §7 discusses the evaluation results of CLP compared with other tools. Finally, §8 discusses related work, before we conclude in §9.

2 Design Overview

CLP is a complete end-to-end system for ingesting, archiving, searching, and analyzing log messages. Figure 1 shows an overview of CLP's compression and search architecture. Within the compression architecture, logs can be ingested either through CLP's real-time ingestion engine (e.g., from rsyslog, Fluentd, Logstash, etc.) or by reading them directly from local or cloud storage (e.g., Amazon S3 [29]). The compression nodes compress the ingested logs into a set of archives. Users can access the compressed logs transparently using a Unix terminal through the Filesystem in Userspace (FUSE) layer or by querying them through CLP's search interface.

CLP allows users to query their logs using a wildcard search followed by a series of operators. An example query is shown in Figure 2, containing four commands pipelined with a Unix-style pipe ('`|`'). The first command returns all log messages matching the search phrase ('`*`' is a wildcard character that matches zero or more characters). Results are piped to the `regex` operator which uses a regular expression to extract the container ID and operation runtime, storing them in user defined variables. Next, the `filter` operator filters for runtimes that are above "`0.1`". Finally, the `unique` operator generates a list of unique container IDs that satisfy the filter. Overall, this query returns the unique containers where the assignment operation took over `0.1` seconds in the `172.128.*.*` subnet. We refer to this type of query as a *pipelined query*.

CLP's search architecture supports pipelined queries by combining search nodes with a MapReduce-style [11] framework. CLP receives queries through its web UI or Python APIs. Queries are first serviced by the search nodes which perform a wildcard search on the archives. Results are then forwarded to the operator nodes, after which the final results are sent back to the user. Users can also create alerts that trigger when newly added log messages satisfy a saved query.

[1] CLP stands for Compressed Log Processor

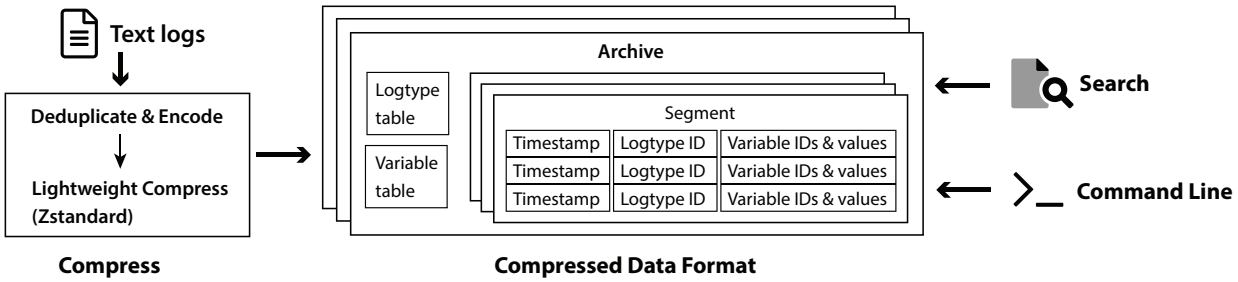


Figure 1: Overall architecture of CLP.

```
"Task * assigned to container*:172.128" |
regex "(?<container>container_\d+).* took (?<runtime>\d+)"
| filter float(runtime) > 0.1 | unique container
```

Figure 2: A query example. CLP operator and keywords are in blue, and user-defined variables are in red.

Note that because search is the first stage of every query, it is also the most important for performance since it operates on compressed data and all other stages operate on the decompressed data it outputs.

We aim to satisfy three objectives with this design: First, logs should be compressed losslessly so that users can delete their original logs without worrying that CLP would destructively transform them (e.g., by changing the precision of floating-point values). Second, users should be able to search their logs for any value, in contrast to index-based search tools which typically only allow searches for indexed values. For example, unlike grep-like tools that respect all characters in a search phrase, indexed-based search tools typically ignore punctuation and stop words (e.g., "and"). Finally, CLP should be performant and scalable so that users can use it to ingest and search a large amount of log data while saving on storage costs. By satisfying these objectives, we aim to bridge the gap between conventional log archival and search, e.g., using gzip and grep, and large-scale log analysis, e.g., using Splunk Enterprise or Elasticsearch.

The core of CLP is implemented in C++ for performance while higher-level functionality is built in a variety of languages from Java to JavaScript.

2.1 Compression

CLP's compression consists of two steps: first it deduplicates highly repetitive parts of each log message and encodes them in a special format, then it applies a lightweight compressor to the encoded data, further reducing its size. This section focuses on explaining the first step.

CLP splits each message into three pieces: 1) the log type, which is typically the static text that is highly repetitive, 2) variable values, and 3) the timestamp (if the message contains one). CLP further separates variable values into two categories: those that are repetitive, such as various identifiers (e.g., a username), and those that are not (e.g., a job's completion time). We refer to the former as *dictionary variables* since CLP extracts and stores them in a dictionary; the latter are called *non-dictionary* variables. Figure 3 shows a log message and how CLP converts it into a compressed form. Overall, this requires parsing the message to extract the aforementioned components and then compressing it using CLP's domain-specific compression algorithm.

2.1.1 Parsing Messages

CLP parses logs using a set of user-specified rules for matching variables. For example, Figure 4 lists a set of rules that can be used to parse the example log message. Lines 3–5 contain three dictionary variable schemas and line 8 contains a non-dictionary variable schema. This is similar to tools like Elasticsearch and Splunk Enterprise that either provide application-specific parsing rules or ask users to write their own. CLP provides a default set of schemas that can be applied universally to all log formats, or users can optimize them to achieve better compression and faster searches on their workloads.

One challenge with using variable schemas is that they can match pieces of a log message in multiple ways. For instance, "`172.128.0.41`" could match the schema for an IP address or it could match two instances of the floating point number schema, joined by a period. However, we have observed that developers typically separate different variable values with one or more *delimiter* characters. Furthermore, they also use delimiters to separate variable values from static tokens in the log type. We call this the *tokenization rule*, which states that *a token is inseparable*. That is, an entire token is either a variable value or part of the log type. In this case, "`172.128.0.41`" will be treated as a single token, so it can only match an IP address instead of two floating point numbers joined by a period. Accordingly, CLP allows users to specify a set of delimiters that ensures their schemas only match variables in a way that respects the tokenization rule.

To parse a log message, CLP first parses and encodes the message's timestamp as milliseconds from the Unix epoch. CLP then tokenizes the log message using the user-specified

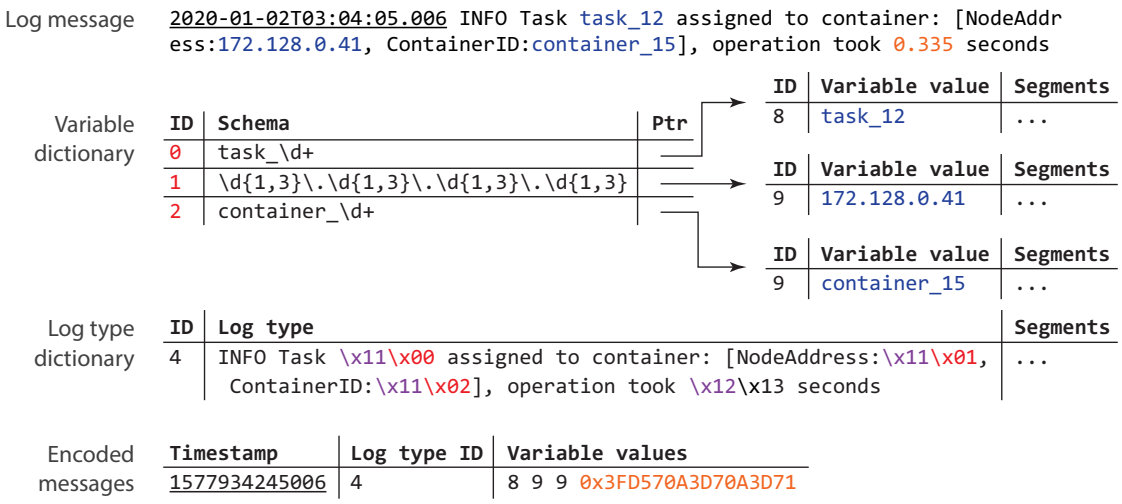


Figure 3: A log message and its encoding. Dictionary variables are in blue; Non-dictionary variables are in orange.

```
delimiters: "[],: "
dictionary_variables:
  "task_\d+"                          # Task ID
  "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" # IP
  "container_\d+"                  # Container ID

non_dictionary_variables:
  "\d+\.\d+" as floating_point_number
```

Figure 4: Schemas used to parse the example in Figure 3.

delimiters. For each token, CLP compares it with each variable schema to determine whether it is a variable value. In Figure 3, CLP identifies three dictionary variables in the log message—"task_12", "172.128.0.41", and "container_15"—and a non-dictionary variable value, "0.335".

2.1.2 Compressing Messages

Once parsed, the dictionary variables are stored in a two-level variable dictionary, referred to as a *vDict*. The first level maps each dictionary variable schema to a unique ID. Each schema is also mapped to a pointer that points to the second level of the vDict, where the actual variable value is stored. In Figure 3, the schemas for the task ID, IP address, and container ID are mapped to IDs 0, 1, and 2 in the first level, and the actual variable values are stored in the second level.

Non-dictionary variable values are stored directly in the encoded log message if possible. For example, "0.335" is encoded using the IEEE-754 standard [1] and stored as a 64-bit value in the encoded message. CLP currently supports encoding floating point numbers and integers as non-dictionary variables. If a non-dictionary variable cannot be encoded precisely within 64-bits (e.g., its value overflows), it is stored as a dictionary variable instead. Non-dictionary variables tend to be unique values like counters, so they do not benefit from being stored in a dictionary. We use a fixed-width 64-bit encoding instead of a variable-width encoding because it is simple to implement, and the space inefficiency is diminished by the lightweight compressor applied to the encoded data.

The remaining portion of the log message is treated as being part of the log type, where variable values are replaced with special placeholder characters. Each unique log type is stored in the log type dictionary, or *ltDict*, and is indexed by an ID. CLP uses byte '\x11' to represent a dictionary variable value. The next one or more bytes after '\x11' are an index into the vDict's first level, i.e., an index to the variable schema. In Figure 3, '\x00', '\x01', and '\x02' in the log type are indices to the three schemas for the task ID, IP address, and container ID in the vDict. CLP uses '\x12' as the placeholder for a floating point non-dictionary value. The next byte, '\x13', in the log type indicates that there is one digit before and three digits after the '.' character in the raw log message, ensuring the floating point value can be losslessly decompressed.

Note that we could choose any bytes for the placeholder characters, but since '\x11' and '\x12' are not printable ASCII characters, they are unlikely to appear in text logs. If they do, CLP will escape them before insertion into the log type.

CLP outputs the encoded message as a tuple with three elements as shown in Figure 3: a 64-bit timestamp, a 32-bit log type ID, and a sequence of 64-bit variable IDs and encoded variable values.

We have experimented with additional encoding schemes that can further reduce the size of the encoded data, but decided not to adopt them due to their undesirable trade-off. For example, we could store variable IDs and non-dictionary variable values using a variable-length encoding, instead of a fixed-length 64-bit encoding. We have also experimented with delta encodings, run-length encodings, and so on. However, these would come at the cost of search performance since

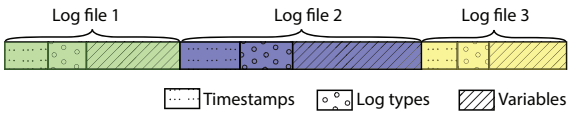


Figure 5: Storing encoded messages in column-oriented manner. It shows a segment that contains three encoded log files.

it is faster to scan fixed-length values than variable-length values. Moreover, the space savings are negligible after the lightweight compressor is applied on the encoded data.

2.1.3 Decompressing Messages

CLP's decompression process is generally a reversal of the compression process. Given an encoded message, CLP uses the message's log type ID to find the corresponding log type in the ltDict. CLP then reconstructs the variable values and replaces the placeholders in the log type. For example, CLP reconstructs the variable value "`task_12`" in Figure 3 as follows: the first '`\x11`' in the log type indicates that it is a dictionary variable, so CLP uses the next byte, '`\x00`' as an index into the first level of the vDict. CLP then uses the variable ID stored in the encoded message (8 in this case) to index the corresponding second level of the vDict, and restores the variable value "`task_12`". Finally, CLP converts the timestamp back to text and inserts it into the message.

2.1.4 On-disk Format

Figure 1 also shows the on-disk format of CLP's compressed logs. CLP encodes each message and stores them in the same temporal order as in the original log file. This ensures the file can be losslessly decompressed. The encoded messages are initially buffered in memory, and once the buffer reaches a certain size, they are compressed using Zstandard before being written to disk, creating what we call a *segment*.

Encoded messages are stored in a column-oriented manner [39], as shown in Figure 5—CLP stores the timestamp column of the messages from log file 1, then its log type IDs, and finally the variable IDs and values column, before storing the three columns of the next log file. Storing columnar data-series reduces data entropy within Zstandard's compression window, significantly improving compression ratio. In addition, columnar data-series can improve search performance: for instance, if users search for a message in a specific time range, CLP can skip messages outside that time range by only scanning the timestamp column rather than all columns.

Multiple segments further belong to an *archive*, where all segments in an archive use the same log type and variable dictionaries. CLP automatically closes and creates a new archive when the dictionaries reach a size threshold. This ensures that the dictionaries do not grow too large such that they have non-negligible loading times for decompression and search. CLP also compresses the dictionaries using the same lightweight compressor applied to the segments.

Each entry in the ltDict and the vDict's second level also has a list of pointers to segments that contain the particular log type or variable value. CLP is I/O bound reading segments, so this serves the purpose of a coarse-grained search index. We index at the granularity of segments since any query that has a hit in a segment requires the segment to be decompressed from its beginning to the matched message. Without the index, any search that matched a dictionary entry required searching all segments in the archive.

For each archive, CLP also stores metadata about the log files and directories that were compressed. For each file, the metadata contains the original filesystem path of the file, the number of log messages it contains, the starting and ending timestamp of the messages in the file, the format of its timestamp (used to reconstruct the timestamp during decompression), and the segment that contains the compressed messages from the log file. In addition, the metadata contains the three offsets in the segment corresponding to the starting locations of the messages in this log file: one for each of the timestamp column, log type column, and variable column. These offsets are used to speedup the search when users use search filters. For example, a user could search for filenames that match a specific pattern, such as `yarn.log` (the log produced by YARN in a Hadoop cluster). Users can also specify the time range of the search, so CLP will first filter log files based on the starting and ending timestamps. In such cases, the metadata as well as the content in the data columns themselves allow CLP to skip scanning parts of data columns or files.

For directories, the metadata in the archive stores the paths of any empty directory that was compressed. An empty directory may be indicative of missing logs or it may be named after an identifier that the user wishes to keep. Thus, to ensure lossless decompression, these paths must be stored.

CLP also supports different compression modes that can offer improved compression at the cost of a minor reduction in performance. This is achieved by changing the lightweight compressor's settings. CLP currently ships with three modes: "Default" that uses Zstandard at level 3 and is concurrently optimized for compression speed and search performance; "Archive" that uses 7z-lzma at level 1 and offers higher compression with slightly reduced search performance; and finally, "Ultra" that uses 7z-lzma at level 9 and offers even higher compression with further reduced search performance. CLP can migrate between these modes by simply decompressing and recompressing the segment.

2.2 Search

Given a search phrase, CLP processes it in the same way that it compresses a log string: CLP tokenizes the phrase, extracts variable values, and constructs the log type. Next, CLP searches the archive's dictionaries for the log type and

#	Log type	Variables	CLP's processing
1	"Task * assigned to container*:\x11\x01"	"172.128*" (IP address)	Log type, var. search, scan segments
2	"Task * assigned to container*:\x12?"	"172.128*" (float num.)	Log type search (no match)
3	"Task * assigned to container*:178.128*"	-	Log type search (no match)
4	"Task * assigned to \x11\x02*:\x11\x01" "172.128*" (IP address)	"container*" (container ID)	Log type search (no match)
5	"Task * assigned to \x11\x02*:\x12?" "172.128*" (floating point num.)	"container*" (container ID)	Log type search (no match)
6	"Task * assigned to \x11\x02*:178.128*"	"container*" (container ID)	Log type search (no match)

Table 1: Processing of the search example in Figure 2. Each row is a sub-query generated by CLP.

dictionary variables. If matches are found for the log type and all dictionary variables, CLP proceeds to search the segments for encoded messages that contain the matching log type ID, dictionary variable IDs, and encoded non-dictionary variables.

However, wildcards in the search phrase complicate this process. CLP supports search phrases that can contain two types of wildcard characters: '*' (henceforth referred to as a *-card) that can match zero or more characters and '?' (henceforth referred to as as a ?-card) that matches any single character. First, it is nontrivial to tokenize a string with wildcards. For example, the string "Task*assigned" could be a single token or two tokens ("Task*" and "*assigned") since a *-card can match both non-delimiter and delimiter characters. Furthermore, it is nontrivial to determine if a token with a wildcard matches a variable schema. Finally, without wildcards, a token will be unambiguously categorized as either a log type, a dictionary variable, or a non-dictionary variable; but with wildcards, a token could belong to multiple categories. We address the first two issues in Section 3 and continue a discussion of the third challenge below.

2.2.1 Handling Ambiguous Tokens

Consider the search command in Figure 2. CLP first inserts a *-card at the beginning and end of the search string, turning it into a substring search to match user-expectations. Then after tokenization, CLP recognizes the following tokens: "*Task", "assigned", "to", "container*", "172.128*". Note that CLP does not consider a lone *-card as a token. For example, the *-card after "Task" is not treated as a token.

Each token is then compared against all of the variable schemas. CLP determines that "*Task", "assigned", and "to" do not match any schemas, hence they are part of the log type. Ambiguity exists for the other two tokens: "172.128*" could match an IP address schema or a floating point number, "container*" could match a container ID, and both could also be part of the log type. This creates a total of six combinations, and CLP generates a sub-query for each possibility.

Table 1 lists the six generated sub-queries. The first three treat "container*" as part of the log type, and "172.128*" is treated as part of an IP address, a floating point number, and the log type in sub-query 1, 2, and 3 respectively. When treating "172.128*" as a floating point number, CLP does not know the value's exact precision, so it inserts a ?-card to match all possibilities. Sub-query 4–6 in Table 1 consider the cases that "container*" is treated as a dictionary variable container ID.

Each sub-query will be processed in three steps. First, CLP searches the ltDict for matching log type. Only when there is a matching log type, it proceeds to the next step of searching the vDict for the dictionary variables. And only when there is at least one matching result for every dictionary variable, CLP proceeds to the third step. It takes the intersection of the segment indexes of the matching log type and dictionary variables, and for each of these segments, CLP decompresses the segment and searches for encoded messages matching the encoded query. If any of the first two steps return no matching result, or the intersection of the segment indexes is empty, the sub-query processing returns with no match. Different sub-queries will be processed in parallel.

For the six sub-queries shown in Table 1, only the first sub-query will exercise all three steps and return the matching log message shown in Figure 3. The processing of the other five sub-queries will return after step one, because the generated log type does not match any log types in the ltDict (these log types are impossible).

2.2.2 Optimizing CLP Queries

The way users write their search phrase can significantly affect the speed of the search. In our evaluation, dictionary search time is negligible compared to a segment scan; furthermore, log type dictionary search time is negligible compared with variable dictionary search. Therefore the best practice is to provide enough information in the search phrase to help CLP narrow down the log type or the dictionary variable values or both. The user can also speedup the search by filtering for a specific time range or log file path, dramatically shrinking the search scope. For example, the user could use a search phrase "172.128" to perform the previous query, but the search performance may be much worse. CLP will determine that "172.128" could be part of an IP address, floating point number, or the log type, and generate three sub-queries with log types being "\x11\x01", "\x12?", and "172.128". However, the first two sub-queries will likely result in numerous matching log

types in practice, i.e., any log type that contains an IP address or a floating point number, so CLP will end-up scanning a large number of segments.

Currently CLP does not use any additional index on its dictionary entries. A search on the ltDict, for example, will sequentially scan each entry. This is not a problem for now as the bottleneck in search is in scanning the segments, because the dictionaries are small. CLP also does not have any index on non-dictionary variables. We plan to add index (e.g., B-trees) to non-dictionary variables in the near future.

2.3 Handling Special Cases

Users have two options for changing variable schemas after log data has already been compressed. The new schema can be applied only to newly compressed data, in which case each archive will also need to retain the schemas that were used to compress the data. Alternatively, the users can ask CLP to update existing archives to use the new schema, and CLP will have to decompress and recompress the data.

CLP can also warn the user if the schemas they provided are not optimal. For example, if the user forgot to specify the schema of a variable, that variable would be encoded as part of the log type, and could "pollute" the log type dictionary where a large number of similar log types are created, with the only difference being that variable value. CLP can detect this case by comparing the edit distance between log types and issue a warning.

Although rarely used, CLP also supports the deletion of log messages. The encoded messages will be deleted from the affected segments, which involves recompressing the segment data using the general-purpose compressor and writing it to disk. The segment index in the dictionaries will also need to be updated.

Currently CLP does not support SQL-style join operations in a single query. However, users can perform joins in their client program using CLP's APIs.

2.4 Distributed Architecture

CLP adopts a simple controller and data node design that enables high scalability, similar to other widely used big data systems [9, 11, 22, 41]. The central controller simply manages metadata and node failures, while the data intensive computation of compression and search as described above are performed by each data node independently. Compressed data is stored on a distributed filesystem to ensure reliability.

The controller maintains three metadata tables: 1) log files, 2) archives, and 3) empty directories. The log files table stores the metadata of each raw log file (its file system path, the number of log messages, etc.) as well as the archive that contains this log file. Note that if the log messages are directly streamed to CLP using a log aggregation tool (e.g., rsyslog, Fluentd, Logstash, etc.), CLP still splits them into logical files once the buffered log messages reach a certain size or time frame. The archives table stores the metadata of each archive including which data node stores this archive. The empty directories table stores the paths of empty directories compressed in each archive.

The purpose of these metadata tables is only to speedup the search. For example, a user can specify a filter to only search log files whose file names match a certain pattern. The information stored in these tables is also stored in the archives, so even if the tables are lost, there is no risk of data loss. Nevertheless, we replicate the metadata tables three times with failover handling.

In order for CLP's compression and search to scale in a distributed system, each archive is independent of other archives and immutable once written. This independence makes compression easily parallelizable without any synchronization between threads writing different archives. The immutability ensures that a search thread can query an archive without synchronizing with a compression thread. To avoid coordination between search threads, each archive is only queried by a single thread for a given query. Thus, CLP parallelizes compression and search at the granularity of individual archives.

File System Integration Using FUSE. CLP has the ability to transparently integrate with a user's existing environment. For example, a user can use GNU `find` to search for files, and use `VIM` to open a compressed log file. We implement this by intercepting the file system operations using FUSE (Filesystem in Userspace) [44]. It walks the directory hierarchy stored in the log files table and decompresses the required data on demand to satisfy I/O requests. Common I/O optimizations such as caching, I/O request re-ordering and batching are performed to further increase CLP's efficiency and performance.

3 Wildcards and Schemas

We face two fundamental challenges in handling wildcards. Recall that CLP's encoding process requires tokenizing the input, extracting each token that matches a variable schema, and finally composing the log type. The first challenge in handling wildcards is determining how to tokenize a string containing wildcards, given that a wildcard could either match a delimiter or non-delimiter character. The second is determining if a token containing wildcards (*a wildcard token*) *could* match a given variable schema. Both of these challenges occur because a wildcard string has a range of possible inputs that it could match, and CLP's task is to encode all possible inputs so that they can be used for search.

3.1 Wildcard String Tokenization

To tokenize a wildcard search string, we need to consider each possible interpretation of every wildcard in the string.

#	*-card interpretation	Spans
1	Delimiters only	"*to", "*", "container*"
2	Non-delimiters only	"*to*container*"
3	Both	"*to*", "*", "*container*"

Table 2: The spans generated by tokenizing "*to*container*" depending on the interpretation of the central *-card.

For example, consider the search string "*to?container*". If the ?-card is interpreted as a delimiter, the string will generate three *spans*: "*to", "?", and "container*". We use the term span to refer to either a contiguous set of non-delimiter characters, or a contiguous set of delimiter characters. Using the schemas in Figure 4 on these spans, CLP will find that the last span matches a variable schema and the rest match the log type in Figure 3. However, if the ?-card is interpreted as a non-delimiter, then the entire string will be treated as a single token and CLP will find neither a matching schema nor log type. Accordingly, CLP must generate sub-queries from each unique tokenization of the search string.

To handle *-cards, CLP technically needs to consider that a *-card can be interpreted as either 1) matching non-delimiters only, 2) matching delimiters only, or 3) matching non-delimiters and delimiters. However, because a *-card matches *zero* or more characters, we can skip a case.

Consider the search string "*to*container*". To simplify the discussion, we only consider the interpretation of the central *-card and assume the others are interpreted as non-delimiters only. Table 2 lists the spans generated for each case. Note that the third tokenization is from interpreting the *-card as zero or more non-delimiters, followed by zero or more non-delimiters and delimiters, followed by zero or more non-delimiters. Comparing the first and third tokenization, we can see that the third is a more general version of the first. As a result, CLP does not need to consider the first tokenization. We can generalize this as follows: If a *-card is interpreted to have a different type than either of the characters surrounding it, the tokenization should split the string at the *-card while leaving *-cards attached to the surrounding character.

3.2 Comparing Expressions

To compare a wildcard token to a variable schema, CLP needs to determine if they overlap in the words that they could match. More formally, let U represent the words matched by the wildcard-containing token, and V represent the words matched by the variable schema. CLP needs to determine if $U \bigcap V \neq \emptyset$. For example, the wildcard token, "task_?", and the variable schema, "task_\d+" both match task IDs with one digit. Therefore, CLP can consider that this token matches the schema. However, this intersection does not imply that $U = V$, so CLP must still consider that "task_?" may be part of the log type (e.g., if the ?-card matches an alphabet). To determine if $U = V$, CLP could verify that $U \bigcap V^c = \emptyset$, where V^c is the set complement of V, but we find that this is rarely true in practice.

This is a standard problem of comparing the accepted input sets of two regular expressions. However, modern regular expression engines support *irregular* expressions (e.g., back-references) that prevent them from supporting this standard operation. Furthermore, we could not find a widely-used engine that supported strictly regular languages. So we built our own engine and use it to compute this intersection as well as enforce rules on the supported variable schemas.

4 Schema Design

Up until this point, we have only discussed schemas that match a single token. However, there are several variable values that fall outside this definition. For example, using the delimiters in Figure 4, the variable value "0.0.0.0:80" is a set of two tokens ("0.0.0.0" and "80") joined by a delimiter (':'). Similarly, " block id : 1073741827 " is a variable value that can only be categorized as a block ID if the schema takes into account the tokens before the actual variable value. To handle these cases, we extend our definition of a schema to include multiple regular expressions.

A schema in CLP is a sequence of regular expressions, where each expression exclusively contains non-delimiters or delimiters; we refer to the former as a *token expression* and the latter as a *delimiter expression*. In addition, the sequence must alternate between token and delimiter expressions, or else the tokenization rule could be violated. Finally, a schema may include non-capturing prefix and suffix expressions that are used to contextualize the schema.

CLP ships with a few default schemas that we have found are effective in capturing most variables. Specifically, we have a schema each for non-dictionary integer and floating point values. In addition, we have a schemas that match any token with a digit or any token preceded by an equals sign. Finally, we treat most non-alphanumeric characters as delimiters except for a few like underscores and periods.

5 Compressed Persistent Caching

Our experience with CLP shows that it is typically bottlenecked by I/O. Although, the dictionaries and segment index help to avoid much of this I/O, queries that match rare log types can still end up reading an entire segment. Thus, we designed a caching mechanism to improve the performance of these queries.

Consider a segment with two log types: *ltA* comprising 90% of messages in the segment and *ltB* comprising 10% of messages. A query for either log type without applying any filtering requires reading the entire segment since *ltA* and *ltB*'s messages are interspersed. A query for *ltB* would read

90% more data (i.e., those belonging to *ltA*) than necessary, and a query for *ltA* would read 10% unnecessary data.

One possible solution is to sort the log types in each segment, but this introduces two problems. First, since the segment is compressed as a single stream, if a log type's messages start in the middle of the segment, queries for that log type will require decompressing all messages before it. Second, since messages are no longer ordered as in the raw log file, each message would also need to store its position in the original log file so that we could maintain lossless decompression.

Another solution is to store each log type in its own segment, but this too introduces complications: For example, compression performance will be decreased since CLP will have to repeatedly open and close several segments, one for each log type in a file (in reality, the number of such single-log-type segments may exceed the number of open files per process that the OS allows, preventing CLP from keeping the files open at the same time). As a result, we do not use this strategy as our primary storage method but rather as a cache.

CLP's policy is to cache recently read, *infrequent* log types by storing each log type in its own segment. These segments are created in addition to the existing segments compressed by CLP, instead of replacing them. Specifically, when a user runs a query containing one or more log types, CLP will attempt to cache messages with those log types (henceforth referred to as caching log types) if the query does not return too many messages. The specific number of messages is configurable and will depend on the user's performance requirements and system resources. Only infrequent log types are cached because they offer both the best speedup and least additional storage cost.

When CLP tries to cache a new log type and the cache is full, it will need to decide whether to evict an existing log type or discard the new log type. Its policy is to evict log types that 1) have not been recently queried, and 2) that contain more messages than the new log type to be cached. The first condition is necessary to ensure that the cache does not eventually become filled with the most infrequent log types due to the second condition. The duration that is considered recent can also be configured by the user and again depends on their deployment. Note that in practice, a user's query may match multiple log types, in which case, CLP creates a persistent cache file for each log type independently.

The format of each log type segment in the cache is similar to the regular segments, but with a few key differences. First, there is no log type column since the entire file has the same log type. Second, each message additionally includes a log file path identifier, a timestamp format identifier and an optional message number. These identifiers are necessary since messages in this file may come from many different log files. Finally, the log type segment is named in a way that it can be easily referenced using the corresponding log type ID.

With the cache enabled, CLP processes a query in two parts: one for the log type cache and one for the non-cache segments. To determine which log types are in the cache, CLP simply uses each log type's ID to locate its corresponding segment. If one exists, it is searched like any other segment and the log type is removed from the query. Then, any remaining uncached log types are searched for in the non-cache segments, completing the query.

Name	Files	Log Messages	Size (GB)
/var/log/*-7GB	9,335	63,197,765	7
OpenStack-33GB	810	74,188,154	33
Apache-6TB	5,293	26,135,489,184	6,304
Hadoop-14TB	18,170	57,323,941,112	14,510

Table 3: The log datasets used to evaluate CLP.

6 Data Scrubbing and Obfuscation

A useful feature of CLP's design is the ability to quickly obfuscate data (e.g., to comply with data privacy laws) using the compression dictionaries. Consider a case where a user wants to obfuscate a username, "johnsmart9", from all log messages. Since this username will be stored in the variable dictionary, it can be easily replaced with an obfuscated string like "x93n4f9". Similarly, if a user wanted to hide all usernames from a certain log type, they could simply modify the log type in the dictionary to contain a generic username in place of the actual username. Moreover, since the dictionaries are typically much smaller than the segments or the raw data, these replacement operations will be much faster than they would be if the logs were not compressed.

7 Evaluation

CLP has been used to compress petabytes of logs from hundreds of different applications, and we have verified that its compression is lossless in all cases. Our evaluation focuses on CLP's performance. Specifically, we explore: 1) CLP's compression ratio and speed; 2) CLP's search performance; and 3) CLP's scalability and resource efficiency.

7.1 Experiment Setup

Table 3 shows the log corpuses used in our evaluation. The /var/log/* corpus contains all of the logs in the `/var/log/` directory generated by a cluster of more than 30 Linux servers over the past six years. The OpenStack-33GB log set was gathered by running the cloud scalability benchmark tool, Rally [45], on top of OpenStack. Apache-6TB contains Apache httpd access logs collected over a 15-year period by the U.S. Securities and Exchange Commission's EDGAR system [43]. The Hadoop-14TB logs were generated by three Hadoop clusters, each containing 48 data nodes, running workloads from the HiBench Benchmark Suite [25] for a month.

Note that the datasets generated by benchmarking tools may be artificially uniform, as benchmarks do not always capture the randomness of real-world deployments. However, this should not affect our claims since we compare CLP relative to other tools on the same datasets.

Experiments were performed on a cluster of 16 Linux servers connected over a 10GbE network, each with an eight-core Intel Xeon E5-2630v3 processor and 128GB of DDR4 memory. Unless otherwise specified, all data is stored on a 3TB (labelled 3TB, real-size 2.73TB) 7200RPM SATA HDD connected to each machine.

We compare CLP with Gzip 1.6, Zstandard 1.3.3, and 7z 16.02 for compression, in addition to ripgrep 12.1.0, Elasticsearch 7.8.0 and Splunk Enterprise 8.0.3 for search. All versions were the latest releases from Ubuntu 18.04's package manager, Elastic, and Splunk at the time of the experiments.

We use ripgrep to search the archives produced by general-purpose compressors. ripgrep is a grep-derivative designed with aggressive system and algorithmic optimizations that allow it to outperform grep significantly. Moreover, ripgrep offers advanced parallelization and can directly search the contents of Gzip, Zstandard, and 7z-lzma archives.

We modified Elasticsearch and Splunk Enterprise's default configuration only enough to ensure they matched CLP's search capabilities without storing more data. In practice, we expect a user in need of CLP's capabilities would do the same. Recall CLP can perform wildcard searches on log messages as well as filtering based on file paths and time ranges. We do not explicitly evaluate the filtering features but supporting them increases the amount of data that Elasticsearch and Splunk Enterprise store in their indexes.

Splunk Enterprise's default configuration matches almost all of CLP's capabilities with the exception of wildcard searches. Due to the way Splunk Enterprise indexes tokens with punctuation like "`AA-BB-123`", it cannot perform queries with wildcards in the middle of the token like "`AA*23`" [38].

For Elasticsearch, we first had to configure an index before logs could be ingested. Typically, Elasticsearch's ingestion tool, Filebeat, configures a default index; but because Filebeat was not fast enough for our use, we ingested logs using our own parser. Elasticsearch indexes are configured with a set of fields, each of which has a type indicating how it should be indexed. Elasticsearch only supports wildcard searches on fields with type "keyword". Alternatively, Elasticsearch can perform full text searches on "text" type fields, but this does not match CLP's capabilities. For example, Elasticsearch's default tokenizer ignores stop words like "and", whereas CLP's wildcard search does not. We initially tested indexing the content of each log message as a keyword-field but found that this required 58% more storage, took 7% longer to ingest, and was 4750% slower to search compared to indexing the content as a text-field. Elastic also recommends indexing unstructured content as a text-field [13]. Thus, we configured Elasticsearch's index with three fields: message_content with type "text," timestamp with type "date," and file_path with type "keyword." Following Elasticsearch's best practices [12], we set the max heap size to 30GB for its Java Virtual Machine.

For CLP, we configured the persistent cache to store less than 0.01% of all compressed messages and used the general-purpose default schemas to parse the logs in all experiments.

7.2 Compression Speed and Ratio

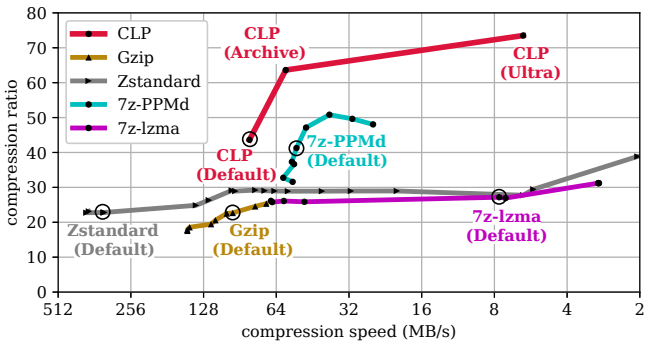


Figure 6: Compression ratio and speed trade-off for CLP and general-purpose compressors. CLP's compression generally exceeds all other compressors and its current speed is competitive.

We first examine CLP's tradeoff between compression ratio and speed compared to general-purpose compressors. Each tool was used to compress a 30GB subset of the Hadoop corpus. In addition, all data was read from and written to a tempfs RAM disk in order to minimize I/O overhead and fully expose the tools' algorithmic performance. For each tool, we measured its single-threaded compression speed since not all tools support multiple threads; for those that do, we observed a minor decrease in per-core performance when running them with multiple threads rather than independent processes. Finally, we vary each tool's compression level from low to high.

Figure 6 shows the compression ratio and speed for the evaluated tools. Overall, CLP achieves higher compression than Gzip or Zstandard. Compared to PPMd, a natural-language optimized compressor, CLP slightly exceeds its compression at their default levels and significantly exceeds it at higher compression levels. In addition, CLP's default level offers performance competitive with Gzip's default level but with double the compression. We use CLP's default mode for all remaining experiments.

To compare CLP's compression speed with Elasticsearch and Splunk Enterprise's ingestion speed, we reuse the previous experiment except each tool is configured to use the number of threads that provides the highest possible throughput. In contrast with general-purpose compressors, Elasticsearch and Splunk Enterprise are designed as multithreaded tools, so their performance generally suffers when they are forced to use a single thread. Figure 7 shows the results: Overall,

#	Query	# results	# log types	# dict. vars.
Log type queries contain no variables, so CLP only searches the log type dictionary and log type columns.				
Q1	“ `org.apache.hadoop.hdfs.server.common.Storage:↩ Analyzing storage directories for bpid` ”	12	1	0
Q2	“ `org.apache.hadoop.hdfs.server.datanode.DataNode:↩ DataTransfer, at` ”	2,026	1	0
Q3	“ `INFO org.apache.hadoop.yarn.server.nodemanager.↩ containermanager.container.ContainerImpl: Container` ”	513,893	12	0
Q4	“ `DEBUG org.apache.hadoop.mapred.ShuffleHandler:↩ verifying request. enc_str=`”	810,033	84,922	0
Non-dictionary integer queries contain an integer non-dictionary variable, so the variable column is searched in addition to the log type search.				
Q5	“ `to pid 21177 as user` ”	12	3	0
Q6	“ `10000 reply:` ”	13,064	24	0
Q7	“ `10 reply:` ”	279,284	24	0
Non-dictionary float queries: contain a float non-dictionary variable.				
Q8	“ `178.2 MB` ”	2,800	3	0
Q9	“ `1.9 GB` ”	1,623,002	5	0
Dictionary variable queries contain dictionary variable, so log type dict., variable dict., and variable columns are searched.				
Q10	“`job_1528179349176_24837`”	51	89,258	3
Q11	“`blk_1075089282_1348458`”	4,261	89,258	3
Q12	“`hdfs://master:8200/HiBench/Bayes/temp/worddict`”	178,076	9	1
Non-matching query: contains a potential log type but does not match any log type.				
Q13	“ `abcde` ”	0	0	0

Table 4: The queries used in our search-performance evaluation, grouped based on how CLP processes them. The quotation marks in each query are used to highlight any leading or trailing spaces and are not part of the query. Similarly, the ↩ symbol indicates a newline that is not part of the query but was inserted for typesetting.

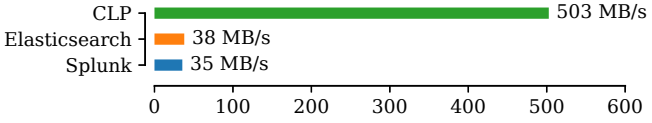


Figure 7: Single-node ingestion speed of 30GB of Hadoop logs for CLP, Elasticsearch, and Splunk Enterprise. CLP far exceeds their ingestion speed.

CLP is able to ingest the corpus at least an order of magnitude faster than both Elasticsearch and Splunk Enterprise.[2]

We also compare the tools' compression on larger datasets. To measure Elasticsearch and Splunk Enterprise's compression ratio, we shutdown the tools to ensure any in-memory data was persisted and then measured the size of their data directory on disk. For Splunk Enterprise, we used a subset of the terabyte-scale datasets since our evaluation license limited the amount of data we could ingest per day. Figure 8 shows the results for Gzip, Zstandard, and 7z-lzma using their default settings in addition to Elasticsearch, Splunk Enterprise, and CLP. For all corpuses, CLP significantly outperformed all of the evaluated tools. On average, using the default compression mode without customized parsing rules, CLP's average compression ratio is 32. CLP's advantage is evident on the

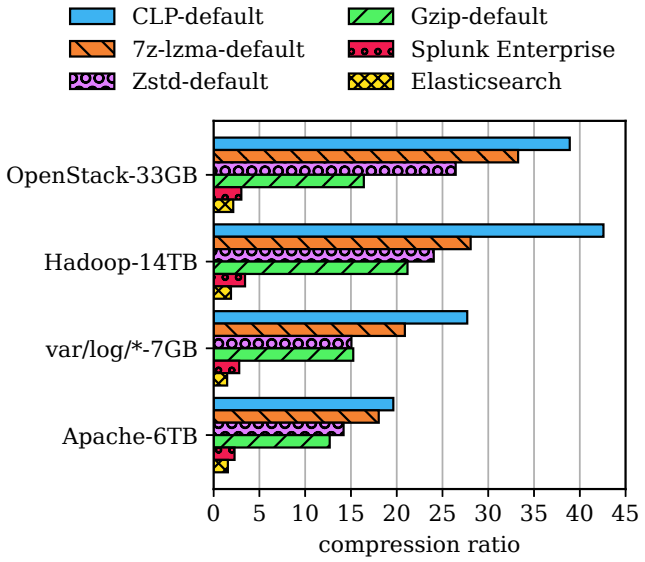


Figure 8: Compression ratio of tools on different corpuses. CLP exceeds the ratio of the others.

[2]Elasticsearch's 38 MB/s ingestion speed can only be achieved when we replaced its own log parsers (Logstash and Filebeat) with CLP's, as they were unable to ingest faster than 1 MB/s. We ported CLP's log parser to connect to Elasticsearch's REST API endpoint.

OpenStack and Hadoop datasets where the log formats contain a large amount of unstructured natural language. The Apache-6TB corpus has the worst compression ratio since the messages largely contain variable values.

In contrast, log indexing tools Elasticsearch and Splunk Enterprise have significantly lower compression ratios at 1.75 and 2.86 respectively. This means that their on-disk data structures, including both the index and the compressed logs, are on the same order of magnitude as the uncompressed log (57% and 35% respectively). (Both tools recommend users keep searchable data structures on fast storage such as an SSD.)

On average, across all experiments, CLP's log type dictionary accounted for 0.03% of the total compressed size and the variable dictionary accounted for 1.07%. All other CLP metadata files were negligible in size.

7.3 Search Performance

Commonly used log search tools fall into two categories: index-based search (e.g., Splunk Enterprise and Elasticsearch) and sequential search (i.e., variations of the grep tool). In comparison, CLP is a mixture. Its dictionaries serve the purpose of lightweight indexing (with the key difference being that CLP's dictionaries *deduplicate* repetitive data instead of duplicating data into a separate index), and when combined with the segment index as well as each file's metadata, CLP can skip files or jump to a specific file in a specific segment. On the other hand, CLP still searches columns sequentially. Thus, we compare CLP with tools from both categories.

We benchmark each tool using the set of queries in Table 4, specific to a 258GB subset of the Hadoop-14TB corpus. In designing the set of queries, we initially tried to make them representative, but faced two challenges. First, real-world datasets and workloads are diverse, meaning we would need a large number of queries to sufficiently represent most use cases. Second, any query set will likely be biased towards or against a tool, and so the benchmark would neglect the strengths and weaknesses of some tools over others. Instead, we designed the queries simply to test CLP by exercising its different execution paths, highlighting its strengths and weaknesses. For each query type, we used multiple queries that differ in the number of results they return from a few to many.

We used a 258GB subset of the Hadoop-14TB corpus since we were limited to one node for several of the evaluated tools. Specifically, our Splunk Enterprise evaluation license does not support distributed searches and ripgrep is a single-node tool. Conversely, we could not evaluate the full corpus on one node for Elasticsearch and Splunk Enterprise since they require more storage than the size of the hard drive attached to each machine.

In designing each query, we also had to ensure that all tools would return the same result set for each query. As explained in §7.1, Elasticsearch does not support precise substring searches on text-fields because it indexes a message by ignoring elements like punctuation. So an Elasticsearch query that includes punctuation may return results which both include and do not include the punctuation. As a result, we only chose queries where differences in interpretation did not affect the results returned. Similarly, because Splunk Enterprise and Elasticsearch cannot accurately support wildcard searches (§7.1), the queries do not explicitly contain wildcards. (CLP's wildcard handling is still exercised as it implicitly adds wildcards to the beginning and the end of each query.)

We ran each query 10 times and report the average of all runs. To emulate searching cold data stored on low-cost storage (hard drives or network storage), the file system page cache is cleared and the tools are restarted (to ensure in-memory caches are cleared) before each run. We also ran each tool with a varying number of threads, and report the configuration that yielded the fastest completion time.

Figure 9 shows the search performance of CLP and the other tools. The averaged normalized completion times of CLP, Elasticsearch, and Splunk Enterprise, are 1x, 1.3x, 4.2x respectively, hence CLP outperforms both. In addition, CLP is faster for queries that return a lot of results (i.e., Q3, Q4, Q7, Q9, and Q12), and competitive for queries that return few results. In queries where CLP is slower, Elasticsearch performed 6–22x less I/O, suggesting its gains are as a result of using search indexes.

Figure 9 also shows CLP's performance when a search is served from its persistent cache. To evaluate CLP's persistent caching, we ran each query twice—once to build the cache, and again to evaluate its performance with the cache. The cache was purged between queries to ensure the next query was not affected by prior caching. The six queries which were persistently cached (Q1–Q5 and Q12) received an average speedup of 43x and a median speedup of 8.64x. Two of those six queries which were previously 4x slower than Elasticsearch are now 5x and 51x faster than Elasticsearch, respectively. Under this configuration, CLP is faster than both Splunk Enterprise and Elasticsearch in every persistently cached query. This shows that the persistent cache can make CLP even more competitive with a negligible effect on compression ratio.

Splunk Enterprise and Elasticsearch also have caching mechanisms but they provide different functionality than CLP. In particular, Elasticsearch and Splunk Enterprise use the entire query (query phrase, timestamp filter, and so on) as the cache key, so only an identically repeated query benefits from the cache. In contrast, CLP's cache key is a log type, so new queries can benefit from the cache if they encompass a cached log type. Also, Elasticsearch and Splunk Enterprise's caches are not persistent.

Finally, Figure 9 shows that CLP is able to exceed the performance of every ripgrep-compressor combination for every query. Analyzing the machine's usage shows that Zstandard is being bottlenecked by disk I/O while both 7z-lzma and Gzip are bottlenecked by the CPU.

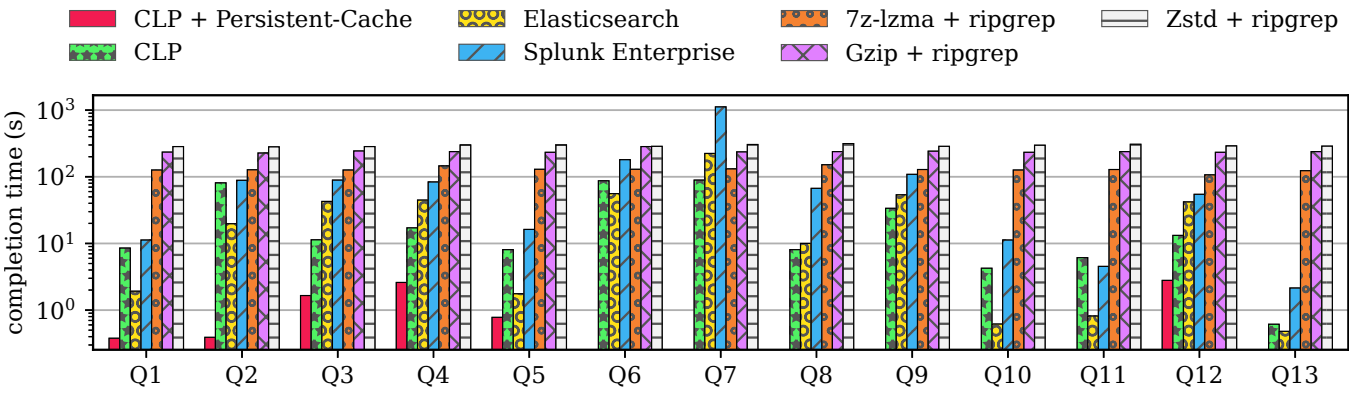


Figure 9: Search performance of CLP, Elasticsearch, Splunk Enterprise, and popular compressed sequential search combinations. CLP is faster for longer queries and competitive for shorter queries. CLP's cache greatly improves its competitiveness.

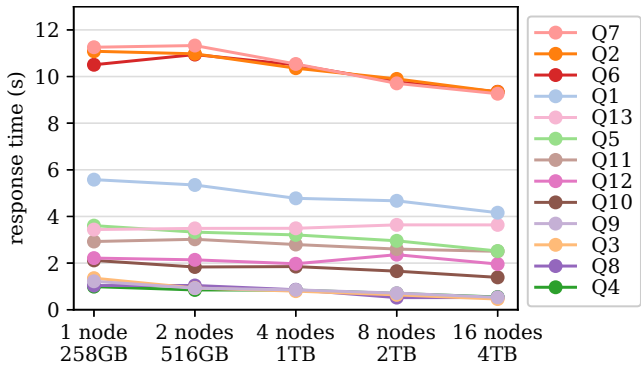


Figure 10: Response time of queries for CLP when both data and resources were horizontally scaled from 1 to 16 nodes.

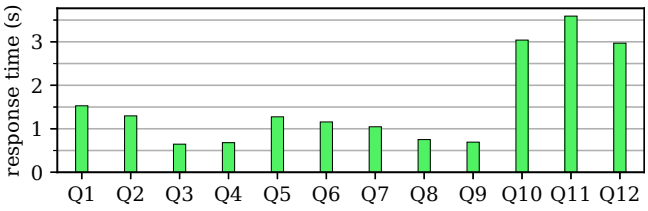


Figure 11: CLP's query response time on a petabyte log corpus. The response time continues to exhibit the pattern observed in the horizontal scalability experiment.

7.4 Horizontal, Vertical, and Capacity Scaling

Since CLP's archives are designed to be independent (§2.4), compression and search are embarrassingly parallelizable tasks. Figure 10 shows that as we scale horizontally, adding more nodes that contain an equal amount of data, CLP's search response time stays nearly constant. Response time is measured from when a query is entered to when the first matching result is returned (in the case of no matching result, it is the query completion time). We show response time instead of completion time because 1) when the output is large, the completion time will be bottlenecked by how fast the user's client can receive results, and 2) in those scenarios, users typically search and refine their queries before arriving at a small amount of output. Nevertheless, completion times also stay nearly constant except for Q3, Q4, Q9, and Q12, whose completion time grows linearly with the output size.

We also repeated the previous experiment with a petabyte of data to evaluate CLP at the scale of logs produced by large internet companies. Since the 3TB hard drives attached to each machine did not have enough free space to store the data, the archives were instead stored on a distributed file system (MooseFS [10]) running on commodity hard drives. The results in Figure 11 show CLP still maintains low response time, but the ordering (by response time) of queries differs from the the previous experiment. This is because CLP was I/O-bound in the previous experiment whereas in the current experiment, MooseFS parallelizes I/O requests across multiple drives, so CLP becomes more CPU-bound. We omitted Q13 from the figure; its response time is 140 seconds. Q13 represents a worst case for CLP as its response time is the same as the completion time, because the search returns "no result." It took 140s to search all log type dictionaries of over 61,000 archives with only 256 threads. In contrast, the previous experiment had one search thread per archive. Overall, the results show that CLP can indeed scale in large Internet companies, while reducing storage costs.

Using MooseFS, we also measured CLP's ingestion speed at the petabyte scale. By adding eight additional nodes to the existing 16-node cluster, CLP was able to reach an ingestion speed of about one petabyte of raw logs per day, exhausting each hard drive's bandwidth.

To evaluate vertical scalability, we tested CLP's search performance on the Hadoop-14TB corpus with *a single thread on a single data-node*. The fastest completion time was for a non-existing result query which took just under a minute, and most queries started emitting results within 10 seconds.

8 Related Work

We discuss three categories of related work: (1) log compression, (2) searching the compressed form of general-purpose indexes for textual data, and (3) log search. Existing log compression tools do not enable search (on compressed data). Singh and Shivanna's [35] method of log compression also aims to deduplicate static text from variable values. They rely on the applications' source code to generate patterns, akin to our concept of a log type. Variables are annotated with the variable's primitive type such as integer or long so they can be encoded into binary bits stored separately. However, they do not propose any search algorithms on the compressed data. In addition, the compression in their work is not entirely lossless in the sense that a number's precision is not encoded. For example, the value "`1.000`" can only be stored as an equivalent floating point number, failing to take into account the number's zero padding.

Separating highly redundant static text from variable values has also been used to design highly efficient log printing libraries. For example, both NanoLog [47] and Log20 [48] only log an ID for each log type at runtime, and reconstruct the textual log message in post-execution phases. Furthermore, some logging systems [34] directly output binary log messages, representing each log type with an ID. While CLP is designed to compress text logs, its search algorithms can be used to search binary logs by associating human-readable static text with each log ID. Hence, users can use CLP's intuitive text search interface to analyze binary logs, as if they are text logs, with minimal storage overhead.

General-purpose text search typically uses indexes such as suffix trees or tries, which will *add* 10-20x the size to the original text data [6, 27]. Compressed forms of these indexes, typically via smart encoding, have been proposed [17–20, 24, 30–33] such that they can be searched without decompression. Succinct [6] further proposed an entropy-based representation of these compressed indexes to further reduce the size of compressed index. However, regardless of how small the index is, it still increases storage space instead of reducing it, and search can only be performed on data that is indexed. In comparison, CLP does not add any additional index; it simply deduplicates the static text and dictionary variables, whereas these works are used to compress indexes.

Several pattern matching algorithms exist for searching data compressed with general-purpose compressors, but none operates on data compressed using an algorithm that practically achieves our compression speed and ratio. Kida *et al.* [26] implemented an algorithm for pattern matching in LZW compressed data, achieving better performance than decompression followed by a search. Similarly, Navarro and Raffinot did the same for LZ78 [49] compressed data. However, LZW has worse compression than CLP while LZ78 uses a prohibitive amount of memory for large data sizes and is more likely to experience dictionary explosion.

Tools like Splunk Enterprise [4] and Elasticsearch [15] allow users to search and analyze logs. They work by treating logs as normal text files and apply standard indexing and search techniques to achieve responsive searches. In contrast, CLP does not need to spend expensive resources to create and maintain additional indexes.

Conversely, Scalyr [3] is a log search tool that uses a "brute-force" approach, instead of using any index. It uses a number of low-level optimizations to achieve a log search speed of up to 1.25 GB/second per core without using any index [28]. It also directly works on raw logs. In comparison, by working directly atop compressed log data, CLP is able to achieve much higher search performance when translated to the raw log size, even when the data is uncached and stored on low-cost HDDs. Through our scalability experiments, we observed that our fastest queries, which return no results purely by scanning through the log type dictionaries, can effectively search through the equivalent of hundreds of gigabytes per second per core from a single HDD.

Grafana Loki [23] makes a trade-off that lies between index-based search tools and Scalyr: it only indexes the labels, i.e., selected fields of the log. Hence the index size is significantly reduced, yet users can only search the labels. Moreover, the index again adds to the storage space.

9 Conclusion

This paper presented the mutually exclusive problem of log archiving and log analytics. We present an end-to-end solution, CLP, that allows users to perform "archivalytics" across their entire history of compressed log messages without the need for decompression. Using an algorithm customized to text logs, CLP is able to achieve higher compression ratio than other compressors while enabling faster search performance than index-based search tools.

Acknowledgements

We thank our shepherd Lalith Suresh and the anonymous reviewers for their insightful comments. Michael Stumm provided invaluable suggestions throughout the project. This research was supported by the Canada Research Chair fund, a Connaught Innovation Award, a Huawei grant, the McCharles fellowship, a Mitacs grant, NetApp fellowships, NSERC discovery grants, and a VMware gift.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[2] Elastic. https://www.elastic.co/, 2021.

[3] Scalyr. https://www.scalyr.com/, 2021.

[4] Splunk. https://www.splunk.com/, 2021.

[5] 104th United States Congress. Title 45 CFR 164.316. In *United States Code of Federal Regulations*. 2003.

[6] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 337–350. USENIX Association, May 2015.

[7] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 445–460. USENIX Association, October 2018.

[8] Brian Knox. Diving in The Deep End: Logging and Metrics at DigitalOcean. Video, November 2013. https://www.elastic.co/elasticon/tour/2015/new-york/logging-and-metrics-at-digital-ocean.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218. USENIX Association, November 2006.

[10] Core Technology Sp. z o.o. MooseFS, 2021. https://moosefs.com/products/#moosefs.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04. USENIX Association, December 2004.

[12] Elastic B.V. *Setting the Heap Size*, May 2021. https://www.elastic.co/guide/en/elasticsearch/reference/7.8/text.html.

[13] Elastic B.V. *Text Data Type*, May 2021. https://www.elastic.co/guide/en/elasticsearch/reference/7.8/text.html.

[14] Elastic N.V. Annual Report. https://www.sec.gov/ix?doc=/Archives/edgar/data/0001707753/000162828020009982/estc-20200430.htm, June 2020.

[15] Elasticsearch B.V. Elasticsearch 7.8.0, June 2020. https://www.elastic.co/downloads/past-releases/elasticsearch-7-8-0.

[16] Facebook, Inc. Zstandard. https://facebook.github.io/zstd/.

[17] Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS 2000, pages 390–398. IEEE, November 2000.

[18] Paolo Ferragina and Giovanni Manzini. An Experimental Study of a Compressed Index. *Information Sciences*, 135(1-2):13–28, June 2001.

[19] Paolo Ferragina and Giovanni Manzini. An Experimental Study of an Opportunistic Index. In *Proceedings of the 12th Annual SIAM Symposium on Discrete Algorithms*, SODA '01, pages 269–278. ACM, January 2001.

[20] Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM (JACM)*, 52(4):552–581, July 2005.

[21] Free Software Foundation, Inc. GNU Gzip, August 2020. https://www.gnu.org/software/gzip/.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, October 2003.

[23] Grafana Labs. *Loki Documentation*, May 2021. https://grafana.com/docs/loki/latest/.

[24] Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[25] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *Proceedings of the 26th International Conference on Data Engineering Workshops*, ICDEW 2010, pages 41–51. IEEE, March 2010.

[26] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, Masamichi Miyazaki, and Setsuo Arikawa. Multiple Pattern Matching in LZW Compressed Text. In *Proceedings of the Data Compression Conference*, DCC '98, pages 103–112. IEEE, March 1998.

[27] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience*, 29(13):1149–1171, November 1999.

[28] Steve Newman. Searching 1.5TB/Sec: Systems Engineering Before Algorithms, May 2014.

https://www.scalyr.com/blog/searching-1tb-sec-systems-engineering-before-algorithms/.

[29] Cloud Object Storage | Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/.

[30] Kunihiko Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *Proceedings of the 11th International Conference on Algorithms and Computation*, ISAAC '00, pages 410–421. Springer, December 2000.

[31] Kunihiko Sadakane. Succinct Representations of LCP Information and Improvements in the Compressed Suffix Arrays. In *Proceedings of the 13th Annual SIAM Symposium on Discrete Algorithms*, SODA '02, pages 225–232. ACM, January 2002.

[32] Kunihiko Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, September 2003.

[33] Kunihiko Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589—-607, December 2007.

[34] Kedar Sadekar. Scalable Logging and Tracking, June 2012. https://netflixtechblog.com/scalable-logging-and-tracking-882bde0ddca2.

[35] Pranay Singh and Srikanta Shivanna. Method and System for Compressing Logs. *US Patent 9,619,478*, April 2017.

[36] Splunk Inc. Annual Report. https://www.sec.gov/ix?doc=/Archives/edgar/data/0001353283/000135328320000008/a01312010k.htm, March 2020.

[37] Splunk Inc. Splunk® Enterprise 8.0.3, April 2020. https://www.splunk.com/en_us/download/previous-releases.html.

[38] Splunk Inc. *Wildcards*, February 2021. https://docs.splunk.com/Documentation/Splunk/8.0.3/Search/Wildcards.

[39] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553—564. ACM, August 2005.

[40] James A. Storer and Thomas G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[41] The Apache Software Foundation. HDFS Architecture, July 2020. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[42] The Apache Software Foundation. Apache Lucene, 2021. https://lucene.apache.org/.

[43] The Division of Economic and Risk Analysis. EDGAR Log File Data Set, June 2017. https://www.sec.gov/dera/data/edgar-log-file-data-set.html.

[44] The Kernel Development Community. *FUSE*, May 2021. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[45] The OpenStack Foundation. Rally, 2021. https://opendev.org/openstack/rally.

[46] Vijay Samuel. Monitoring Anything and Everything with Beats at eBay. Video, February 2018. https://www.elastic.co/elasticon/conf/2018/sf/monitoring-anything-and-everything-with-beats-at-ebay.

[47] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 335–350. USENIX Association, July 2018.

[48] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581. ACM, October 2017.

[49] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

Polyjuice: High-Performance Transactions via Learned Concurrency Control

Jiachen Wang[†*], Ding Ding[‡*], Huan Wang[†], Conrad Christensen[‡], Zhaoguo Wang[†], Haibo Chen[†], and Jinyang Li[‡]

[†]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[†]Shanghai AI Laboratory
[†]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[‡]Department of Computer Science, New York University

Abstract

Concurrency control algorithms are key determinants of the performance of in-memory databases. Existing algorithms are designed to work well for certain workloads. For example, optimistic concurrency control (OCC) is better than two-phase-locking (2PL) under low contention, while the converse is true under high contention.

To adapt to different workloads, prior works mix or switch between a few known algorithms using manual insights or simple heuristics. We propose a learning-based framework that instead explicitly optimizes concurrency control via offline training to maximize performance. Instead of choosing among a small number of known algorithms, our approach searches in a "policy space" of fine-grained actions, resulting in novel algorithms that can outperform existing algorithms by specializing to a given workload.

We build Polyjuice based on our learning framework and evaluate it against several existing algorithms. Under different configurations of TPC-C and TPC-E, Polyjuice can achieve throughput numbers higher than the best of existing algorithms by 15% to 56%.

1 Introduction

Concurrency control (CC) algorithms lie at the foundation of modern database systems [18]. A CC algorithm synchronizes a transaction's access to storage objects to maximize concurrent execution while guaranteeing correctness. As today's database systems are no longer disk-bound, the CC algorithm in use becomes crucial to a database's performance.

Traditional CC algorithms, such as two-phase-locking (2PL) [17] and optimistic concurrency control (OCC) [28], take fixed algorithmic steps regardless of the workload. Thus, it comes as no surprise that the relative performance of different algorithms varies depending on the transaction workload. Figure 1 shows the throughput of 2PL, OCC and IC3 [61] on a multi-core database under the TPC-C workload with a varying number of warehouses. OCC has the highest throughput under low contention (more warehouses) while the other two outperform OCC under high contention (fewer warehouses). Similar results have also been reported by others [68].

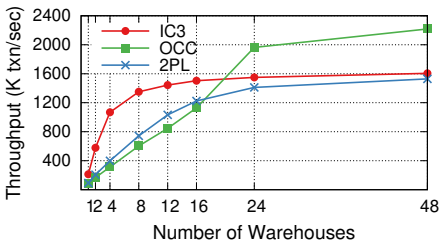


Figure 1: IC3, OCC, 2PL performance on TPC-C, 48 threads.

To adapt to different workloads, prior works propose a federated approach by simultaneously supporting a small number of existing CC algorithms, including 2PL and OCC. These systems require users to partition the workload either by data [55] or by transaction type [49, 53, 66]. The decision of which algorithm to use for each partition is either based on manual insights [49, 53, 66] or simple runtime metrics [55]. While this federated approach can improve performance, it has limitations. First, by limiting itself to using a small number of known algorithms, it lacks the flexibility to customize concurrency control to fully exploit the workload. Second, by relying on manual insights or simple heuristics, it lacks a systematic solution to optimize concurrency control for performance.

This paper presents a learning-based framework to optimize concurrency control for a given workload. We assume that the workload is known a priori such as past workloads, e.g. in the form of stored procedures. To enable learning, we design a "policy space" of fine-grained actions (a.k.a. algorithmic steps): each policy can be viewed as a CC algorithm that uses specific actions to synchronize different data accesses made by different transactions. All policies perform an explicit validation before transactions commit to ensure serializability. We use offline training to learn the highest performing policy

∗ Jiachen Wang and Ding Ding contributed equally to this paper.

for a given workload. This framework is expressive: it can learn new CC algorithms as well as existing ones. It also allows explicit optimization for performance via systematic searches of the policy space.

We have realized the design for learned concurrency control in a system called Polyjuice for multi-core in-memory databases. The core technical challenge of Polyjuice is to design the policy space. Inspired by reinforcement learning, we view each policy as a function that maps each state (i.e., the execution context of a data access) to actions that control the interleavings of accesses made by concurrent transactions. In Polyjuice, the state specifies what type of transaction is being used and which of its accesses are under execution. The actions support multiple ways of interleaving control, including deciding which data version to read, whether to expose an uncommitted write, how long to wait before access, and whether to perform early validation before commit.

Polyjuice represents each policy using a table: the rows correspond to different states and the columns correspond to different kinds of actions. Polyjuice uses evolutionary algorithm based training to search the policy space for the policy that has the highest commit throughput for a given workload.

We train and evaluate Polyjuice's performance on micro-benchmarks, TPC-C and TPC-E, and compare with existing algorithms, including Silo [57](OCC), 2PL, Tebaldi [53], CormCC [55] and IC3 [61]. Our experiments show that, for TPC-C and TPC-E with moderate to high contention, Polyjuice can find a CC policy whose throughput is better than the best of existing algorithms by 15% to 56%. Detailed analysis shows that Polyjuice can learn an interesting policy that is different than any of the existing algorithms to exploit the workload in subtle ways(§7.3). For workloads with almost no contention, Polyjuice learns the same policy as OCC and incurs 8% slowdown due to its implementation overhead.

As Polyjuice requires offline training, it is not suitable for dynamic workloads that can change rapidly and unpredictably. However, our analysis of an e-commerce website trace shows that real-world workloads are fairly predictable in terms of its peak hour workload characteristics including the likelihood of conflict. This suggests that it is practical to use Polyjuice to optimize a database's peak performance by training on traces of recently observed peak workloads.

In summary, our paper makes the follow contributions:

- We present the first framework to learn concurrency control using a policy space of fine-grained actions.
- We design Polyjuice's policy space according to the framework so that it can encode a variety of existing CC algorithms while allowing the exploration of new ones.
- We show that Polyjuice's policy, represented as a table, can be optimized simply using an evolutionary algorithm.
- Even for the heavily-studied TPC-C benchmarks, Polyjuice can find interesting and novel policies not seen in existing algorithms to improve transaction throughput under moderate to high contention.

2 Background and Motivation

Existing works have realized the inadequacy of using one fixed concurrency control algorithm for different workloads. For the solution, they propose a federated approach of mixing a few (typically 2 or 3) known CC algorithms [53, 55, 60, 66]. In this section, we discuss the limitations of this federated approach and motivate the need for a more expressive learning-based approach.

The federated approach of adapting CC to a workload is characterized by its *coarse-grained* way of mixing different algorithms. Specifically, this approach coarsely partitions the workload. The same CC algorithm is used within a workload partition, while a different algorithm may be used for a different partition. Two ways of partitioning can be found in existing work. CormCC [55] partitions by data: all accesses to data in the same partition use the same CC algorithm. Tebaldi [53] and Callas [66] group (a.k.a. partition) transactions by types: all transactions belonging to the same group (a.k.a. partition) use the same CC for all their data accesses.

The coarse-grained way of mixing CC algorithms is limited in its ability to fully exploit workload characteristics for performance. For example, with CormCC, if transactions T and T' both only access data within the same partition, they would synchronize all of their accesses using the same CC algorithm. Similarly for Tebaldi and Callas, if transactions T and T' are of the same type, they would always use the same CC algorithm. This is not optimal: if different data accesses of T and T' have different contention characteristics, they may be better served by different methods for controlling concurrency.

A second limitation of existing federated CC work is their reliance on manual insights to partition the workload or to determine which CC algorithm to use for each partition. Callas and Tebaldi manually assign transactions to groups and choose a specific CC algorithm for each group. CormCC partitions the TPC-C workload by warehouse based on manual insights and uses simple runtime statistics (e.g. read/write ratio) to decide which CC algorithm to use for each partition.

Our approach. We aim to optimize CC for a given workload in a *fine-grained* way using a learning-based approach. Instead of partitioning the workload and using a single CC algorithm for all data accesses within the partition, we propose to allow each data access to use one of many different fine-grained "actions" to mediate potentially conflicting accesses. When deciding what action(s) to take to maximize performance, we are not concerned with correctness; instead, we rely on a separate validation mechanism to abort non-serializable transactions. As fine-grained actions lead to exponentially many choices for a given workload, it is impossible to rely on manual insights to choose the best action(s). A more practical solution is to use a learning-based approach to explicitly optimize the choice of actions for the given workload.

The main challenge of our approach is to design the learning framework with fine-grained actions for concurrency control. Ideally, the framework should be expressive enough to encode **most** existing CC algorithms and to allow the synthesis of new ones. In the next section, we discuss how to design such a learning framework.

3 Learning Concurrency Control

In this section, we examine how to frame concurrency control as a fine-grained learning task.

System settings. Our target setting is an in-memory database running on a single multi-core machine. We assume the kinds of transaction to be run on the database are known a priori, e.g. in the form of stored procedures. A number of existing work also exploit a known-workload in designing CC algorithms [41, 61, 66]. Our work focuses on learning concurrency control for read-write transactions, and reuses existing mechanisms to support logging and snapshot-based read-only transactions [57]. Although our learning framework is general enough to represent multi-version concurrency control (MVCC), our later system design does not support it because existing snapshot-based read-only transactions can already capture much of MVCC's performance benefits.

3.1 The learning framework

Our framework for learning concurrency control is inspired by reinforcement learning (RL). As one of the major branches of machine learning, RL involves learning how to interact with an environment to maximize a numerical reward. The key ingredients in RL are: a *policy* that maps perceived states of the environment to actions to be taken when those states are reached, a *reward* signal that defines the optimization goal, and the *environment* under which the learning system operates. In our context, the policy corresponds to the CC algorithm; the reward corresponds to some performance metric to be maximized; the environment captures the transaction workload and system setup under which the CC operates.

It is straightforward to decide on the optimization objective (a.k.a. reward). In this work, we use transaction throughput. Compared to latency or abort rate, transaction throughput is widely used as the key end-to-end performance metric for in-memory databases.

It is non-trivial to design a "policy space" to represent various CC algorithms. At a high level, a CC algorithm executes a transaction by controlling how its data access can interleave with potentially conflicting accesses from other concurrent transactions. As mentioned previously, we do not attempt to learn how to guarantee correctness. Instead, a learned CC algorithm always invokes a manually-designed validation procedure as part of transaction commit to ensure serializability. What we do learn is a policy that determines what actions to take in order to maximize performance for a given workload. A good CC policy balances how long transactions execute vs. how likely transactions are aborted, resulting in a high reward, as measured by how many transactions successfully commit per second. Aside from the CC policy, how long a database backs off before retrying an aborted transaction can also affect the performance. We separate the backoff policy from the CC policy, and this section focuses on the latter.

The policy space of concurrency control. Taking a page from reinforcement learning, we represent the policy as a mapping from some state of execution to a specific action to take upon encountering that state. Taking different actions in different states allows us to specialize a CC algorithm to optimize for a given workload. Thus, the state space should include information that is necessary to distinguish circumstances that require different actions, e.g. the type of transaction that is making the access, the type of access etc. In a later section (§4.2), we provide a concrete design of the state space. In the rest of this section, we focus on designing the action space.

Ideally, the action space should encompass a set of fine-grained actions that can be mixed and matched to represent many different CC algorithms. These actions can be classified into two categories: 1) actions that control how the data access of concurrent transactions can interleave during transaction execution, and 2) actions that control when and how to perform validation in order to detect whether an executed transaction has violated serializability. Next, we discuss the spectrum of actions available to use in each of the two categories.

Available actions for interleaving control. These actions mediate potentially conflicting data accesses, thereby affecting the set of dependencies that arise among concurrent transactions. There are 3 types of dependencies: write-write $\xrightarrow{ww}$ (a.k.a. write dependency), write-read $\xrightarrow{wr}$ (a.k.a. read dependency), or read-write $\xrightarrow{rw}$ (a.k.a. anti-dependency) [1]. What are the knobs of control that can affect these dependencies?

To discover these knobs in their full generality, let us assume a hypothetical yet still practical database design that keeps track of each read and write access of transactions in a per-object access list, similar to the approach taken in [42, 61]. As a transaction T performs data accesses, it may insert its reads/writes to the corresponding per-object access lists while also updating T_{dep}, the set of transactions that T becomes dependent on. Using this flexible way of tracking dependencies enables a wide range of design choices for interleaving control, as we will see next.

When executing transaction T, a CC algorithm has the following action choices:

- *Read control*. There are two dimensions to these actions:
  1. *Wait*. This can let some dependent transaction $T' \in T_{dep}$ perform its conflicting write earlier than T's read,

	Interleaving control				Validation	
	Read wait	Read version	Write wait	Write visibility	Early validation	Validation method
2PL*	Until T_{dep} commits	latest committed	Until T_{dep} commits	Yes	Yes	n/a
OCC [28]	No	latest committed	No	No	No	physical cts
TicToc [69]					No	logical cts
Sundial [70]	No	latest committed	Until T_{wdep} commits	No	No	logical cts
Callas RP [66] IC3 [61], DRP [41]	Until T_{dep} finish certain access	latest un-committed	Until T_{dep} finish certain access	piece-end	piece-end	n/a or physical cts
MVTSO [2] (MVCC)	Until $T' \in T_{wdep}$ commits if $ts(T') < ts(T)$	largest committed $< ts(T)$	No	Yes	Yes	physical ts

Table 1: The choices made in existing CC algorithms according to the action space described in §3. T refers to the current transaction. T_{dep} refers to the set of transactions that T is dependent on (due to its conflicting access so far). T_{wdep} is the subset of T_{dep} whose writes have conflicted with T. $ts(T)$ refers to the timestamp assigned to T by MVTSO [2].

resulting in $T' \xrightarrow{wr} T$. Otherwise, a dependency cycle may arise with $T \xrightarrow{rw} T'$, resulting in aborts.

2. *Which version of data to read, including either committed or uncommitted version.* This amounts to choosing which location in the access list to insert the read, thereby affecting dependencies. Specifically, since a read returns the latest write w before itself in the access list, there is a write-read dependency, $T' \xrightarrow{wr} T$, for every T' whose write appears before this read in the list. Additionally, a read also results in a set of read-write dependencies $T \xrightarrow{rw} T'$, for every T' whose write appears after this read in the list.

- *Write control.* There are two dimensions to these actions:
  1. *Wait*. The rationale for this action is similar to that for reads.
  2. *Whether or not to make this write visible to the future reads of other transactions.* The write is buffered if it is not exposed. Otherwise, this write as well as all of T's previously buffered writes are made visible by appending them to the corresponding per-object access lists. The cumulative way of exposing writes makes sense because otherwise, any transaction that has read this but not a previous write of T would violate serializability and get aborted. Unlike a read, there's no flexibility to insert a write in any location but the end of the list; this is because we cannot allow a write to affect past reads. Exposing a write does not imply that uncommitted data will be read because transactions can choose to read committed versions only. In terms of the resulting dependencies, exposing a write causes $T' \xrightarrow{ww} T$ or $T' \xrightarrow{rw} T$ for any T' whose write or read appears before this write in the list.

Available actions for validation. Actions in this category can control two aspects of validation:

- *When to validate.* A transaction may validate its accesses at any time during execution, instead of only at commit time. Early validation can abort a transaction quicker to reduce wasted work.
- *How to validate.* The most precise form of validation is to explicitly check whether committing transaction T would form dependency cycles with other committed transactions [42]. However, such graph-based validation is expensive to implement for in-memory databases. A practical alternative is OCC-style validation [28,57] which uses each transaction's physical commit-timestamp (cts) as its serialization order. Although such validation is conservative and has false aborts, it is fast. Prior work has also proposed validation based on logical commit-timestamps [69].

3.2 Decomposing existing CC algorithms

We take a deep dive to study existing algorithms through the the lens of our framework. At a high level, existing algorithms differ from each other by the distinct combinations of action choices they have, even though their choices remain the same regardless of state.

As summarized in Table 1, traditional 2PL [17] and OCC [28] algorithms both read the latest committed data. OCC does not wait to perform any accesses nor does it expose the writes. By contrast, 2PL exposes writes in order to block future conflicting accesses. We can approximate 2PL's blocking behavior by the action choice that makes transaction T wait for all its dependent transactions T_{dep} to commit before its data access. This approximation is slightly less aggressive than that of 2PL, which makes T wait for T' to commit if the current access *will* make T dependent on T'. We use the term 2PL* to refer to 2PL with this approximated blocking. Sundial [70] handles write-write conflicts with 2PL and read-write conflicts with OCC; thus, it blocks write access until all its write dependencies T_{wdep} commit and has no blocking

for reads. As for validation, traditional algorithms do it only at commit time, except for 2PL whose deadlock detection or prevention mechanism can be viewed as a form of early validation done at every access.

Apart from traditional CC, our framework also applies to a class of recently proposed algorithms including Callas RP [66], IC3 [61] and DRP [41]. These algorithms structure each transaction as a series of pieces [50], and try to pipeline the execution of these pieces to enhance performance under contention. As shown in Table 1, unlike traditional CC, they make a transaction's writes visible and allow reads of uncommitted data. Furthermore, they make transaction T wait before an access until T's dependent transactions finish execution up to a certain point, determined by applying a static analysis of the transaction workload.

Although our design for learnable CC (§4) does not support MVCC, we can nevertheless examine MVCC algorithms using our framework. Table 1 shows the actions made by MVTSO [2]. Other MVCC algorithms [30, 46, 67] have similar actions but use different validation methods. Under MVTSO, a transaction reads the largest committed version smaller than its timestamp. Writes are exposed so that future reads by transactions with larger timestamps will wait for this transaction to commit. MVTSO also performs a form of early-validation and aborts T if there exists $T' \in T_{dep}$ such that $T' \xrightarrow{rw} T$ and T' has been assigned a larger timestamp.

Not all CC algorithms can be expressed by our framework. In particular, our framework tracks dependencies and controls the interleaving of data access at runtime, and therefore cannot encode those CC algorithms that pre-define dependencies according to some globally-agreed ordering prior to execution, e.g. Calvin [56], Granola [7], Eris [31] and RoCoCo [42]. Moreover, our framework assumes that each access of the transaction is executed one after another by a single thread, and hence cannot encode algorithms like Bohm [13] that uses multiple threads to execute a single transaction.

4 Polyjuice Design

We design Polyjuice according to the framework of §3. The design consists of two parts: 1) a suitable policy space. 2) a training procedure to optimize the policy for a given workload. This section describes the policy space. The next section (§5) discusses training.

4.1 Overview

System architecture. Polyjuice is a multi-core in-memory database. There is no multi-version support. For each data object, Polyjuice stores the latest committed data as well as a per-object access list. The access list contains all uncommitted writes that have been made visible, as well as read accesses. A transaction uses the access list to track the dependencies for each data access. Polyjuice uses a pool of workers that run concurrently: each worker executes a transaction and commits it according to the learned CC policy, which has been trained offline. Fig. 2 shows Polyjuice's system architecture.

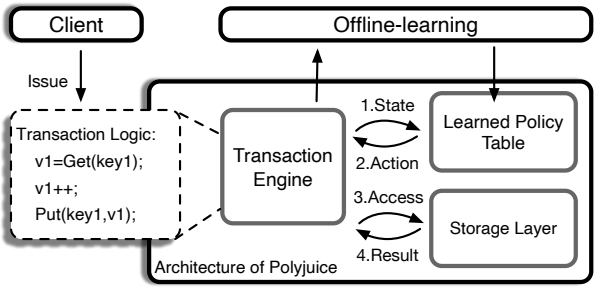


Figure 2: System architecture. Before executing a specific data access in the transaction, Polyjuice consults the learned actions in the policy table (step 1, 2). Then, Polyjuice performs the access in the storage layer according to the actions.

Policy Representation. As discussed in § 3, we consider each learnable CC algorithm as a policy function p that maps from the *state space* (S) to the *action space* (A), $p : S \rightarrow A$. Both the state and action space consists of a number of dimensions; the size of the state/action space is exponential w.r.t. the number of dimensions.

We represent each policy function as a table: there are as many rows in the policy table as there are different states; there are as many columns as there are action dimensions. Such tabular representation is practical only if the state space is not too huge, which is the case in the workloads that we have studied. § 9 discusses the limitation of large state space and potential solutions.

For a given CC policy table, a cell $c_{i,j}$ at row i and column j indicates that for the access with execution context (state) i, the system should take the action given by cell $c_{i,j}$ for action type (a.k.a. dimension) j. In Polyjuice, each cell contains either a binary number for a binary action (e.g. whether to make writes visible or not), or an integer for a multi-valued action (e.g. how to wait for dependent transactions). Fig. 3 shows the CC policy table; details on its rows and columns are explained in §4.2 and 4.3. Polyjuice learns the backoff time for retrying aborted transactions separately (§4.5).

Policy-based Execution. In Polyjuice, the database is given the learned policy table with which to perform concurrency control. To execute a transaction according to the policy, Polyjuice looks up in the policy table at each data access to determine the corresponding set of actions. Some of these actions are to be performed prior to the data access, e.g. whether and how long to wait, while others are to be done after the access, e.g. making a write potentially visible by appending it to the access list. After finishing execution, Polyjuice commits a transaction after performing the final validation to ensure serializability (§ 4.4).

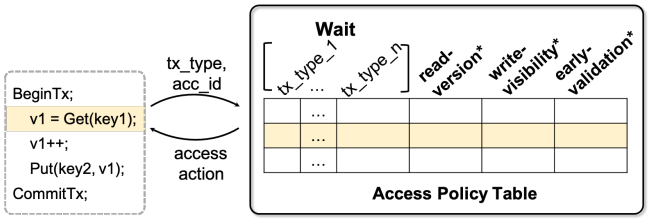


Figure 3: Policy table (* indicates a binary field).

4.2 CC policy: state space

The term *state* is from the RL literature. In our case, state can be viewed as the execution context of the current data access. Ideally, the state space should be able to distinguish execution contexts that are best served by different actions. It should also be limited in size so that the resulting policy table is not too huge and can be searched efficiently during training.

Polyjuice's state space contains the following information:

1. The type of the transaction being executed. For a given workload whose transactions are specified in stored procedures, the type can be identified by the stored procedure name.
2. Which access of the transaction is being executed. We use an integer access-id to identify each access. Access-id is determined by the static code location that invokes the access. Using static information for access-id provides a good trade-off: it can discriminate most accesses while avoiding blowing up the state space.

It is tempting to include other useful information, such as which type of access (read/write/commit) and which data table is being executed. Interestingly, for most workloads, both of these can be uniquely determined by the access-id and thus we omit them from the state space. We have also experimented with adding the contention level of the accessed data to the state space. However, we found that doing so only benefited a few contrived micro-benchmarks. In practical workloads including TPC-C and TPC-E, distinguishing transaction type and access-id is sufficient to capture the main contention characteristics. Even for artificial workloads, it is difficult to find a scenario where including contention level results in noticeable performance improvements. Including contention level makes it possible to differentiate accesses with the same access id. However, Polyjuice's wait action (§ 4.3) cannot take advantage of such differentiation.

Size of state space (a.k.a. number of different states). The state space size determines the number of rows in the CC policy table. Let n be the number of different transaction types in a workload, and d_1, d_2, ..., d_n be the number of static data accesses for transaction of type $1, 2, ..., n$. Then the state space size (i.e. number of different states) is: $d_1 + d_2 + ... + d_n$.

4.3 CC policy: action space

Polyjuice's action space contains knobs in two categories: interleaving control and validation.

Supported actions for interleaving control. There are three classes:

- *Wait*. This action is invoked *before* a read or write. How to specify how long the wait should be? A naive design is to use absolute time intervals, but this makes the wait action sensitive to execution time variations, resulting in fragile policies.

  Since the goal of waiting is to let another potentially conflicting transaction to go ahead with its data access, we quantify how long transaction T should wait by how much progress the transactions that T depends on have made so far. This design is inspired by existing protocols like Callas RP [66] and others [41, 61]. More concretely, we group transactions by type, and measure the execution progress of a transaction type by access-id. The special value NO_WAIT indicates no waiting. Suppose the wait action for transaction type X has access-id a, then transaction T must wait for all T's dependent transactions of type X to finish execution up to and including a. For a workload with n different types of transactions, the wait action consists of n access-ids, one for each transaction type.
- *Read-version*. This action has a binary choice: CLEAN_READ for reading the latest committed version, DIRTY_READ for reading the latest uncommitted (but visible) version. Although there may be more than one uncommitted copy of data, there is no point in reading an earlier version because doing so would result in more dependencies and higher abort likelihood.
- *Write-visibility*. This action is invoked *after* a write access and is also binary: PRIVATE keeps the write in the private buffer, PUBLIC makes all private writes buffered so far visible by appending them to the access list.

Supported actions for validation. Validation always happens before commit (§ 4.4). Polyjuice also supports the action of early-validation, which can occur *after* any read/write. If it's set, this binary-valued action checks if the reads and writes done since the last validation have violated serializability. Earlier accesses, which have passed previous early-validation, are likely to have already been serialized and thus not checked. Early-validation does not guarantee correctness but avoids wasting work by detecting non-serializable access early.

Polyjuice supports the wait action before early-validation. The encoding of the wait action is the same as that for reads/writes. To reduce the action space, we consolidate the two kinds of wait actions into one. In particular, Polyjuice uses the wait action corresponding to the next access-id if early-validation is enabled for the current access-id.

Upon failing early-validation, Polyjuice retries the transaction from the point of its last successful validation. In order

to reduce the cost of the failed validation, we defer appending reads and visible-writes to their corresponding access lists until a successful early-validation. Otherwise, failing early-validation means having to remove previously appended reads/writes from access lists, and to abort transactions that have read those discarded writes. Conceptually, we can separate the decision of early validation from that of appending reads/writes to access lists. However, in our experience, doing so complicates the implementation without improving the final learned CC performance.

Size of action space (a.k.a. number of different action choice combinations per state). Let n be the number of different transaction types in a workload, and $d_1, d_2, ..., d_n$ be the number of static data accesses for transaction of type $1, 2, ..., n$. Then the number of different action choice combinations can be calculated as: $d_1 * d_2 * ... * d_n$(wait choices) $* 2$(read-version) $* 2$(write-visibility) $* 2$(early-validation).

4.4 Validation for correctness

Polyjuice uses an OCC-style physical timestamp-based validation in the final commit phase to ensure correctness. To commit a transaction T with validation, a worker takes 4 steps: 1) it waits for all T's dependent transactions to commit (or abort). 2) it locks each record in T's writeset 3) it validates each record in the readset by checking two conditions; whether the version-id of the current committed version in the database is different from that kept in the readset, and whether the record is being locked by another transaction. If either condition is true, T is aborted. 4) if validation succeeds, it applies T's writes to the database along with their version-ids, and releases the locks.

Our validation algorithm is identical to that of Silo [57] except for two additional mechanisms which are crucial for correctness. First, we use a unique version-id for committed as well as uncommitted versions, because the latter may be read from the access list. Second, we add the additional first step of waiting for T's dependent transactions to finish committing.

We provide a brief correctness argument here. A more detailed proof is in the Appendix of the extended version [59]. We argue the correctness of Polyjuice by reduction to Silo: if Polyjuice commits a transaction, then Silo would also commit it. According to step-1, Polyjuice ensures that if a transaction T is committed successfully, then before T starts the validation, all of its dependent transactions (e.g. T_{dep}) have been committed. This allows us to prove that executing T is equivalent to executing another hypothetical transaction T' which starts execution after all transactions in T_{dep} commit, reads/writes the same data as T, and starts validation at the same time as when T starts its validation. Therefore, if T passes the validation in Polyjuice, T' can pass the validation of Silo and successfully commit itself.

4.5 Learning backoff time

Separate from the CC algorithm, it is also important for performance to use an appropriate backoff time for retrying an aborted transaction. Existing systems, e.g. Silo, use simple binary exponential backoff which doubles the backoff time with each failed attempt. This simple strategy is inadequate as it often results in backoff times that are too short in the first couple of retries but too large after several successive retries. Furthermore, this strategy does not distinguish between different transaction types when adjusting backoff times. This is suboptimal: intuitively, one can increase the backoff time more aggressively for a transaction type more prone to contention.

For learning the backoff time, Polyjuice uses a separate backoff policy table. The rows (a.k.a. state space) of this table enumerate 3 dimensions: 1) the transaction type 2) the status of the current execution (commit or abort). 3) the number of aborted attempts prior to the current execution with a fixed cutoff: our current implementation uses 0, 1 or 2 to indicate whether there has been 0, 1 or 2+ aborts so far. The action space of the backoff policy table is inspired by recent work on learnable congestion control in networking [22]. Specifically, a worker adjusts the backoff time for each transaction type multiplicatively whenever it commits/aborts a transaction:

$$backoff = \begin{cases} backoff \times (1 + \alpha_{t,i,aborted}), & abort \\ backoff / (1 + \alpha_{t,i,committed}), & commit \end{cases}$$

In the above equations, $\alpha_{t,i,committed}$ or $\alpha_{t,i,aborted}$ is the learned parameter (a.k.a. action) in the policy table for transaction type t, number of prior aborted attempts i and execution status $committed$ or $aborted$. To enable easier training, we use bounded discrete values for α. In particular, α can be zero, resulting in unchanged backoff time.

5 Training Policies

Overview The policy space discussed in §4 is exponentially large: there are a^s different policies, where s is the number of different states and a is the number of different actions per state. The goal of training is to efficiently search for a good policy for a given workload.

Polyjuice performs training offline. During regular execution, Polyjuice logs executed transactions together with their inputs. Using a separate training machine, Polyjuice emulates the target workload by reissuing transactions with their logged inputs. We measure a policy's commit throughput under the emulated workload.

Polyjuice uses Evolutionary Algorithm (EA) for training. We have also explored the policy-gradient method from the RL literature (§5.2). Despite EA's simplicity, we have found it to be more efficient than the alternative (§7.5).

5.1 Training using Evolutionary Algorithm

EA is an optimization approach to search for a solution with good fitness by evolving a population of individuals via nature-inspired mechanisms such as crossover, mutation, and selection [10, 16, 20]. In Polyjuice, the fitness of an individual (aka a candidate policy) corresponds to the policy's commit throughput under the given workload.

EA starts by initializing the first generation of the population. The size of the population for each iteration is a configurable hyperparameter. To create a new children generation, EA performs mutation on the policies (including CC and backoff policies) of the current generation (parents). It then evaluates the "fitness" of each mutated child by measuring its throughput. Finally, EA selects N individuals according to their fitness to survive to the next generation.

Mutation. EA mutates each cell of a parent's CC and backoff policy table independently with probability p. If the cell corresponds to a binary choice such as read-version or write-visibility, the mutation flips the choice. If the cell corresponds to an integer choice (e.g. any of the wait actions), the mutation varies the integer value by some distance uniformly sampled from the interval $[-\lambda, \lambda]$. The mutated integer is clipped to always lie within the valid range. The initial values of mutation probability (p) and mutation interval (λ) are configurable hyperparameters. We decrease p and λ gradually as the training progresses to facilitate convergence. This is akin to the decrease in learning rate in gradient descent methods or the gradual reduction of temperature in simulated annealing.

Crossover, another popular EA mechanism, is not effective in our context. Crossover endows a child's policy with some rows from one parent and some rows from the other parent. Unfortunately, such a child is likely to perform worse than either of its parents. This is because, in most good policies, the wait actions of different rows are not independent but highly correlated. Thus, mixing the rows of different policies often results in worse performance.

At the end of each iteration, EA chooses N individuals with the best performance from the current population to survive to the next iteration. In our experiments, this simple selection mechanism trains faster than tournament selection [10, 16, 20].

Warm start. Instead of using all random policies, we seed the initial population with several known good policies, including OCC, 2PL*, and Callas RP/IC3. These policies are likely not optimal for the given workload, but they provide some good initial policies to give EA a "warm start" in training.

5.2 Alternative training method

Some recent works have used RL training methods to solve systems problems such as task scheduling [36], adaptive video streaming [35], multi-GPU dataflow systems [39, 40], congestion control [22], etc. We have experimented with the policy-gradient method for training a parameterized stochastic policy [62]. More concretely, we parameterize the policy table by representing each table cell using one or a set of parameters to denote the probability distribution of the action values. Suppose the cell at coordinate i, j corresponds to some action with M possible choices, we use M parameters, $p^0_{i,j}, p^1_{i,j}, \ldots p^{M-1}_{i,j}$, which are fed into a softmax function to denote the probability distribution of M choices.

For training, each iteration samples a batch of policies according to the probability distribution specified by the current table parameters. We measure the throughput of each sampled policy and use it as the "reward" in RL. Policy gradient maximizes the expected reward by performing gradient descent [62]. Our way of applying policy gradient is inspired by [3]. We compare RL- and EA-based training in §7.5.

5.3 Training for real-world deployments

Since Polyjuice relies on offline training to optimize its policy for a specific workload, this raises the question of how to use it in the real-world with changing workloads. We acknowledge that Polyjuice is not suitable for very dynamic and unpredictable workloads. However, we observe that many real-world workloads are fairly *predictable* on a day-to-day basis after analyzing the trace of an e-commerce website. This has motivated us to suggest the following deployment strategy for Polyjuice.

Optimize for the peak workload. Real-world systems are provisioned for the anticipated peak workload. Hence, our goal is to use Polyjuice to improve commit throughput during the peak time, in which the server receives the most requests in a day. There is no need to optimize for non-peak workloads because an under-utilized database is not a bottleneck for application performance. Therefore, we only need to train the policy tailored to the peak workload, and run the same policy during non-peak times as well.

Predict and retrain. Our analysis of the real-world trace shows that one can predict tomorrow's peak workload characteristics using the statistics gathered from today's peak workload (§7.6). Given this observation, one can collect the trace of the peak hour today, retrain the policy based on the trace, and run this policy for tomorrow. Doing so naively requires Polyjuice to retrain the policy every day. We can defer retraining if the predicted peak workload does not differ significantly from the one targeted by the current policy. Our analysis of the real-world trace shows that the peak workload can remain stable for many days after a significant change. Hence, deferral can greatly decrease the number of retraining times. One is right to be concerned that deferred training and prediction errors can result in running a policy optimized for a different workload than the actual one happening. We also evaluate the effect of this discrepancy in §7.6.

6 Implementation

We implemented Polyjuice in C++ using the codebase of Silo [57] by replacing Silo's concurrency control mechanism with Polyjuice's policy-based algorithm. We implemented Polyjuice's offline training separately in Python (and RL-based training in TensorFlow). The result of training is the policy table, which is written to disk as a file and later loaded into memory by the C++ database. Each worker thread in the C++ database maintains a pointer to the in-memory policy table. When switching the policy, we reset the policy pointer in each worker thread. Polyjuice doesn't need to atomically switch the policy pointers of all threads. This is because Polyjuice's validation procedure can ensure correctness regardless of the policies used during execution.

Like Silo, transaction logic is written in C++ using a few API calls (e.g. Get/Put/CommitTx). Each Get/Put/CommitTx API call's access-id is its corresponding sort order based on the API invocation's line number. For range queries, our current prototype reuses Silo's existing mechanism which always reads the committed value.

The pseudocode of how Polyjuice executes a transaction according to the policy is included in the Appendix of the extended version [59].

7 Evaluation

7.1 Experimental setup

Hardware. Our experiments are conducted on a 56-core Intel machine with 2 NUMA nodes. Each NUMA node has 28 cores (Xeon Gold 6238R 2.20GHz) and 188GB memory.

Workloads. We use three benchmarks, TPC-C [5], TPC-E [6], and a micro-benchmark with ten types of transactions. In our experiments, each worker retries an aborted transaction indefinitely until success, to ensure that committed transactions adhere to the workload's specified mix ratio of different transaction types. If we had not done this and let a worker give up an aborted transaction and start a new one with a different type, we would incorrectly learn a policy that intentionally aborts some transaction types to maximize aggregate throughput.

Baselines for comparison . We compare Polyjuice with five existing algorithms: OCC (Silo) [57], 2PL [17], IC3 [61], Tebaldi [53] and CormCC [55]. For Silo and IC3, we use the authors' source code. For Tebaldi and CormCC, we simulate them in our codebase to provide an apples to apples comparison. For 2PL, we implement it in Silo's codebase with an optimized WAIT-DIE mechanism. The optimization avoids aborts if locks are acquired following a global order, as is the case with our TPC-C and microbenchmark.

Methodology. For the training, we use 300 iterations by default. After each iteration, we pick 8 policies from the current population. For each of them, we generate another 4 children policies and add them to the selection pool. Therefore, there are a total of $8 * 5 = 40$ policies at each iteration. To evaluate the performance of the learned policy as well as other baseline algorithms, we run the workload five times, with each run taking 30 seconds. By default, the graphs show the median.

7.2 TPC-C

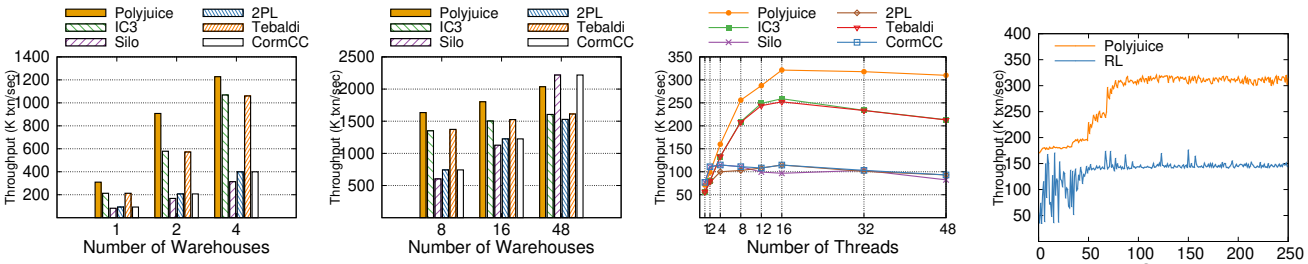
For the TPC-C benchmark, we evaluate the three read-write transactions only, as the remaining two read-only transactions can be processed with the snapshot mechanism derived from Silo. We vary the number of warehouses in the benchmark to change the level of contention.

By default, we use the 3-layer configuration for Tebaldi, which divides the read-write transactions into two groups (NewOrder, Payment vs. Delivery) isolated by 2PL [53]. Tebaldi's 2-layer configuration puts all read-write transactions into the same group, which is the same as IC3. We simulate CormCC according to its paper [55]. In particular, we partition the workload by warehouse so that all accesses to the same warehouse are protected by the same CC. Moreover, as all warehouses are inter-changeable in our benchmark, all partitions should also use the same CC protocol. Based on this observation, we measure the performance of 2PL and OCC, and pick the one with the better performance as the CC protocol for each partition.

Throughput. Fig. 4a and 4b show the throughput of various algorithms with 48 threads under different contention levels. Fig. 4a gives the throughput under high contention. Polyjuice achieves significant performance improvements. Specifically, with two warehouses, its throughput reaches 907K TPS, which is more than 1.5× of other algorithms. IC3 and Tebaldi have higher throughput than other existing algorithms because they can exploit a form of "pipelined" execution. Both have the same throughput, which differs from the original paper, as we disable their manual optimization for commutativity and uniqueness. Compared with IC3 and Tebaldi, Polyjuice achieves 56% improvement because of two factors: First, it can avoid unnecessary waiting because it uses the runtime information to infer the CC action, while IC3 only leverages the static information. Second, Polyjuice can either read dirty or clean versions of data. This flexibility enables it to achieve more efficient interleavings. We provide a detailed analysis with an example in § 7.3.

Fig. 4b shows the throughput under moderate and low contention. Polyjuice outperforms the others for 8 and 16 warehouses. For 48 warehouses, in which each worker corresponds to its local warehouse, Polyjuice is slightly slower (8%) than Silo, even though Polyjuice learns the same policy as Silo. This is because Polyjuice needs to maintain additional metadata in each tuple, which affects the cache locality.

Scalability. Fig. 4c shows the scalability of Polyjuice under high contention (1 warehouse). Polyjuice has the same scalability as IC3 and Tebaldi, which can scale to 16 threads.



(a) High Contention (b) Moderate to Low Contention (c) Scalability (1 WH)

Figure 4: TPC-C Performance and Scalability

Figure 5: EA v.s. RL

		Polyjuice	IC3	Tebaldi	Silo	2PL	CormCC
	Neworder	163/151/179/245	251/246/296/345	246/240/291/354	1084/20/62/263	450/31/49/178	450/31/49/178
Latency(μs)	Payment	163/151/181/252	247/242/291/340	242/236/285/348	6/4/9/24	658/19/97/1554	658/19/97/1554
AVG / P50 / P90 / P99	Delivery	172/167/194/269	156/152/177/223	155/151/175/208	108/101/120/248	183/145/279/621	183/145/279/621

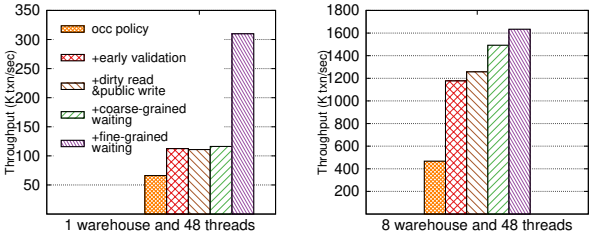
Table 2: Latency for each transaction type in TPC-C with 1 warehouse and 48 threads

Compared with them, Silo and 2PL do not scale beyond four threads because they cannot exploit parallelism under high contention. CormCC also has the scalability issue because it is limited by the protocols (2PL and OCC) it uses.

Performance of each transaction type. We also study the throughput and latency for each type of read-write transaction with 1-warehouse and 48 threads (Table 2). For Polyjuice, the throughput of each type is 132K (NewOrder), 126K (Payment) and 11K (Delivery) TPS, which follows TPC-C specified ratio (45:43:4) very closely. This is because each worker retries an aborted transaction infinitely until it succeeds before starting a new transaction. Therefore, the ratio of the per-type commit throughput is exactly the same as how each worker generates these types. For the latency of NewOrder, Polyjuice has higher P99 latency than 2PL, but lower latency than Silo, IC3 and Tebaldi. For Delivery, the outcome is flipped: Polyjuice has lower P99 latency than 2PL, but higher latency than Silo, IC3 and Tebaldi. For Payment, Polyjuice has lower P99 latency than IC3, Tebaldi and 2PL.

Factor analysis. To better understand the advantages of Polyjuice, we perform a factor analysis to examine the benefits of different actions. We start with a policy including only the actions of OCC (Table 1). Then, we gradually add other actions into the action space and measure the performance improvements. We classify the waiting actions into coarse-grained waiting and fine-grained waiting. The former means the actions of waiting for the dependent transaction to commit and learning the backoff. The latter refers to waiting for a certain access of the dependent transaction to finish.

Fig. 6a and 6b show the factor analysis result with 1 and 8 warehouses. For the 1-warehouse workload, adding "early validation" into the action space can improve the performance by 70%, because it can detect the conflicts earlier and reduce the retry cost. Polyjuice gets a performance boost after applying fine-grained waiting actions (116K to 309K TPS) due to full



(a) High Contention (b) Moderate Contention

Figure 6: Factor Analysis On TPC-C Benchmark

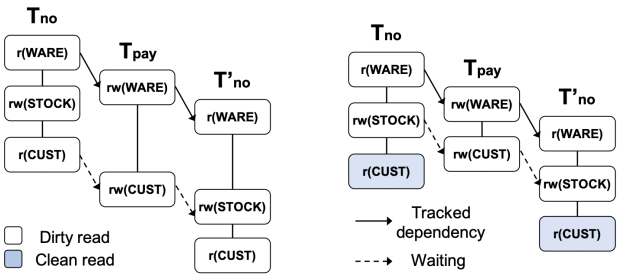
exploitation of the potential parallelism. However, each action has a different effect factor with different workloads. For the 8-warehouse workload, adding "early validation" achieves larger improvement (467K to 1177K TPS) than others.

7.3 A case study of learned policy

We analyze an example learned policy to understand how it outperforms existing CC algorithms.

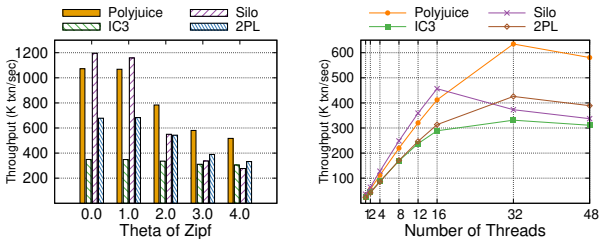
Fig. 7 shows an example of how IC3 and our learned policy mediate the data access of 3 concurrently executing transactions: T_{no} (NewOrder), T_{pay} (Payment) and T'_{no} (NewOrder). All three access the same warehouse. Fig. 7 shows a few crucial data accesses for each transaction: For NewOrder transactions (T_{no}, T'_{no}), these accesses are: read from WAREHOUSE table (r(WARE)), followed by an update to STOCK table (rw(STOCK)), and finally read from CUSTOMER table (r(CUST)). The crucial accesses of Payment (T_{pay}) are: update to WAREHOUSE (rw(WARE)) and update to CUSTOMER (rw(CUST)).

The three transactions conflict because they access the same record in WAREHOUSE. Fig. 7 shows a specific dependency pattern that can arise from their WAREHOUSE ac-



(a) IC3 interleaving. (b) Polyjuice interleaving.

Figure 7: Polyjuice's learned policy results in a more efficient interleaving for TPC-C than IC3.



(a) Throughput (b) Scalability (θ is 3.0)

Figure 8: TPC-E Performance and Scalability

cess, $T_{no,r(WARE)} \rightarrow T_{pay,rw(WARE)} \rightarrow T'_{no,r(WARE)}$ as all WAREHOUSE accesses use dirty reads. As shown in Fig. 7a, to avoid the dependency cycle, IC3 makes T_{pay}'s read of CUSTOMER wait for T_{no}'s CUSTOMER update to finish. This is because IC3 always uses dirty reads, so $T_{pay,rw(CUST)}$ must be ordered after $T_{no,r(CUST)}$ in accordance with their WAREHOUSE access' ordering. IC3 also makes T'_{no} STOCK update wait for T_{pay}'s CUSTOMER update, even though these two access different tables. This is because IC3 only tracks the immediate dependency: by waiting for T_{pay}'s CUSTOMER update, it ensures that T_{no} and T'_{no} will not form a dependency cycle even though T'_{no} is not aware of the transitively dependent T_{no}.

Fig. 7b shows the interleaving obtained by Polyjuice, which is more efficient. Unlike IC3, the learned policy makes T_{pay}'s CUSTOMER update wait for T_{no}'s STOCK access which is earlier than $T_{no,r(CUST)}$. This shorter wait works because the learned policy also makes T_{no}'s CUSTOMER read a committed version, which helps avoid the conflict between $T_{no,r(CUST)}$ and $T_{pay,rw(CUST)}$. This is in contrast to IC3, which makes $T_{no,r(CUST)}$ perform a dirty read. The learned policy still makes T_{no}'s STOCK update wait for T_{pay}'s CUSTOMER update like IC3 does, but the overall interleaving is more efficient.

Apart from IC3, neither CormCC nor Tebaldi can exploit this interleaving. CormCC does not allow dirty reads. Tebaldi uses the same action (either dirty or clean read) for all accesses

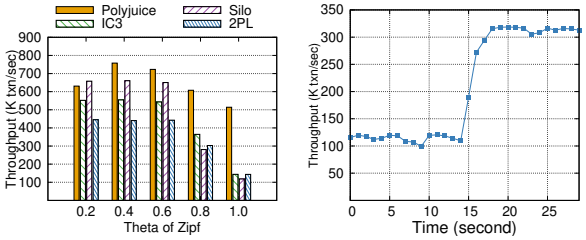


Figure 9: Micro-benchmark with 10 tx types.

Figure 10: Throughput during policy switch.

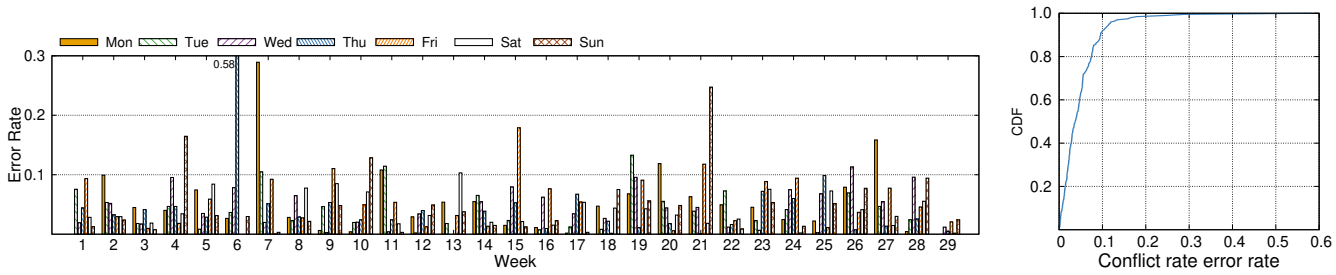
within a transaction. Fig. 7b's interleaving requires using dirty reads for NewOrder's WAREHOUSE access and clean reads for CUSTOMER access.

7.4 Bigger benchmarks

We use two bigger benchmarks to check if Polyjuice can learn a CC policy in a much larger search space. The first benchmark includes three read-write transactions from TPC-E, TRADE_ORDER, TRADE_UPDATE and MARKET_FEED. Compared with the state space of TPC-C (total 26 states), this benchmark is much more complex (total 65 states). The second benchmark is a micro-benchmark with ten types of transactions each with 8 accesses performing random updates (total 80 states). For each type of transaction, the last operation updates records in a unique table to distinguish it from other types. We build this benchmark because the action space grows exponentially with increasing transaction types.

TPC-E. We vary the contention in TPC-E by controlling the updates on SECURITY table. Specifically, all updates follow the Zipf distribution and we vary the θ of Zipf from 0.0 to 4.0 to increase the contention. We didn't evaluate Tebaldi as it doesn't provide a manual grouping strategy for TPC-E. Similarly, we didn't evaluate CormCC as it is unclear how to partition the data for TPC-E.

As shown in Fig. 8a, the throughput of Polyjuice is 42%, 49% and 55% higher than other algorithms when contention is high ($\theta = 2, 3, 4$). Unlike TPC-C, in this experiment, the improvement of Polyjuice is mainly attributed to the learned backoff. Specifically, Polyjuice learns a different backoff mechanism from Silo's design. We find out that in Silo, the frequent aborts of TRADE_ORDER result in a large backoff under high contention and the system spends a lot of time waiting before retry. In Polyjuice, for TRADE_ORDER transaction, it wouldn't increase the backoff even though the transaction is aborted. Although the abort rate remains high compared with Silo, the overall throughput is higher. Fig. 8b shows the scalability of Polyjuice under TPC-E with $\theta = 3$. Polyjuice's performance can scale to 18.5× with 48 threads over that with a single thread, which is higher than IC3 (12.3×) and 2PL (16.6×). Silo (9.4×) does not scale due to the frequent transaction aborts.



(a) Error rates for each day

(b) CDF of error rates

Figure 11: Error rates of conflict rate

Microbenchmark with 10 Types of Transactions. For this benchmark, we change the access distribution of the first operation to vary the contention level. Specifically, we change the θ of Zipf from 0.2 to 1.0 in the range of 4K. Other operations randomly update the records in the range of 10M, which results in little contention. Fig. 9 shows the result, Polyjuice's throughput is at least 66% higher than other concurrency control mechanisms under high contention scenarios. This is because the learned policy pipelines the operations on some of the high-contention records while optimizing the waits for low-contention records.

7.5 Training

We have also implemented policy-gradient based RL training for the same workload. We initialize RL with an IC3-like policy to improve its training under this high contention workload. The initialization sets the parameters corresponding to IC3 actions with a high probability (in our case, 80%). The comparison result is shown in Fig. 5 for TPC-C with 1 warehouse and 48 threads. The RL agent converges after around 100 iterations, but the throughput of the learned policy is only 178K TPS. In contrast, Polyjuice can learn a 309K TPS policy in 100 iterations. Our training runs on a single machine for now; each iteration takes 80 seconds, most of which are spent on evaluating policy performance.

7.6 Coping with real-world workloads

7.6.1 Trace analysis

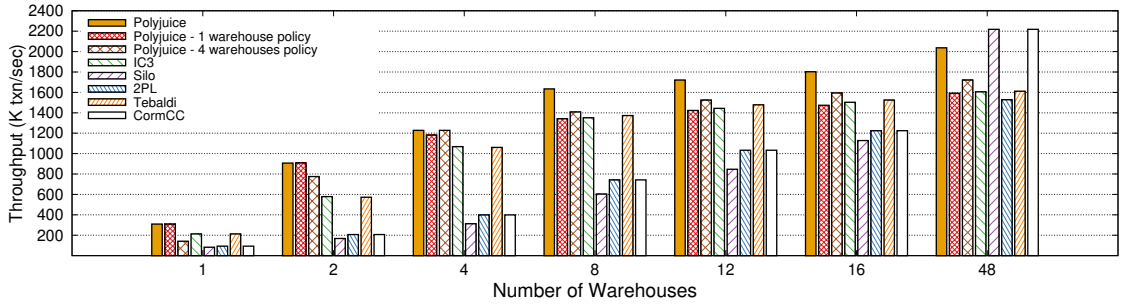
The trace. Our analysis is based on the trace of a real-world e-commerce website, downloaded from Kaggle [24]. The trace includes a log of requests sent to the web server, including the request time and several parameters. There are three types of requests: `VIEW`, for when a user views a product; `CART`, for when a user adds a product to the shopping cart; and `PURCHASE`, for when a user purchases a product. As `VIEW` corresponds to a read-only request, we only include the two types of read-write requests `CART` and `PURCHASE` in our analysis.

Workload predictability. For this analysis, we extract all the logged requests from Oct. 7th 2019 to Apr. 26th 2020 (29 weeks). After removing 6 invalid days, there are 197 days in total. We only consider the peak-hour workload for each day, since there is no need to optimize settings when the database is under-utilized and its commit throughput is limited by the incoming request rate instead of the CC performance.

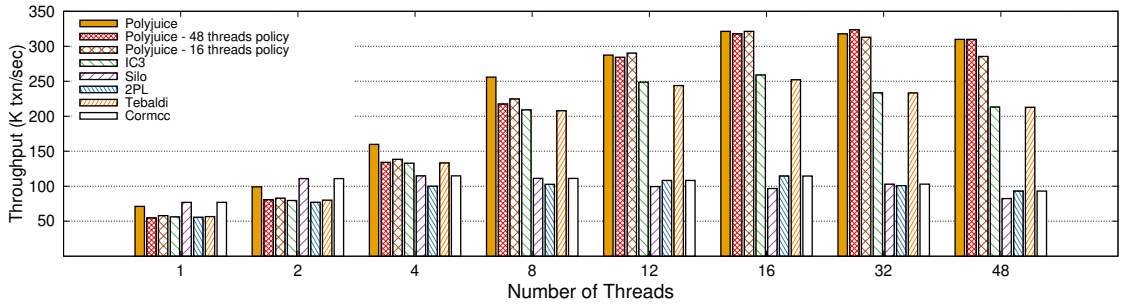
As proposed in § 5.3, we predict tomorrow's peak workload characteristic to be the same as today's peak. How accurate is such a prediction? For our analysis, we characterize a workload by its contention level, which has the most effect on the learned policy. However, since the trace does not contain information on how long each request executes, we approximate the likelihood of contention by considering two requests to be in conflict with each other if they are sent by different users but operate on the same product id during some time window. We define $conflict\_rate = conflict\_requests/total\_requests$ within n minutes. In our analysis, we set $n = 5$ and split an hour into 12 intervals. We use the mean of the 12 conflict rates to represent the contention in this hour and pick the hour with the most requests as the peak workload in a day. We note that *conflict_rate* is heavily influenced by the request rate; the bigger the request rate, the higher the measured conflict rate.

Fig. 11 shows the error when predicting tomorrow's peak hour contention level using today's peak hour statistics. The error rate is calculated as $error\_rate = abs((tomorrow - today)/today)$. The smaller the *error_rate* is, the closer the next day's peak workload contention matches that of today. Fig. 11a shows the error rate of the conflict rates for all 196 days (except for the first day), and Fig. 11b shows the CDF of the error rates distribution. We can see that, there are only 3 days when the error rate of prediction is larger than 20%. After manually checking these 3 days, we find out that they are due to a significantly higher or lower request rate, which affects the conflict rate.

We also analyze how frequently one needs to retrain. As suggested in § 5.3, we assume retraining is deferred until the predicted conflict rate differs from the one used for training the current policy by 15%. For the trace analyzed, we only need to retrain 15 times to cover a period of 196 days.



(a) TPC-C, 48 threads.



(b) TPC-C, 1 warehouse.

Figure 12: Throughput under different workloads

7.6.2 Cost of policy switching

We evaluate the cost of switching the policy in terms of: 1) how long it takes to fully switch the policy 2) whether commit throughput is affected by policy switching. The result is shown in Fig. 10. We run the TPC-C 1 warehouse workload with 48 threads, and plot the throughput for each second. At the beginning, we run the workload with the OCC policy. Starting in the 15th second, we switch the policy to the one optimized for 1 warehouse. The result shows that it takes about 3 seconds to fully switch to a new policy, and switching does not negatively impact performance. In fact, because we are switching to a better policy, the performance quickly improves during switching.

7.6.3 Running a policy trained on a different workload

We also study what happens if the workload optimized by the policy differs from the one actually being executed. For these TPC-C experiments, we use fixed learned policies and measure their performance under various workloads that are different from those used in training.

In the first set of experiments, we use two fixed policies, which are trained using 48 threads on 1 warehouse or 4 warehouses. Fig. 12a shows the performance of fixed policies as we vary the number of warehouses, compared to existing algorithms and Polyjuice when it is always trained on the correct workload. If the evaluation workload is different from the workload used for training, the fixed policies can be sub-optimal. For example, the performance of Polyjuice (1-warehouse) is 71% of Silo under 48 warehouses. However, the performance differences between fixed and optimal policies are small when the evaluation workload is not too far off from the training workload.

In the second set of experiments, we use fixed policies trained on 1 warehouse using 48 or 16 threads. Fig. 12b shows the performance of fixed policies as we vary the number of threads. The results are similar, in that a trained policy is fairly robust to training and evaluation workload mismatch.

8 Related Work

Concurrency control. We can categorize recent CC works according to their design choices. 1) Scheduling based CC: IC3 [61], Callas [66], DPR [41] and RoCoCo [42] allow ongoing transactions to expose their writes and track dependencies at runtime, then schedule the read/write operations according to the tracked dependencies. Ding et al. [11] schedules read operation after conflicting transaction's commits to avoid aborts for OCC protocol. 2) Deterministic databases: Granola [7], Deterministic CC [15,47,48,56] and Eris [32] schedule a transaction's execution according to a predetermined order. PWV [14] adds early write visibility to the deterministic CC to further improve the performance. 3) Changing the validation algorithm to avoid unnecessary aborts: TicToc [69] avoids unnecessary aborts by using logical timestamps for validation. BCC [71] changes the validation phase by detect-

ing a special pattern. 4) Partially rolling back to reduce the abort cost [64].

In addition, there are a number of works applying MVCC into their systems. Bohm [13] combines the MVCC with deterministic CC to achieve non-blocking operations. Cicada [33] uses logical timestamps with MVCC to increase the possibility of constructing safe interleavings. Obladi [8] integrates MVCC on top of ORAM to provide security along with high performance.

All above CC protocols leverage a fixed set of design choices. Compared to them, Polyjuice is able to adapt the design choices according to the characteristics of a given workload. Some work [43, 54, 73] focus on distributed databases, which must do replication in addition to concurrency control. They propose new algorithms to handle inconsistent orderings in both concurrency control and replication.

Hybrid concurrency control. There are existing works that combine multiple concurrency control mechanisms for better performance. MOCC [60] develops a specific algorithm to combine OCC and 2PL for high-contention workloads. Sundial [70] proposes a new hybrid CC algorithm based on 2PL and OCC with logical timestamps. CormCC [55] proposes a more general hybrid method by formalizing all CC into four phases. Each operation can use any CC's policy as long as all CCs perform each phase according to the same order. Tebaldi [53] groups transactions and assigns different CC protocols to each group. However, existing algorithms are either specific for combining OCC or 2PL, or need programmers to provide heuristics to choose the execution policy for each operation. Compared to them, Polyjuice is able to automatically adapt the policy for each operation according to the workload.

Learned systems. Many system optimizations can be done by machine learning models trained from historical data. In the area of databases, examples include cardinality estimation [25, 29, 45, 63], join order planning [27, 37, 44] and configuration tuning [58]. Besides databases, works have been done to improve buffer management systems [4], sorting algorithms [74], memory page prefetching [19, 72] and memory control [21], task scheduling [26], CPU scheduling [51], locking priority [12] and cache replacement [52]. Although these works try to leverage machine learning to make systems self-aware, but none of them targets on the concurrency control. Thus, they have different model design from Polyjuice.

9 Discussion

As a first attempt on learnable CC, Polyjuice has limitations, some of which we hope to address in the future.

Not suitable for rapidly changing workloads. In our experience, training takes on the order of several hundred seconds. Thus, Polyjuice is not suitable for scenarios in which workload changes quicker than every few minutes.

Inaccurate workload emulation. Training reissues executed transactions with their logged inputs. However, since transaction interleavings during training differ from that of the original execution, a transaction's outputs also differ. Polyjuice works only if such emulation inaccuracies do not significantly affect the workload access pattern.

Large state space. Polyjuice represents a^s potential policies in a table format, where s is the number of different states and a is the number of different actions per state. As the number of transactions and the number of accesses in each transaction increase in the workload, both s and a increase. The resulting much enlarged search space will make training via EA less effective. One potential solution is to follow the breakthrough of deep reinforcement learning, and use a function approximator like a deep neural network to approximate the policy table with parameters far fewer than the number of table cells. It is a well-known challenge to make deep RL work effectively.

More expressive policy space. There are several interesting directions to expand the policy space, such as supporting multi-version databases, explicit CPU scheduling of execution, fine-grained instead of binary contention levels.

Weaker and mixed isolation levels. Polyjuice currently only guarantees serializability. Some applications can work with weaker or mixed isolation levels [9, 23, 34, 38, 65]. It is an interesting extension to generalize to these scenarios.

Acknowledgements

Chien-chin Huang and Minjie Wang contributed valuable ideas in the early stage of this project. We thank the anonymous reviewers for the valuable comments. We are especially grateful to our shepherd, Deniz Altınbüken, for helping improve the paper's presentation. Jiachen Wang, Huan Wang, Zhaoguo Wang and Haibo Chen were supported by National Key Research and Development Program of China (No. 2020AAA0108500), National Natural Science Foundation of China (No. 61902242), and the HighTech Support Program from Shanghai Committee of Science and Technology (No. 20ZR1428100). Ding Ding, Conrad Christensen, and Jinyang Li were supported by NSF grant 1816717, and a gift from NVIDIA and AMD. Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) and Jinyang Li (jinyang@cs.nyu.edu) are the corresponding authors.

References

[1] Atul Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. *Ph.D. Thesis*, 1999.

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2), 1981.

[3] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hard-

ware. In *International Conference on Learning Representations (ICLR)*, 2019.

[4] Xinyun Chen. Deepbm: A deep learning-based dynamic page replacement policy.

[5] The Transaction Processing Council. TPC-C Benchmark. http://www.tpc.org/tpcc/.

[6] The Transaction Processing Council. TPC-E Benchmark. http://www.tpc.org/tpce/.

[7] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 223–235, 2012.

[8] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 727–743, 2018.

[9] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1615–1628, 2016.

[10] Lawrence Davis. Handbook of genetic algorithms. 1991.

[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.

[12] Jonathan Eastep, David Wingate, Marco D Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th international conference on Autonomic computing*, pages 215–224, 2010.

[13] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. In *VLDB*, 2014.

[14] 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, 2017.

[15] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 15–26, 2014.

[16] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2), 1988.

[17] J. N. Gray, R. A Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, 1975.

[18] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[19] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[20] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[21] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 39–50. IEEE Computer Society, 2008.

[22] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059, 2019.

[23] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: Compositional reasoning and inference for weak isolation. In *45th Symposium on Principles of Programming Languages (POPL)*, 2018.

[24] Michael Kechinov. ecommerce behavior data from multi category store. https://www.kaggle.com/mkechinov/ecommerce-behavior-data-from-multi-category-store.

[25] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[26] Tim Kraska, Mohammad Alizadeh, Alex Beutel, E Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. CIDR, 2019.

[27] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[28] H. T. Kung and John Robinson. On optimistic methods for concurrency control. In *ACM Transactions on Database Systems (TODS)*, 1981.

[29] M Seetha Lakshmi and Shaoyu Zhou. Selectivity estimation in extensible databases-a neural network approach. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 623–627. Morgan Kaufmann Publishers Inc., 1998.

[30] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[31] Jialin Li, Ellis Michael, and Dan Ports. Eris: Coordination-free consistent transactions using network multi-sequencing. In *SOSP*, 2017.

[32] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[33] 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*, pages 21–35, 2017.

[34] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[35] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *SIGCOMM*, 2017.

[36] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.

[37] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 3. ACM, 2018.

[38] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 453–468, 2017.

[39] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. 2018.

[40] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2430–2439. JMLR.org, 2017.

[41] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[42] 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)*, pages 479–494, 2014.

[43] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 517–532, 2016.

[44] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 4. ACM, 2018.

[45] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. Quicksel: Quick selectivity learning with mixture models. *arXiv preprint arXiv:1812.10568*, 2018.

[46] Dan Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. In *VLDB*, 2012.

[47] Kun Ren, Jose M Faleiro, and Daniel J Abadi. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1583–1598, 2016.

[48] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, 2019.

[49] L. Sha, J.P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE transactions on Computers*, 37(2), 1988.

[50] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3), 1995.

[51] Yangjun Sheng, Anthony Tomasic, Tieying Sheng, and Andrew Pavlo. Scheduling oltp transactions via machine learning. *arXiv preprint arXiv:1903.02990*, 2019.

[52] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 529–544, 2020.

[53] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 283–297, 2017.

[54] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[55] Dixin Tang and Aaron J Elmore. Toward coordination-free and reconfigurable mixed concurrency control. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 809–822, 2018.

[56] 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*, pages 1–12, 2012.

[57] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[58] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[59] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control, 2021.

[60] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores (extended version). *Hewlett Packard Labs Technical Report HPE-2016*, 58, 2016.

[61] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1643–1658, 2016.

[62] Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[63] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB), page to appear*, 2019.

[64] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1689–1704, 2016.

[65] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining {ACID} and {BASE} in a distributed database. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 495–509, 2014.

[66] 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*, pages 279–294, 2015.

[67] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. 2012.

[68] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *PVLDB*, 2014.

[69] 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*, pages 1629–1642, 2016.

[70] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. In *PVLDB*, 2018.

[71] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment*, 9(6):504–515, 2016.

[72] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*, pages 305–311. ACM, 2017.

[73] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[74] Hanqing Zhao and Yuehan Luo. An $o(n)$ sorting algorithm: Machine learning sorting. *arXiv preprint arXiv:1805.04272*, 2018.

A Artifact Appendix

Abstract

This artifact provides the source code of Polyjuice and scripts to run the main experiments in this paper. Polyjuice is a fast in-memory database, which is based on a new concurrency control framework and uses the evolutionary algorithm to search for the optimal concurrency control policy under a specific workload.

Scope

This artifact (including the document, source code, and the scripts) is used to run the main experiments in Polyjuice. We note that the reported performance is based on the policies learned on dedicated machines. Therefore, if you run on different hardware, the performance numbers might be different from those in the paper. The artifact aims to verify the following claims:

TPC-C/TPC-E/Microbenchmark performance. The results should show that Polyjuice outperforms other baselines under high/moderate contention. Polyjuice's performance is slightly lower than Silo under low-contention workloads (e.g. TPC-C 48 threads - 48 warehouse, TPC-E zipf 0.0 and 1.0, Microbenchmark zipf 0.2).

TPC-C/TPC-E scalability. The results should show that Polyjuice scales better than Silo and 2PL.

TPC-C factor analysis. The results should show that for the 1-warehouse workload, there is a performance boost after adding fine-grained waiting actions. For the 8-warehouse workload, adding early validation achieves large improvement.

Training. The results should show that the training using EA has better performance than RL.

Switching policy. The results should show that it takes several seconds to fully switch to a new policy, and the process of switching does not negatively impact the database's performance.

Contents

- **README:** A detailed document showing how to download, compile and run the source code of Polyjuice.
- **Source code:** We provide the source code of Polyjuice, as well as TPC-C, TPC-E and microbenchmark.
- **Scripts:** We provide the scripts to run all the main experiments in our paper.

Hosting

An open-source version of Polyjuice is available at https://github.com/derFischer/Polyjuice. We recommend using the latest commit on the `master` branch of the repository, which would be maintained by the authors.

Code license: Apache License 2.0.

Requirements

Hardware Dependencies. Most of our experiments will use 48 physical cores. Using fewer cores or hyper-threads might produce different performance results.

Software Dependencies. Our project depends on libraries as listed and we give their installation commands on Ubuntu 18.04 with apt-get. On our machine, GCC 7.5.0/8.3.0. We recommend using the same version of Python and GCC as ours because otherwise it may fail to compile.

Library	Install Command
libnuma	apt-get install libnuma-dev
libdb	apt-get install libdb-dev libdb++-dev
libaio	apt-get install libaio-dev
libz	apt-get install libz-dev
libssl	apt-get install libssl-dev
autoconf	apt-get install autoconf
libjemalloc	apt-get install libjemalloc-dev

Training Dependencies. Our training code is based on TensorFlow 1.14.0 and we use Python 3.6.9 on our machine. Using TensorFlow 2.0 may fail to run the training code since some APIs in version 2.0 are different from those in 1.0.

AE Methodology

Sumission, review and badging methodology: https://www.usenix.org/conference/osdi21/call-for-artifacts.

Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing

Sijie Shen, Rong Chen, Haibo Chen, Binyu Zang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai Artificial Intelligence Laboratory
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Many application domains can benefit from hybrid transaction/analytical processing (HTAP) by executing queries on real-time datasets produced by concurrent transactions. However, with the increasingly speedy transactions and queries thanks to large memory and fast interconnect, commodity HTAP systems have to make a tradeoff between data freshness and performance degradation. Fortunately, we observe that the backups for high availability in modern distributed OLTP systems can be retrofitted to bridge the analytical queries and transactions in HTAP workloads. In this paper, we present VEGITO, a distributed in-memory HTAP system that embraces freshness and performance with the following three techniques: (1) a lightweight gossip-style scheme to apply logs on backups consistently; (2) a block-based design for multi-version columnar backups; (3) a two-phase concurrent updating mechanism for the tree-based index of backups. They collectively make the backup fresh, columnar, and fault-tolerant, even facing millions of concurrent transactions per second. Evaluations show that VEGITO can perform 1.9 million TPC-C NEWORDER transactions and 24 TPC-H-equivalent queries per second simultaneously, which retain the excellent performance of specialized OLTP and OLAP counterparts (e.g., DrTM+H and MonetDB). These results outperform state-of-the-art HTAP systems by several orders of magnitude on transactional performance, while just incurring little performance slowdown (5% over pure OLTP workloads) and still enjoying data freshness for analytical queries (less than 20 ms of maximum delay) in the failure-free case. Further, VEGITO can recover from cascading machine failures by using the columnar backup in less than 60 ms.

1 Introduction

For more than four decades, online transaction processing (OLTP) and online analytical processing (OLAP) are two separate pillars in the database community, with their own design targets and specific fields. Nowadays, many application domains are highly demanding the combination of OLTP and OLAP, such as fraud detection [24, 77, 78], business intelligence [54, 80, 85], healthcare [27, 93], personalized recommendation [117], and IoT [16]. The fundamental reason behind this trend is that much information is most valuable when it first appears, but the value diminishes over time [10, 115]. For example, on 2018 Alibaba's Double 11 Online Shopping Festival (similar to Black Friday Day in the US), the peak throughput reaches 6,000,000 transactions per second, and Alibaba's real-time monitoring system behind it expects to provide a time delay of 20 ms [12]. Meanwhile, users may search for the hottest items and receive personalized advertisements [117]. Vendors also need to detect and prevent online transaction fraud [24] and rely on immediate information to adjust price and stock timely [63].

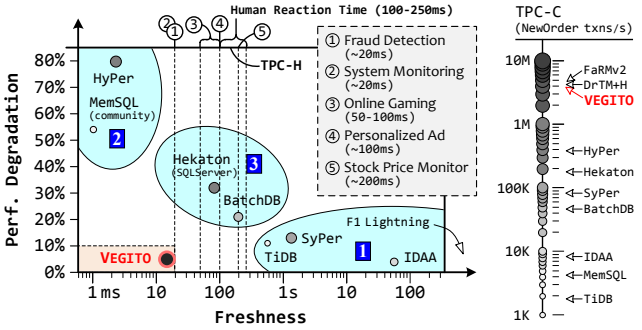


Fig. 1. *The performance-freshness tradeoff for existing HTAP systems with three different architectures and the OLTP performance of existing HTAP systems for TPC-C [95] or CH-benCHmark [29]. VEGITO is located in the desired area of HTAP systems and can offer comparable performance with modern distributed in-memory OLTP systems [46, 87, 107]. **Sources**: published results of HTAP systems [33, 34, 43, 54, 60, 62, 65, 72, 112] and real-time requirements of various application domains [12, 45, 63, 76, 77].*

In response, many recent academic and industrial efforts have been devoted to developing hybrid transaction/analytical processing (HTAP) systems [4, 14, 31, 43, 48, 54, 55, 60, 62, 80, 84, 88, 112], which are expected to support *real-time operational analytics* by breaking the walls between OLTP and OLAP systems. More specifically, analytical queries should be executed on real-time datasets quickly updated by transactions simultaneously. This implies two overarching goals for HTAP systems [43, 60, 101, 112]. **Freshness**: the maximum time delay between the tuple's value written by transactions and read by analytical queries should be near real-time (e.g., tens of milliseconds) in the failure-free case. **Performance**: the transactions and analytical queries should be executed concurrently with little performance degradation (e.g., less than 10%), compared to specialized systems.

Several architectures were proposed (see Fig. 2) but can hardly satisfy both goals of HTAP systems simultaneously, as depicted in Fig. 1. Alternative 1 (DUAL-SYSTEM) connects two OLTP and OLAP-specific systems and performs both workloads at (almost) full speed. However, the cross-system data transfer will cause a large delay (seconds to

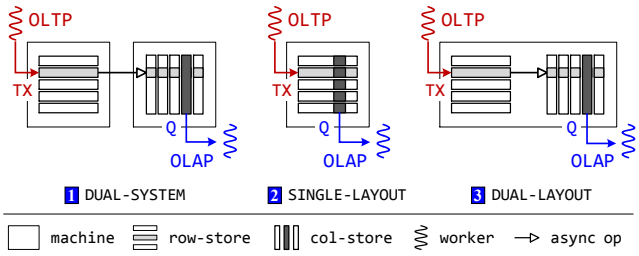


Fig. 2. *A comparison of three existing HTAP architectures.*

minutes) [72, 112].[1] It also doubles the memory cost and incurs additional CPU and network overhead. Alternative 2 (SINGLE-LAYOUT) directly builds an HTAP system derived from one specialized system (e.g., OLTP), which uses a single layout (e.g., row store) for both transactions and analytical queries to ensure data freshness. Thus, it will certainly prefer one type of workload while sacrificing the performance significantly in another (e.g., more than 50% performance degradation [43, 60]). Alternative 3 (DUAL-LAYOUT) carefully combines two different modules into a single system with intra-system data transfer, which ameliorates this problem with a performance-freshness tradeoff. However, the fundamental issues remain (i.e., noticeable performance degradation and time delay).

This paper presents VEGITO, as highlighted in Fig. 1, a distributed hybrid transaction/analytical processing system that retrofits the high availability mechanism (e.g., primary-backup replication) to support HTAP workloads. Specifically, VEGITO executes transactions and analytical queries on primary and backup replicas separately; the transaction updates are always replicated *synchronously* to *multi-version columnar* backups and *tree-based* indexes. Unlike many prior HTAP systems [43, 54, 101], VEGITO keeps both primary and backup replicas in main memory and adopts a symmetric model—each machine both runs HTAP workloads and stores data. However, modern distributed in-memory OLTP systems [34, 74, 83, 89, 107, 113] can provide extremely high throughput (millions of transactions per second) never encountered and targeted by existing HTAP systems (see Fig. 1); it poses new challenges to key components in VEGITO, causing severe performance degradation of both OLTP and OLAP workloads (see §3.2).

To remedy this, we first introduce a classic concept (epoch) into a new context (HTAP) and further propose three new techniques. First, VEGITO introduces a lightweight *gossip-style* scheme to allocate consistent epoch numbers for dependent (distributed) transactions. It allows all logs on a single backup from different transactions to be drained in *parallel*. Meanwhile, to refrain from strict synchronization among different backups, VEGITO demands the analytical query to use the latest stable epoch for reading multiple backups *consistently*. Second, VEGITO chooses a *block-based* design to build multi-version columnar backups for analytical queries, instead of conventional wisdom (chain-based design). Since the transaction updates are applied in rows while the tuples are stored in columns, VEGITO further proposes two optimizations—*row-split* and *column-merge*—to exploit both spatial and temporal locality for writing a columnar backup in rows. Third, VEGITO introduces a *two-phase* concurrent updating mechanism for the tree-based index (e.g., B^+-tree) of backups, which is essential to achieve high performance for analytical queries. VEGITO splits the insert operations within an epoch into two phases (i.e., location and update) and parallelizes two phases with two different approaches (i.e., task and data parallelism), respectively.

In addition, while VEGITO uses columnar backups to support OLAP workloads, it still preserves the same availability guarantees for *free*; namely, the backup is still *fault-tolerant*. VEGITO retrofits the replication protocol carefully and restricts changes to the *data layout* of the backup. Thus, it retains the capability of failure recovery. Besides, the original recovery protocol could be used as usual in most cases.

We implemented VEGITO by extending DrTM+H [107], a state-of-the-art distributed in-memory OLTP system. The extensions include retrofitting fault-tolerant backups to run analytical queries and integrating an efficient distributed in-memory OLAP engine, similar to MonetDB [6]. To demonstrate the efficacy of VEGITO, we have conducted a set of evaluations using several micro-benchmarks and a typical HTAP benchmark, CH-benCHmark [29], which combines TPC-C [95] and TPC-H [96] to form a complex mixed workload. For OLTP-only workloads, VEGITO (with 3-way replication) can commit 3.7 million NEWORDER transactions per second when running the TPC-C transaction mix on 16 machines. For OLAP-only workloads, VEGITO can run TPC-H-equivalent queries in 216 ms on average (geometric mean) using a single thread. These results are comparable to the excellent performance of specialized counterparts (e.g., DrTM+H [107] and MonetDB [6]). For HTAP workloads (i.e., CH-benCHmark), VEGITO can achieve a peak throughput of 1.9 million NEWORDER transactions per second and 24 TPC-H-equivalent queries per second simultaneously on a cluster of 16 machines with up to a 1.2 TB dataset. It outperforms state-of-the-art HTAP systems with three different architectures (i.e., MemSQL [4], TiDB [8], and SQL Server [54]) by several orders of magnitude on transactional performance (from 2,911X to 53,138X).[2] Meanwhile, different from prior systems, which have severe performance degradation and poor data freshness (e.g., more than 70% OLTP performance degradation in MemSQL and an 1,500-millisecond delay in TiDB), VEGITO limits the adverse effects to less than 5% and 20 ms simultaneously in the failure-free case. Further, VEGITO can recover from cascading machine failures by using the columnar backup in less than

[1] The data transfer relies on general ETL (Extract, Transform, and Load) tools [2] or specialized techniques (e.g., log shipping [65, 103]).

[2] Note that MemSQL is an in-memory HTAP system, while TiDB and SQL Server are on-disk HTAP systems.

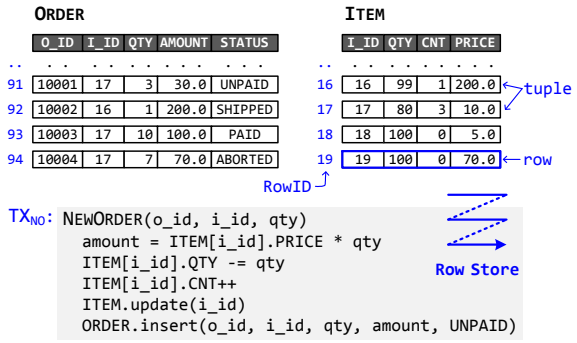


Fig. 3. *A simplified dataset on a row store and a sample transaction (TX_{no}) in TPC-C.*

60 ms. On the other hand, VEGITO can provide comparable failure-free freshness with Amazon Aurora [101].[3] Yet, it also supports fault tolerance and a much higher rate of transactions (e.g., 1.9 million vs. 1,232 NEWORDER transactions per second) by scaling out and processing transactions in the main memory.

In summary, the contributions of this paper are:

- A new distributed in-memory HTAP architecture that retrofits fault-tolerant backups to support hybrid transaction/analytical processing (§3) without compromising high availability (§5).
- Three key techniques with epoch scheme to collectively make a fresh, multi-version columnar backup with tree-based indexes for analytical queries (§4).
- A set of evaluations that confirm the efficacy of VEGITO for HTAP workloads even facing millions of transactions per second (§6).

2 BACKGROUND

Online Transaction Processing (OLTP). The workloads for OLTP systems (e.g., database) usually contain repetitive, short-lived transactions to retrieve and modify tuples (e.g., create/read/update/delete) with ACID guarantees, which are the basis of many applications such as stock exchange, e-commerce, and online order processing. Fig. 3 illustrates a simplified dataset and NEWORDER transaction in TPC-C [95], a popular OLTP benchmark. The sample transaction (TX_{no}) adds a new order (o_id) for selling qty items (i_id), which will append a tuple to ORDER table and update ITEM table. OLTP systems use the row store to exploit data locality and access patterns in transactions, where all attributes of a single tuple are stored continuously.

Moreover, modern in-memory OLTP systems [26, 30, 34, 46, 57, 113] are becoming mainstream, which scale out by sharding a large volume of data across multiple shared-nothing machines and supporting distributed in-memory

[3] Amazon Aurora [101] reports that each read replica typically lags behind the writer by a short interval (20 ms or less). We interprete it as the failure-free freshness. For freshness with failures, we have neither the number for Aurora nor for VEGITO.

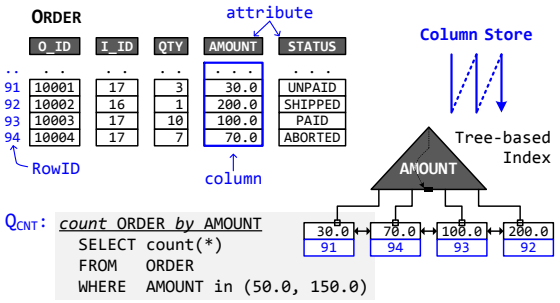


Fig. 4. *A simplified dataset on a column store with a tree-based index and a sample analytical query (Q_{cnt}) in TPC-H.*

transaction processing with high throughput and low latency. They usually rely on replication schemes (e.g., primary-backup replication [52] or Paxos state machine replication [30]) to provide high availability even with failures.

Online Analytical Processing (OLAP). By contrast, the workloads for OLAP systems (e.g., data warehouse [32, 40]) usually contain analytical queries to consistently read several attributes of massive tuples (e.g., select/join/filter/aggregate), which also are the basis of many other applications such as business intelligence, financial reporting, and data mining. Fig. 4 illustrates a simplified dataset with a tree-based index, and a sample analytical query in TPC-H [96], a popular OLAP benchmark. The sample query (Q_{cnt}) counts the number of orders (count(*)) with a given range of amounts (from 50.0 to 150.0), which needs to scan ORDER table and the index for AMOUNT attribute. Thus, the column store (aka columnar store) with tree-based indexes is widely used to exploit data locality and access patterns in analytical queries.

Hybrid Transaction/Analytical Processing (HTAP). OLTP and OLAP systems have their own design targets and specific fields, yet many application domains are highly demanding the combination of them; *analytical queries should be executed on real-time datasets quickly updated by transactions simultaneously*. For example, massive new orders are submitted by users (TX_{no} in Fig. 3). Meanwhile, the seller may want to see the number of orders with a given range of amounts in real-time (Q_{cnt} in Fig. 4). The HTAP system should meet the following two goals—freshness and performance—also appearing in recent literature [49, 60, 70, 75, 82, 100, 103].

Freshness. The maximum time delay between the tuple's value written by transactions and read by analytical queries should be near real-time (e.g., tens of milliseconds).

Performance. The transaction and analytical workloads should be executed concurrently with little performance degradation (e.g., <10%) compared to specialized systems.

Nowadays, several HTAP architectures are proposed but could hardly meet two goals simultaneously in the failure-free case—for example, a maximum delay of 20 ms and 10% performance degradation, as depicted in Fig. 1.

3 Approach and Challenges

Opportunity: fault-tolerant backup. It is important and imperative for distributed transaction processing (OLTP) systems to provide high availability (HA), which is guaranteed by replicating tuples on remote machines before committing a transaction. A common approach is to use vertical Paxos [52] with primary-backup replication [4, 7, 26, 34, 46, 59, 69, 97, 108][4], where each shard is commonly configured to use 3-way replication (one primary and two backups). The transaction will *synchronously* ship updates (i.e., logs) to all backups before committing on the primary.

We observe that the *consistent* and *fresh* backup in high availability (HA) provides the foundation for hybrid transaction/analytical processing (HTAP)—running OLTP and OLAP workloads on primary and backup replicas separately. Specifically, for **freshness**, high availability guarantees strong consistency between primary and backups by using synchronous log shipping. Thus, analytical queries can always see the latest updates of transactions. For **performance**, running different workloads on different replicas can avoid interference naturally and deploy optimizations individually (e.g., row store and column store). Moreover, reusing fault-tolerant backups and synchronous log shipping for **free**—there is no compromise on availability (§5)—can avoid extra memory for read replicas and CPU for data synchronization to support real-time OLAP.

3.1 Our approach

VEGITO is a distributed in-memory hybrid transaction/analytical processing system, which targets concurrent OLTP and OLAP workloads over one large volume of data. It scales out by partitioning data into many shards spreading across multiple shared-nothing machines while allowing both transactions and analytical queries to span any number of machines. VEGITO can provide serializability for both transactions and analytical queries. We build VEGITO out of two independent components: execution layer and memory store. An overview of VEGITO's architecture is shown in Fig. 5, which also illustrates the execution of transaction and analytical workloads.

The execution layer employs a worker-thread model by running n worker threads atop n cores playing different roles; each worker thread executes a transaction (e.g., TX_{no}) or a query (e.g., Q_{cnt}) at a time, according to its role (TP or AP worker thread). Following prior modern transaction processing systems [26, 34, 46], VEGITO also leverages 3-way primary-backup replication for high availability. To reduce the overhead of replication, a non-volatile write-ahead log (WAL) is used to buffer the updates (logs) for each backup. Transactions (synchronously) append updates to the WAL of backups involved before committing on the primary, and

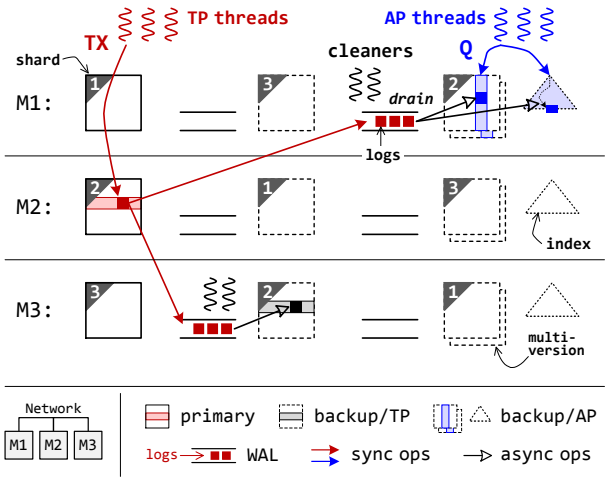


Fig. 5. *An overview of* VEGITO *with three machines and three shards. The transaction (TX) updates an attribute of a tuple in the 2nd shard (M2), and the query (Q) scans the attributes of all tuples.*

then auxiliary (cleaner) threads drain the logs in a lazy and batched manner (asynchronously). VEGITO runs TP and AP worker threads over the primary and one of the backup replicas (aka backup/AP), respectively.[5]

The memory store adopts a general key/value store over a distributed hash table to support a partitioned global address space. Each machine (e.g., M1) stores several primary and backup replicas of different shards. The primary and backup/TP use row stores, while the backup/AP uses a multi-version column store. Specifically, each key-value pair stores an attribute of a tuple. All of the attributes of a single tuple are stored continuously in row stores, while the same attributes of all tuples are stored continuously in column stores. The memory store provides a general key-value store interface (e.g., get and put) and a specific row/column store interface (e.g., row and column) to the above execution layer. Further, tree-based indexes are also maintained with backup/AP for range scans in analytical queries.

Each client contains a client library that parses and ships transactions (TX) and analytical queries (Q) to TP and AP worker threads, respectively. As shown in Fig. 5, the transaction will be executed on the primary using a concurrency control protocol (e.g., two-phase locking [18] or optimistic concurrency control [51]). Before committing the transaction on the primary, all updates are first appended to the write-ahead log (WAL) queue at each machine with a backup. The logs will be applied to backup replicas asynchronously by cleaner threads. On the other hand, the analytical query will be executed on the columnar backup replicas (backup/AP).

Further, the architecture of VEGITO can integrate existing OLTP and OLAP systems instead of writing code from scratch, including transaction/analytical engine, key-value store, data replication and recovery support [26].

[4] Our work is also applicable to other replication-based HA mechanisms, like chain replication [99] and state machine replication [30, 43].

[5] This paper uses backup/TP to denote the vanilla backup that provides high availability of transaction processing, and uses backup/AP to denote the columnar backup that also supports analytical processing.

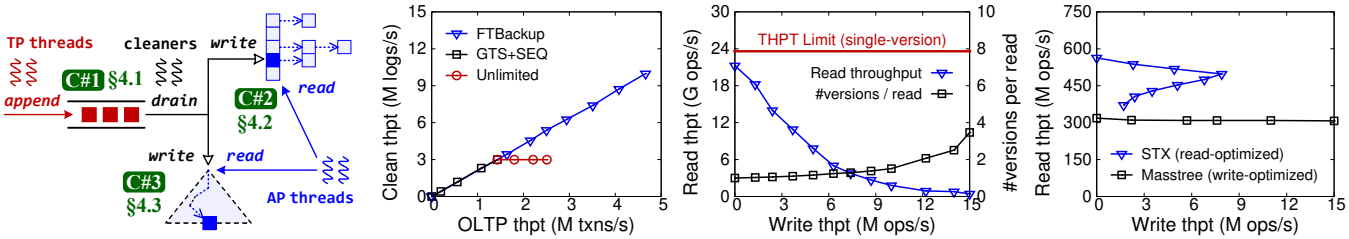


Fig. 6. *(a) A diagram of challenges in* VEGITO. *A performance analysis of three key components in our HTAP architecture, including (b) WAL queue, (c) multi-version column store, and (d) tree-based index. The open-loop clients will send as many transactions and analytical queries as possible until the throughput of some component saturated. Note that each transaction will produce two logs per shard (3-way replication).* ***Workload****: all transactions in TPC-C [95] as OLTP workloads and two simplified analytical queries (similar to Q02 and Q01 in TPC-H [96]) as OLAP workloads for (c) and (d) respectively.* ***Testbed****: A cluster of 16 machines; each machine hosts 8 TP, 10 AP, 4 log-cleaner, and 2 client threads (see §6.1).*

3.2 Challenges

We note that recent HTAP systems also propose to run analytical queries on a separate, read-optimized snapshot of transactional data [43, 60, 92, 112]. However, none of them could meet two goals simultaneously—freshness (a maximum delay of 20 ms) and performance (10% performance degradation)—even under much lower OLTP throughput (e.g., several thousand transactions per second [30, 43, 101]). Differently, our approach reuses *synchronous* log shipping to keep backups consistent and fresh; however, it indeed raises new challenges for minimizing performance degradation on transaction and analytical processing, especially when facing millions of transactions per second.

C#1: consistent and parallel log cleaning. To avoid blocking transaction committing, the updates are appended to WAL queues synchronously and then applied to the backup asynchronously by cleaner threads in parallel (see the left part of Fig. 6(a)). This design is enough and efficient to maintain a fault-tolerate backup [26, 34, 46]. In Fig. 6(b), OLTP throughput (*FTBackup*) can reach about 4.7 million transactions per second, and WAL queues are never full. However, OLAP workloads demand consistent backups. It means that the cleaner threads should drain logs following the dependency in transactions. A common solution is to record a global timestamp in each log and drain logs in sequence [55, 65, 103, 112]. This causes a significant loss (70%) in throughput (*GTS+SEQ*), dropping to 1.4 M txns/s of OLTP throughput. Given WAL queues with unlimited memory (*Unlimited*), we further decouple the performance bottlenecks of transaction processing and log cleaning. Performance degradation can happen for two reasons. First, the OLTP throughput is limited to 2.5 M txns/s due to assigning global timestamps for every transaction in a cluster of 16 machines, causing high contention [106]. Second, draining logs sequentially limits the clean throughput to 3.0 M logs/s and further limits the OLTP throughput to 1.4 M txns/s, since all transactions would be blocked when WAL queues are full. Therefore, VEGITO needs a new approach to draining logs at each machine in a *consistent* and *parallel* way.

C#2: multi-version column store building. The backup for analytical processing (backup/AP) should store tuples in a columnar format to achieve high performance. Meanwhile, cleaner threads and AP threads will write and read the same backup simultaneously, especially with different flavors of locality (row-wise writes vs. column-wise reads). Multi-version concurrency control (MVCC) [19, 110] is commonly used to resolve conflicts between read and write operations by maintaining multiple snapshots. The chain-based design (see the top right corner of Fig. 6(a)) is widely used by multi-version (row) stores [33, 39, 50, 58, 60, 110], and prior HTAP systems [20, 68, 84] also follow this design. However, column-wise reads with a given version (snapshot) have to access pointer-linked tuples (chains) and frequently check their versions, causing massive cache misses and severe performance degradation for analytical queries [58]. As shown in Fig. 6(c), the read throughput drops more than 90% (from 21.2G to 1.8G ops/s) with growing write throughput (10M ops/s), even just accessing 0.5 more versions per read on average. Note that the query (similar to Q02 in TPC-H [96]) simply reads one column updated by TPC-C transactions (like QTY attribute of ITEM table in Fig. 4), and the median latency is about 180 ms over a single-version store, close to the average latency of queries in CH-benCHmark [29]. Therefore, VEGITO demands a new approach to building a multi-version column store that can preserve the locality of both row-wise writes (log cleaners) and column-wise reads (analytical queries).

C#3: concurrent tree-based index updating. The tree-based index (e.g., B^+-tree) is imperative to support range scans for analytical queries. In HTAP systems, the index has to serve both writes (cleaner threads) and reads (AP threads) simultaneously (see the bottom right corner of Fig. 6(a)). Although several research efforts have been devoted to building concurrent tree-based data structures [17, 61, 86, 94, 102, 104, 114], to our knowledge, none of them satisfies our requirements and exploits HTAP workload characteristics. Fig. 6(d) shows the throughput of range scans with growing write throughput for read-optimized and write-optimized tree-based index structures (i.e., STX [21] and

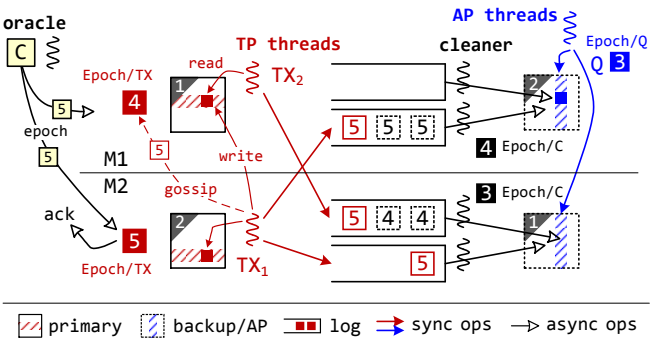


Fig. 7. *An example of lightweight epoch assignment and parallel log cleaning for two machines and two shards.*

Masstree [61]). STX can outperform Masstree by 1.8× for read-only workloads (564 M vs. 319 M ops/s). However, its write throughput is limited to 7.9 M ops/s due to high contention on concurrent writes and collapses when the clients send more write requests. In addition, the read throughput drops up to 34% because of the interference from heavy updates. On the contrary, Masstree is highly optimized to handle fast concurrent writes at the expense of read performance (e.g., unordered keys in leaf nodes). Thus, VEGITO should optimize tree-based index for concurrent writes and reads.

4 DESIGN AND IMPLEMENTATION

To overcome the challenges, we introduce a classic concept (*epoch*) into a new context (HTAP). A centralized (epoch) oracle partitions time into non-overlapping epochs.[6] Epoch is the granularity at which VEGITO guarantees the consistency and visibility of backup/AP replicas to analytical queries. It opens opportunities to exploit parallelism and preserve locality for providing consistent, fresh, and columnar backups. In this section, we detail main techniques in our epoch-based solution employed by VEGITO.

4.1 Consistent and Parallel Log Cleaning

To provide *consistent* backups for OLAP workloads, log cleaner threads on multiple machines should apply logs following the *dependency* in transactions; all logs of one transaction should be applied *atomically*, and all logs from different transactions should be applied *in order*.

A traditional approach is to assign global or vectorized timestamps to logs of transactions and then apply logs sequentially at both machine and thread levels according to their timestamps [55, 65, 101]. When facing millions of transactions per second, this approach would incur excessive cost to transaction processing, and sequential log cleaning would be extremely slow (see *GTS+SEQ* in Fig. 6b).

The epoch-based approach simplifies the assignment and comparison of timestamps with a local scalar value (epoch number), and also allows logs within an epoch (assigned the same epoch number) to be drained in parallel. However, dependent transactions at the epoch boundary must be assigned epochs matching the serial order; namely, committed transactions in earlier epochs never transitively depend on transactions in later epochs. Further, logs in different epochs should still be drained in order.

Consistent epoch assigning. VEGITO introduces a lightweight gossip-style scheme to assign consistent epoch numbers for dependent transactions. An epoch oracle will periodically broadcast a new epoch to update the transaction epoch number (Epoch/TX) on each machine atomically; it always waits for ACKs from all machines such that the epoch gap among machines must not be bigger than 1. Each transaction will assign Epoch/TX on machines involved to its logs during committing (see Fig. 8). For stand-alone dependent transactions on the same machine, the order of epoch numbers can always agree with the serial order due to using the concurrency control scheme (e.g., 2PL or OCC), similar to Silo [98]. For distributed transactions, the epochs from different machines are likely the same, which means all transactions on these machines are in the same epoch. In rare cases, the distributed transaction involves machines within different epochs, as shown in Fig. 7. The distributed transaction TX_1 executes on two machines and observes the epoch on one machine (M1) is behind the other (M2). Suppose a local transaction TX_2 on M1 depends on TX_1, assigning a smaller epoch number (Epoch/TX=4) to TX_2's log would violate the serial order. To avoid this, TX_1 is responsible for synchronizing the epoch (Epoch/TX=5) on machines involved using point-to-point messaging (*gossip*), so that TX_2 will commit its log with the correct epoch number (5).

Fig. 8 shows the commit protocol [34, 107] with a new distributed epoch synchronization step.[7] It first gains the latest epoch number (Line 1-3) from machines involved (mset), and then updates the epoch if needed (Line 4-6). Specifically, we can blindly overwrite the epoch instead of using atomic operations as there are at most two epoch numbers across the cluster. Finally, it will send logs to backups with the consistent epoch number (Line 8). Note that epochs are synchronized after write locking and before read validation. Placing it after write locking ensures that all transactions in the later epochs would see at least the conflict tuple locked; placing it before read validation ensures that committed transaction in earlier epoch never reads the tuple updated by transactions in later epoch. Thus, assigning epochs obeys both dependencies and anti-dependencies.

Parallel log cleaning. Logs from different machines (and threads) are buffered in different queues [26, 34], and mul-

[6] Note that the oracle only needs to periodically broadcast a new epoch to all machines in the cluster, instead of transactions or queries involved. Hence, it will definitely not be the bottleneck in the cluster with thousands of machines even using very small epochs (e.g., a few milliseconds) [15, 30].

[7] The single-machine epoch scheme usually replies on total-store-order (TSO) architectures (like x86-64) to synchronize the epoch among worker threads, like Silo(R) [98, 116].

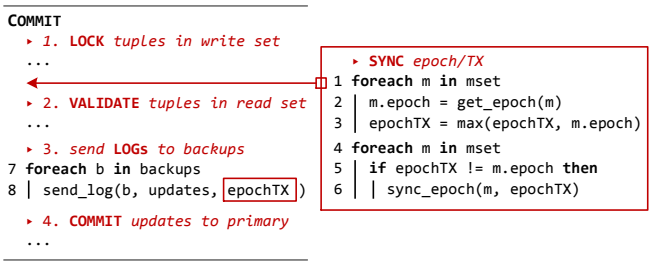


Fig. 8. *Commit protocol with a lightweight epoch scheme run at the end of every transaction.*

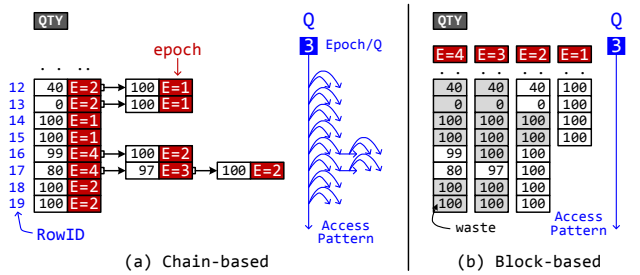


Fig. 9. *Different designs of a multi-version column store for QTY attribute in* ITEM *table. The grey box indicates that the entry is wasted since the tuple is not changed in this epoch.*

tiple cleaner threads can drain logs of the same epoch in parallel. VEGITO uses a hybrid design to exploit both intra-machine and inter-machine parallelism. First, each machine maintains a cleaner epoch number (Epoch/C), meaning that logs at this epoch have been drained. As shown in Fig. 7, after the cleaner thread on M2 applies the last two logs at epoch 4 from M1, the cleaner epoch will increase to Epoch/C=4, and then two cleaner threads could start to drain logs at epoch 5 on M2 in parallel. The runtime schedules queues dynamically across available cleaner threads to achieve load balance [81].

Second, to reduce waiting time among cleaner threads, VEGITO refrains from the synchronization among the cleaner threads on different machines, so that the backup replicas on different machines may not keep up the pace of change. For example, in Fig. 7, the cleaner threads on M1 and M2 are draining logs in different epochs (5 and 4 respectively). To remedy this, VEGITO supports multi-version backup replicas (see §4.2) and makes analytical queries read consistent backups at a (stable) query epoch (Epoch/Q), which is the minimum value of cleaner epochs on machines involved, like Epoch/Q=3 in Fig. 7.

4.2 Locality-preserving Multi-version Column Store

To avoid contention between cleaner threads (write) and AP threads (read), backup/AP replicas require to adopt a multi-version *column* store (MVCS) at the epoch level. Specifically, the cleaner threads will generate a new version of column store for each epoch by applying logs in parallel. Meanwhile, AP threads will run analytical queries over the latest stable version of the column store to retrieve *fresh* results.

The chain-based design is widely used by multi-version (row) stores [33, 39, 50, 58, 60, 110]. Prior HTAP systems [20, 68, 84] also follow this design. As shown in Fig. 9(a), the column store maintains an array for each attribute to store the latest value of tuples with its version (e.g., epoch). Each entry also maintains a backward chain for values in the earlier versions, which is designed to support atomic updates efficiently. Before updating new value in-place, the cleaner thread will copy the original entry and update the chain atomically. Although this design preserves the locality for recent values, analytical queries commonly access the latest *consistent* data. For example, in Fig. 9(a), the cleaner threads are currently draining logs at epoch 4, and the queries can only access the values up to epoch 3. Consequently, the AP thread has to traverse the chains of updated entries and frequently check the versions (see the access pattern in Fig. 9(a)). Besides, the garbage collection for chains would also be complicated and time-consuming.

VEGITO proposes a *block-based* design to exploit optimal performance for analytical queries. As shown in Fig. 9(b), the design is straightforward, which maintains an array for each epoch. When starting a new epoch, the cleaner thread copies the array of last epoch and applies logs to it. This creates a complete snapshot on demand in each epoch. Given an epoch, the AP thread can scan the array with perfect locality but without interference from the cleaner threads. Moreover, the cleaner thread can also garbage collect expired arrays efficiently and reuse the memory easily. However, this design has an apparent and critical drawback (see Fig. 20 in §6.7): data copying may waste lots of CPU and memory resources, especially for append-only attributes (e.g., the attributes in ORDER table). In Fig. 9(b), most of the entries are wasted (grey box) to repeatedly store tuples, which are not changed in the current epoch.

To remedy this, we optimize the naive block-based design by exploiting both *spatial* and *temporal* locality observed in transaction workloads, the data source of MVCS.

Row-split. We observe that *the transactions may focus on updating tuples in a small scope for a while*, like discounted products, batch orders, and social events. Thus, VEGITO first splits values into multiple pages, and each page enables a copy-on-write mechanism independently to implement fine-grained on-demand data copying. There are no new copies for pages without updates at the current epoch. As shown in Fig. 10(a), values of attribute QTY are grouped into two pages. The first page has only two copies for epoch 1 and epoch 2 (i.e., E=1 and E=2), since there is no update in later epochs. To balance the read (AP threads) and write (cleaner threads) performance of the multi-version column store, VEGITO uses 4KB page size, which is enough to exploit the cache locality [44, 91]. Although an insert can be treated as a normal update and triggers the copy-on-write mechanism for a new epoch as well, it is costly for insert-mostly tables

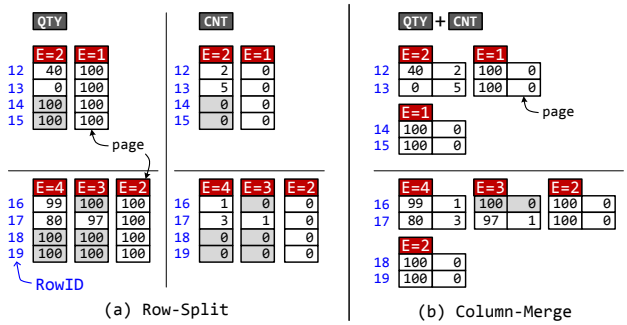


Fig. 10. *Optimizations for block-based multi-version column store with (a) row-split and (b) column-merge for QTY and CNT attribute in* ITEM *table.*

(e.g., ORDER). VEGITO avoids page copying from inserts by appending new values and maintaining offsets for each epoch.

Column-merge. We further observe that *a certain type of transactions usually updates a fixed set of attributes at a time*. For example, NEWORDER transaction in Fig. 3 always updates two attributes (QTY and CNT) of ITEM table together. Thus, VEGITO will merge related attributes for the same tuple into a single page. As shown in 10(b), QTY and CNT attributes of ITEM table are merged. Using column-merge improves the performance of draining logs for cleaner threads and also reduces data copying operations with the same page size, due to finer-grained partitioning for each attribute. VEGITO can automatically discover correlations between attributes from transaction logs and reorganize them into a single page at the start of next epoch.[8] The epoch-based reorganization will not interfere with running queries at all since new pages will not be read by current analytical queries. Further, analytical queries could benefit from the optimization two epochs later.

Finally, after enabling the two optimizations, only 2% of updates incur page copying when using 4KB page size and 15ms epoch interval in a typical HTAP workload (i.e., CH-benCHmark [29]). The median latency to copy a 4KB page is about 6 microseconds. Moreover, our two optimizations are orthogonal to the preceding techniques for the column store [32, 40], such as compression and range filter, so both are applicable in a complementary manner. We leave it to future work.

4.3 Two-phase Concurrent Index Updating

The order-preserving indexes commonly use tree-based data structures to support range scan operations. The update operation may involve more complicated steps (e.g., traversal and split), which will cause new challenges to support fast concurrent updates (by cleaner threads) and lookups (by AP threads). Without loss of generality, the rest of this paper will use B$^+$-tree to explain the issues and introduce our design, since it is widely adopted by OLTP and OLAP systems [37, 43, 47, 101, 105, 109].

[8] We collect the statistics on the column family to decide how to merge columns with conflicting requirements.

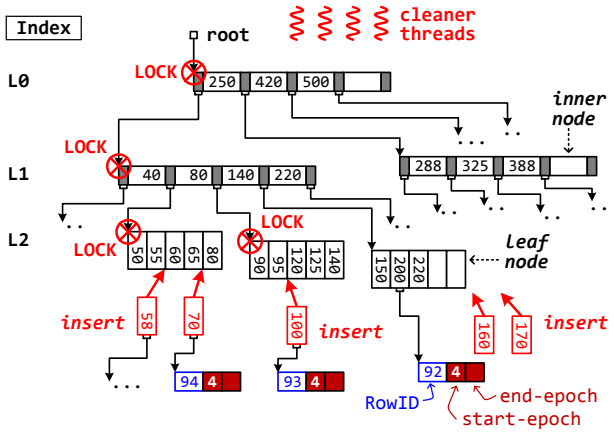


Fig. 11. *An example of traditional concurrent index updating.*

As shown in Fig. 11, B$^+$-tree contains two types of nodes: inner nodes and leaf nodes. The inner node stores the values and links to the next level. The last level (L2) contains (sorted) leaf nodes, which are used to store the sorted value of the indexed attribute (AMOUNT) with a link to the row ID and its start/end epochs. The end epoch is used to delete a value, which will simply write an end epoch. The cleaner thread will garbage collect expired values in a lazy and batched manner. The update operation is treated as a delete operation for the original value and an insert operation for the new value. So that we mainly consider the insert operations by cleaner threads and the lookup operations by AP threads.

The **INSERT** operation consists of three steps.

- **Locate leaf node.** Search a leaf node to store the value by traversing from the root
- **Split/Insert leaf node.** Split the leaf node if it is full, and then insert the value into the sorted leaf node.
- **Split/Insert inner node.** (*Recursively*) Split the upper-level inner node if it is full, and then insert a value and a link into the sorted inner node.

We observe that the throughput of insert operations drops with the increase of threads, even using an optimized B$^+$-tree [104] (see Fig. 21(a) in §6.8). The main reason is that the second step of the insert operation may block other concurrent insert operations. As shown in Fig. 11, for inserting the value 70, the cleaner thread has to recursively lock the leaf node and the upper-level inner nodes due to node split. It will block the concurrent insert operations on the whole subtree (e.g., 58 and 100). Even worse, the node split may cause some blocked operations to lock or rollback the first step since the leaf node has changed (e.g., 58 and 70). Even no split, the second step may still hold the lock of the leaf node for a long time, since it has to move values for keeping

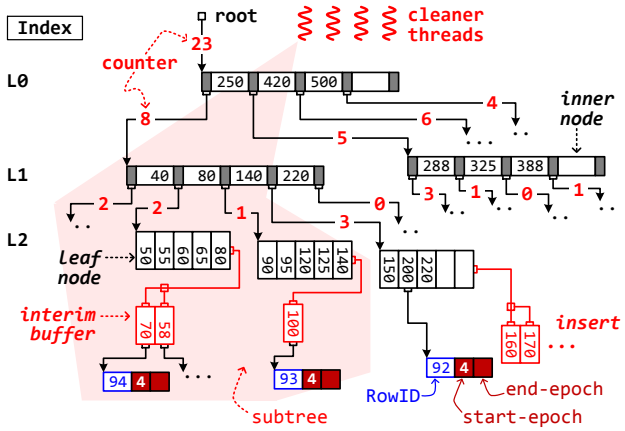


Fig. 12. *An example of two-phase concurrent index updating.*

them in order. However, these efforts may be in vain, as the later insert may move these values again (e.g., 160 and 170). On the other hand, the existence of insert operations will also significantly impact the performance of lookup operations by AP threads (see Fig. 21(b) in §6.8). The lookup operation has to protect access to the inner/leaf node and the value, since the insert operations may change them concurrently.

Fortunately, the epoch-based design provides an opportunity to fully parallelize the index updating. More specifically, the insert operations in the current epoch only need to become visible by the lookup operations until the next epoch. VEGITO introduces a *two-phase index updating* mechanism that splits the insert operations within an epoch into two phases (i.e., *location* and *update*) and parallelizes them using two different approaches (*task parallelism* and *data parallelism*), as shown in Fig. 12.

In the location phase, each thread searches a leaf node to append the value into its interim buffer, and recursively updates the counter at each level. The interim buffer is unordered, and atomic instructions (e.g., CAS) are used to append the value and increase the counter. VEGITO simply uses a vector to implement the interim buffer and resize it according to workloads. Note that the interim buffer is absolutely transparent to lookup operations; thus there is no read/write conflict.

In the update phase, we first use a top-down greedy strategy to partition the tree into non-overlapping subtrees according to the counters at each level, so that each subtree has a similar amount of tasks. Then each thread will insert the values within a subtree in a batch, which also avoids redundant node splits and data movements. Finally, we use a single thread to split the top-level (L0) node if necessary. Consequently, there are no conflicts between cleaner threads in both location and update phases. However, the lookup operations (by AP threads) still may conflict with the update phase of the insert operations. To minimize the impact on lookup operations, cleaner threads can leverage RCU [64] mechanism or HTM [41] to implement the update phase.

5 No Compromise: Availability

VEGITO assumes that the OLTP system has already provided high availability using replication (e.g., 3-way primary-backup replication [26, 34, 46]) and other fault-tolerant techniques (e.g., failure detection and non-volatile WAL). VEGITO reuses this mechanism to support HTAP workloads and still preserves the same availability guarantees for free—namely, *there is no need for extra replicas*. Because VEGITO just reorganizes the *data layout* of one backup replica (backup/AP), from row-wise store to column-wise store; the backup/AP can still provide the capability of failure recovery. Besides, the original recovery protocol [26, 34] is used as usual in most common cases.

Backup failure. When the backup/TP fails, VEGITO will rebuild a row-wise backup from the primary by following the original protocol. When the backup/AP fails, VEGITO will rebuild a column-wise backup to the next epoch, because both the primary and the backup/TP do not store epochs associated with tuples for memory savings and good locality. Meanwhile, VEGITO needs to re-execute analytical queries involved with the new epoch.

Primary failure. When the primary fails, VEGITO always *prefers* to promote a surviving backup/TP to be the new primary and rebuild a new backup/TP on another machine later in the background, which still follows the original protocol. When both the primary and the backup/TP fail (a rare case), VEGITO rebuilds a new primary based on the surviving backup/AP on the same machine (∼42 ms for 12 warehouses of TPC-C, see §6.5), instead of promoting it, and then migrates the backup/AP to another machine later in the background. This *rebuild-and-migrate* design avoids lengthy data reorganization between row store and column store, compared to the conventional *promote-and-rebuild* approach. Note that our block-based design also simplifies and accelerates this procedure. Therefore, VEGITO can still offer comparable performance against promoting a backup/TP (∼7% overhead). In addition, it also avoids interrupting analytical queries.

It should be noted that the recovery scheme prefers OLTP performance. VEGITO chooses to abort the analytical query that accesses failed machines and retry it after recovery, since the long-running analytical query is unusual in HTAP workloads [29] (e.g., real-time analytics), especially for in-memory systems. If the long-running analytical query is a serious problem, for example the query latency exceeds the mean time to failure (MTTF) of HTAP systems, both the primary and the backup/TP should store epochs associated with tuples. VEGITO thus could suspend and resume the analytical query after recovery.

6 Evaluation

We implemented VEGITO by extending DrTM+H [107], a state-of-the-art distributed in-memory OLTP system. The

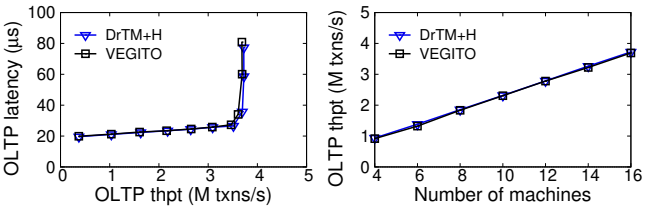


Fig. 13. *Comparison of (a) performance and (b) scalability between* VEGITO *and DrTM+H using CH-benCHmark with OLTP-only workloads.*

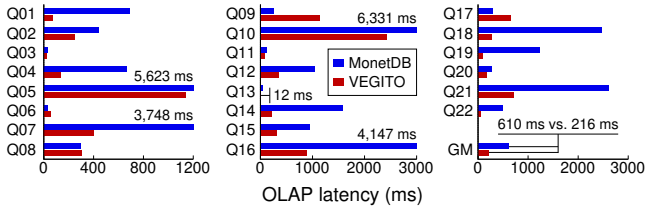


Fig. 14. *Comparison of single-threaded latency (ms) between* VEGITO *and MonetDB using CH-benCHmark with OLAP-only workloads, equivalent to TPC-H with SF=10.*

extensions include retrofitting the high availability mechanism (3-way primary-backup replication) for hybrid transaction/analytical processing and integrating a distributed in-memory OLAP engine, similar to MonetDB [6, 44], a column-store database that maps analytical queries into a series of array operations [23].

6.1 Experimental Setup

Hardware configuration. All experiments were conducted on a rack-scale cluster of 16 machines. Each machine has two 12-core Intel Xeon processors, 128GB of RAM, two ConnectX-4 100Gbps IB NICs and an Intel 10GbE NIC. Unless otherwise noted, we reserve 4 cores for log cleaner threads and 2 cores to generate transactions and analytical queries in parallel for local worker threads, which avoid the impact of networking between clients and servers, as done in prior work [26, 97, 98, 105, 111]. For HTAP workloads, we pin 8 TP threads and 10 AP threads on the remaining cores.

Benchmarks. We use CH-benCHmark [29], a typical HTAP benchmark derived from unmodified TPC-C [95] (OLTP benchmark) with some necessary tables to fulfill equivalent queries from TPC-H [96] (OLAP benchmark). It contains 5 types of transactions and 22 analytical queries. We run the full mix and report OLTP throughput as the number of NEWORDER transactions committed per second and OLAP throughput as the number of analytical queries executed per second. CH-benCHmark scales by partitioning a database into multiple warehouses spreading across multiple machines. We deploy 12 warehouses on each machine with 3-way replication, namely 12 primary, 12 backup/TP and 12 backup/AP replicas. Each machine hosts about 6GB initial data, which rapidly grows up to 75GB (a total of 1.2TB) through continually running transactions (e.g., NEWORDER) for about 40 seconds. To eliminate the effect of growing data, the analytical query will access a fixed size of latest data by using LIMIT statement.

Comparing targets. We choose DrTM+H [107] and MonetDB [6] (v11.33.3) as the representative in-memory OLTP and OLAP systems, respectively, to show that both OLTP and OLAP performance of VEGITO are comparable to specialized counterparts. VEGITO follows 3-way replications of DrTM+H except for replacing one of backup/TP replicas with one backup/AP replica. To eliminate the performance discrepancy, VEGITO also uses the query plans generated by MonetDB for all of the analytical queries. VEGITO optimizes distributed joins as MemSQL by adding reference tables (copies) on each machine and aggregating intermediate results to avoid the whole table transferring [4]. In addition, the default intervals of epoch and garbage collection (GC) are set as 15 ms and 1 second, respectively.

For HTAP workloads, we mainly focus on the performance degradation and the freshness in VEGITO against three state-of-the-art HTAP systems with three different architectures—namely TiDB v4.0 [8] with TiFlash [9] (DUAL-SYSTEM), the community edition of MemSQL v7.0 [4] (SINGLE-LAYOUT), and SQL Server 2019 [7] (DUAL-LAYOUT). Note that MemSQL is an in-memory system, while TiDB and SQL Server are on-disk systems. For SQL Server, we host all data in main memory by using tmpfs, an in-memory file system. TiDB demands all data on the disk with the ext4 file system. In addition, we deploy TiDB on the cluster with different settings[9] and always report the best results of them. Differently, MemSQL and SQL Server can only run on a single machine, and we deploy them on one of our testbed machine without replication (just 12 warehouses). Finally, to avoid the impact of compiling and interpreting analytical queries, we directly evaluate the performance of executing analytical queries on servers.[10]

6.2 Overall Performance

OLTP-only workloads. We first compare OLTP performance of VEGITO and DrTM+H using CH-benCHmark with OLTP-only workloads, like TPC-C [95]. As shown in Fig. 13, the peak throughput of VEGITO reaches 3.7 million NEWORDER transactions per second when running the full mix on 16 machines (each has 14 TP threads), just 1% lower than DrTM+H. This is thanks to our epoch-based scheme and gossip-style parallel log cleaning, which avoid blocking transactions. The best published TPC-C performance we know of is from FaRMv2 [87], which can commit 5.4 million NEWORDER transactions per second on 90 machines. In

[9] As recommended in TiDB's official website [9], we deployed TiKV and TiFlash in both the same and different nodes.

[10] The systems evaluated in our paper use different approaches to run analytical queries—namely VEGITO (hand-written C++), MonetDB (interpreted SQL), MemSQL (compiled SQL), TiDB (complied SQL), and SQL Server (compiled SQL).

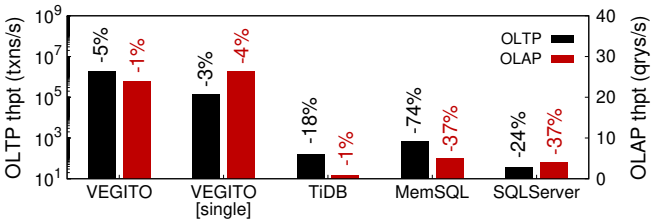


Fig. 15. *Comparison of OLTP and OLAP performance of different HTAP systems using CH-benCHmark with hybrid workloads. The labels upon histogram are performance degradation. Note that the left y-axis (OLTP throughput) is in log scale.*

general, VEGITO can offer OLTP performance comparable to state-of-the-art specialized systems (e.g., DrTM+H and FaRMv2) and scales well on a cluster with tens of machines.

OLAP-only workloads. We further compare OLAP performance of VEGITO and MonetDB using CH-benCHmark with OLAP-only workloads, like TPC-H [96]. Fig. 14 compares the single-threaded latency of analytical queries on VEGITO and MonetDB; both of them use the same query plans generated by MonetDB. To compare with the published TPC-H results [35, 38, 67], we scale the database in CH-benCHmark following a similar approach in TPC-H by a scale factor of 10 (SF=10). As shown in Fig. 14, the average (geometric mean) latency of VEGITO (GM) outperforms MonetDB by 2.8× (216 ms vs. 610 ms). The main performance improvement in VEGITO is due to combining some operators manually and using efficient string operations by hand-written C++. VEGITO also outperforms published TPC-H results for various query processing engines [35, 38]. Specifically, the average (geometric mean) latency of all TPC-H queries (SF=10) using a single thread is 568 ms for HyPer [68], 541 ms for Umbra [67], 1,125 ms for Hyrise [36] and 619 ms for MonetDB [6].[11] Overall, VEGITO's OLAP performance matches state-of-the-art specialized systems.

HTAP workloads. Fig. 15 shows both OLTP and OLAP throughput of VEGITO and other available HTAP systems using CH-benCHmark with hybrid workloads. To study performance degradation, we evaluate each system twice. We first run OLTP and OLAP workloads separately and tune the number of clients to use half of CPU resources. Then, we run HTAP workloads with the same number of clients to saturate CPU resources with a balance between OLTP and OLAP engines. The results of performance degradation in Fig. 15 (labels) are the difference between the two runs.

VEGITO can perform 1.9 million TPC-C NEWORDER transactions and 24 TPC-H-equivalent queries per second simultaneously. The OLTP throughput of VEGITO is several orders of magnitude higher than that of its competitors (11,808× for TiDB, 2,911× for MemSQL, and 53,138× for SQL Server). This means that the bridge between two ends of the world in VEGITO—parallel log cleaning, column store

[11] Note that we calculate the geometric mean of the query times based on the reported results of every query [35, 38].

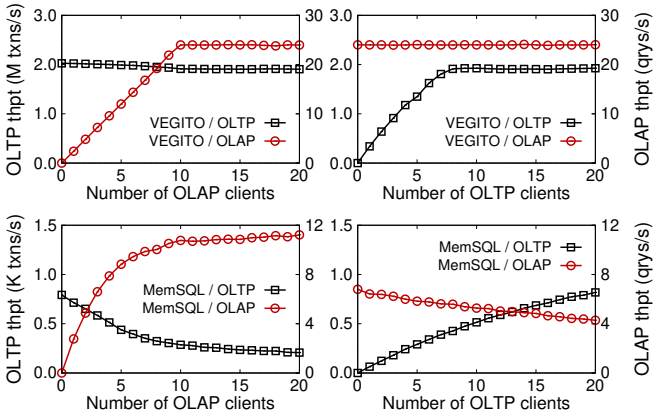


Fig. 16. *Performance degradation on* VEGITO *and MemSQL with the increase of OLAP and OLTP workloads, respectively.*

building, and tree-based index updating—is strong enough to face the challenge of extremely high throughput, which is never appeared in prior published results of HTAP systems, to the best of our knowledge. Compared to single-machine HTAP systems, like MemSQL and SQL Server, VEGITO still has orders of magnitude higher OLTP throughput per machine (120 K txns/s) with support for scaling out and fault tolerance.

Moreover, VEGITO also provides little throughput degradation when running hybrid workloads, just 5% for OLTP and 1% for OLAP respectively. In contrast, existing HTAP systems, TiDB, MemSQL, and SQL Server, suffer from significant performance degradation, reaching 18%, 74%, 24% for OLTP and 1%, 37%, 37% for OLAP respectively. It matches well with the characteristics of different HTAP architectures (see Fig. 1).

We further deploy and evaluate VEGITO on a single machine by hosting all three replicas of each shard (one primary and two backups) on the same machine. VEGITO still synchronously send transaction logs between primary and backups by the NIC. As shown in Fig. 15, on a single machine, VEGITO can perform 132 thousand TPC-C NEWORDER transactions and 26.5 TPC-H-equivalent queries per second simultaneously. Note that running analytical query on a single machine is more efficient due to eliminating network overhead.

6.3 Performance Degradation

To study the impact of performing hybrid workloads simultaneously, we follow Gartner's recommendation to instruct one kind of clients (e.g., OLTP) to sustain a configured throughput (i.e., about half of peak throughput) and allowing another kind of clients (e.g., OLAP) to saturate the throughput [28].

VEGITO can provide strong performance isolation by dedicating a fixed number of worker threads for OLTP and OLAP workloads. We carefully put the memory of two classes of threads into different cache lines (e.g., write epoch and read epoch, write offset and read offset) to mitigate the impact

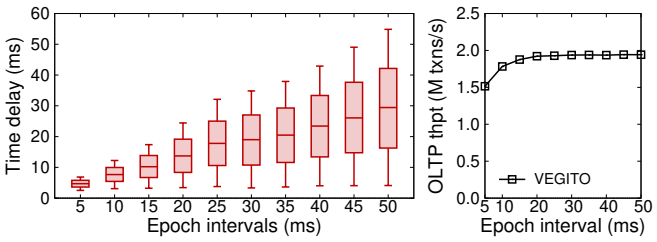


Fig. 17. *(a) The time delay and (b) OLTP throughput of* VEGITO *with the increase of epoch intervals in the failure-free case.*

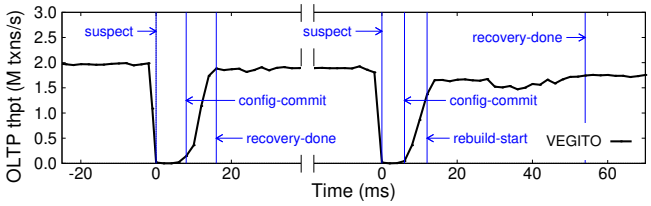


Fig. 18. *The timeline of failure recovery.* ***suspect****: the failed machine is detected;* ***config-commit****: new configuration is committed at all surviving machines;* ***recovery-done****: the recovery of primary is done;* ***rebuild-start****: backup/AP starts to rebuild primary.*

on the cache. As shown in Fig. 16, with the increase of AP clients, OLTP performance degradation of VEGITO is less than 5%. After OLAP performance is saturated, OLTP performance also remains stable. When the roles are reversed, OLAP performance degradation becomes trivial (1%) since OLAP worker threads always use a stable epoch to perform analytical queries on a specified column store and index. In contrast, MemSQL suffers from severe performance degradation of both OLTP and OLAP workloads, even if there are adequate resources. In Fig. 16, the performance degradation of MemSQL reaches 74% and 37% for OLTP and OLAP respectively, with the increase of another type of workloads. This is largely due to the high contention between OLTP and OLAP engines over shared data.

6.4 Freshness

The freshness is defined as the maximum time delay between an update was committed by the transaction (OLTP workload) and this update can be read by the analytical query (OLAP workload). Fig. 17(a) shows the freshness of VEGITO with the increase of epoch intervals in the failure-free case. The median time delay is about 70% of the epoch interval, and the maximum delay is up to 1.3× of epoch interval. It implies that we could roughly limit the freshness in VEGITO by setting an appropriate epoch interval.

Moreover, by setting the epoch interval, there would be a tradeoff between the freshness (OLAP) and the performance degradation (OLTP) in VEGITO. In Fig. 17(b), when using a relative short epoch interval (less than 10 ms), performance degradation would become non-trivial (10%) since epoch-based design limits the parallel log cleaning within an epoch, and the cost to build a column store for each epoch is hard to be amortized. Considering the latency of analytical queries (see Fig. 14), the epoch interval with tens of milliseconds would be moderate and reasonable. The default epoch interval is set as 15 ms, providing a freshness less than 17.4 ms.

As a reference, on our testbed, the maximum delay in TiDB, MemSQL, and SQL Server are about 1,534 ms, 1.2 ms, and 46 ms, respectively. The results are compatible with the characteristics of different HTAP architectures (see Fig. 1). Further, VEGITO can provide a comparable failure-free freshness with Amazon Aurora [101], which reports the read replica typically lags behind the writer by 20 ms or less.

6.5 Recovery

VEGITO follows a 3-way primary-backup replication of DrTM+H except for replacing one of backup/TP replicas with one backup/AP replica. During the evaluation, we kill one machine by turning off its networking, and the primary on the failed machine will be recovered by promoting its backup/TP on one of the surviving machines. We disable the primary to re-replicate a new backup/TP for emulating a rare case. Then, we kill the recovered primary again, and its backup/AP will be used to rebuild a new primary locally and migrate itself to another machine in the background.

Fig. 18 shows the timeline with OLTP throughput aggregated at 2 ms intervals, which is a zoomed-in view around the failure. VEGITO uses about 10 ms for failure detection and reconfiguration. Promoting backup/TP to primary takes about 8 ms, and rebuilding primary based on backup/AP takes 42 ms for 12 warehouses with the initial size (about 2GB). Note that the recovery load is handled by a single machine (limited by DrTM+H), causing a relatively long rebuilding time that mainly depends on the data size. Thus, it could be easily balanced across the cluster by fine-grained sharding [34, 69]. The throughput is not fully recovered since the failed machines are not back. Besides, rebuilding primary will slightly impact on throughput (10%) due to sharing CPU cores.

6.6 Parallel Log Cleaning

To study the performance impact of different log cleaning approaches, we implement three approaches on VEGITO.

- **Parallel/Inconsistent**: a fully parallel scheme used by OLTP-specific systems [26, 34], which can provide high availability but not ensuring the consistency of backups.
- **GTS+SEQ**: a global timestamp-based scheme used by prior HTAP systems [55, 65, 103, 112], which provides consistent backups by draining logs in a sequential way.
- **Parallel/Consistent**: a lightweight gossip-style scheme used by VEGITO, which also ensures the consistency of backups but drains logs in parallel.

Fig. 19 shows the throughput of OLTP and log cleaning with the increase of machines. *Parallel/Inconsistent* is used by OLTP-specific system to build fault-tolerant backups (see

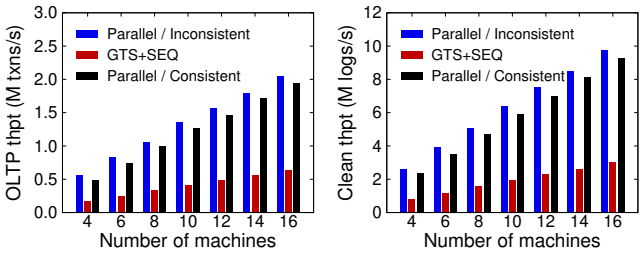


Fig. 19. *The comparison of (a) OLTP and (b) clean throughput for different log cleaning approaches using CH-benCHmark.*

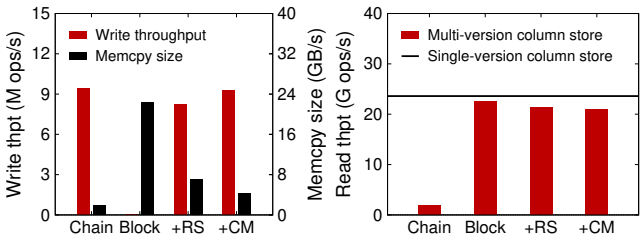


Fig. 20. *The comparison of (a) clean throughput, memory copy size, and (b) read throughput for different types of multi-version column stores.*

FTbackup in Fig. 6(b)), which are not consistent for analytical queries. *GTS+SEQ* just achieves up to 31.5% and 30.9% throughput for OLTP and log cleaning, respectively. There are two main reasons. First, assigning a global timestamp for each transaction will increase the execution time. Second, draining logs in a sequential way limits the throughput of cleaner threads and further blocks the execution of transactions. By contrast, for OLTP and log cleaning, our approach in VEGITO (*Parallel/Consistent*) only incurs about 4.5% and 4.7% slowdown compared to *Parallel/Inconsistent* and outperforms *GTS+SEQ* by up to 3.0× and 3.1×. It can drain about 9.3 million 1 KB logs per second in parallel. According to the TPC-C specification, there are 1% of accesses to a remote warehouse in NEWORDER transactions by default [95], resulting in about 9% of distributed transactions. Consequently, our gossip-style scheme only increases 7% of remote accesses due to the epoch synchronization step in the commit protocol (see Fig. 8). In the worst case, namely 100% of distributed NEWORDER transactions, our approach can still limit the performance degradation of OLTP throughput to 15% or less. The overhead of additional remote accesses increases to 21%.

6.7 Multi-version Column Store

For multi-version column store (MVCS) in VEGITO, the conventional (chain-based) approach could achieve the best performance to build the store (by cleaner threads) but the worst performance to scan the store (by AP threads). To study the effect of our locality-preserving design and optimizations, we implement four types of MVCS on VEGITO and report the steady-state throughput for them.

- **Chain**: a chain-based design [20, 68, 84].
- **Block**: a block-based design without optimizations.
- **+RS**: a block-based design with row-split optimization.
- **+CM**: a block-based design with row-split and column-merge optimizations.

As shown in Fig. 20, as expected, *Chain* can achieve the best write throughput (9.4 M ops/s), which outperforms the naive block-based design (*Block*) by 157× due to fewer memory copy operations. On the contrary, *Block* can provide about 95% of read throughput over a single-version column store, which outperforms *Chain* by about 12.4×. However, both write and read throughputs are important for HTAP systems. The row-split optimization (*+RS*) can achieve about 87.3% of write throughput of *Chain* and 95.4% of read throughput of *Block*. The column-merge optimization (*+CM*) further provides a tradeoff between two operations. It improves write throughput by 13% due to exploiting the locality of attributes updated by transactions, while reduces 2% of read throughput since one column of tuples will spread more pages.

Further, GC for the block-based design is very efficient and incurs a negligible impact on OLAP performance. It only uses one core with less than 10% of CPU utilization lasting about 70 ms (retrieve about 4.8GB), compared to 35% and 350 ms used by GC for the chain-based design.

6.8 Concurrent Index Updating

To study the performance of different tree-based indexes with concurrent read and write operations, we compare three typical data structures.

- **STX+HTM**: a generally-used C++ B$^+$-tree library [21], using hardware transactional memory (HTM) to support multiple writers and readers, as done in DBX [105].
- **Masstree** [61]: a trie-like concatenation of B$^+$-trees with cache-friendly design, using a combination of fine-grained lock and version.
- **B$^+$-tree w/ 2PU**: a standard B$^+$-tree with two-phase concurrent index updating, which is adopted by VEGITO.

We first evaluate the performance of insert operations (write-only) with the increase of worker threads using write-intensive transactions (NEWORDER) in CH-benCHmark. As shown in Fig. 21(a), *STX+HTM* does not scale with the increase of writers due to heavy contentions on node splits. *Masstree* is heavily optimized for concurrent operations by using fine-grained locks and optimistic mechanism, but it still cannot avoid contention thoroughly. For *B$^+$-tree w/ 2PU*, the insert operation is very efficient (single writer) due to using a lazy and batched manner to avoid redundant operations (node splits and data movement). Moreover, *B$^+$-tree w/ 2PU* also scales well with concurrent writers, thanks to avoiding

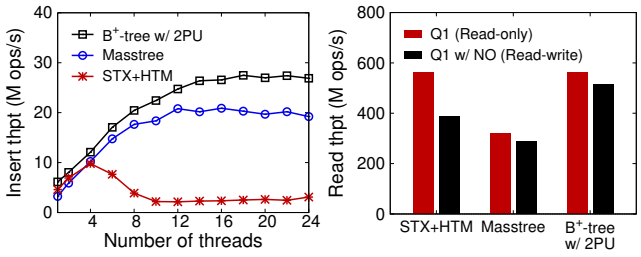


Fig. 21. *The comparison of (a) insert and (b) read throughput for different tree-based indexes.*

redundant node splits and data movements. Therefore, *B^+-tree w/ 2PU* outperforms *STX+HTM* and *Masstree* by up to 8.7× and 1.4×, respectively.

We further evaluate read performance for different indexes using Q01 from CH-benCHmark with and without NEWORDER transactions (NO), as shown in Fig. 21(b). For the read-only workload, *STX+HTM* achieves the best performance (564 M ops/s) by using more balanced tree, while *Masstree* just provides 56.6% of throughput (319 M ops/s) due to out-of-order keys in leaf nodes. For the read-write workload, the read performance in *STX+HTM* significantly decreases by 31% due to massive read-write contentions, and *Masstree* can still achieve 290 M ops/s. The read throughput of *B^+-tree w/ 2PU* achieves 563 M and 515 M ops/s for read-only and read-write workloads, respectively, which is competent for HTAP workloads.

7 RELATED WORK

HTAP systems. The increasing importance of real-time operational analytics has stimulated considerable interest in both academia and industry. There are three classes of systems.

DUAL-SYSTEM. Connecting two specialized systems is a common design alternative [56, 62, 71, 80, 82, 112]. Recently, several systems [42, 55, 65, 103] also propose to use a single node (primary) for OLTP workloads and multiple nodes (backups) for OLAP workloads, where transaction logs are shipped to backups *asynchronously*. Google F1 Lightning [112] is a loosely coupled HTAP solution (HTAP-as-a-service) that aims at providing a transparent experience to OLTP systems. TiDB [43] is a Raft-based HTAP database that *asynchronously* replicates logs from a row store (TiKV) to a column store (TiFlash). MySQL allows running analytical queries on (row-based) backups and provides semi-synchronous replication [66]. Further, many cloud databases also allow OLTP and OLAP workloads to run on different instances, which are also replicated by log shipping in the background, like Amazon Aurora [1] and MS Azure [5]. Differently, VEGITO runs analytical queries over multi-version columnar backups for *efficiency* and ships transaction updates before committing on the primary for *freshness*.

SINGLE-LAYOUT. There are several efforts aiming at building HTAP systems from one specialized system (i.e., OLTP or OLAP) [3, 13, 48, 84, 88]. HyPer [48] is an in-memory HTAP system, which leverages hardware-assisted virtual memory snapshots, session-based OLAP, and hot/cold page management [49] to maintain consistent snapshots for OLAP. AnKer [88] leverages virtual memory snapshots and adds new system calls to accelerate page copying. L-Store [84] introduces an update-friendly lineage-based data store to support both OLTP and OLAP workloads. Many SQL-on-Hadoop systems [3, 25, 31] have extended existing OLAP engines with transactional support. Using a single layout may prohibit certain optimizations (e.g., frequency compression [79]) and cause poor performance for part of workloads [14]. To avoid data contention between transactions (read-write) and analytical queries (read-only), MVCC scheme becomes essential. Prior work [53, 68] has also reported 20–45% throughput degradation due to using MVCC schemes even under low contention.

DUAL-LAYOUT. Recent systems support HTAP workloads by combining two different data layouts in a single system [4, 11, 14, 22, 54, 60, 90]. MemSQL [4] adopts an in-memory row store for OLTP workloads at scale and an on-disk column store for OLAP workloads. SAP HANA [90] stores records in either row or column format for both transactional and analytical workloads. It further uses life cycle management to ship and merge records asynchronously. SQL Server [37, 54] has added updatable columnstore indexes and batch mode processing to speed up analytical queries. Peloton [73] proposes a hybrid data layout (i.e., FSM [14]) for HTAP workloads, which stores tuples with different formats and supports online reorganization. BatchDB [60] alternates between the execution of transactions and a batch of queries (e.g., 200 ms). OLTP updates are first queued and then propagated to OLAP replicas in-between two batches of queries.

8 CONCLUSION

This paper presents VEGITO, a distributed in-memory HTAP system that retrofits high availability mechanism to meet two overarching goals simultaneously—performance (e.g., 10% performance degradation) and freshness (e.g., a maximum delay of 20 ms). Evaluations using CH-benCHmark show the efficacy of VEGITO for HTAP workloads even facing millions of concurrent transactions per second.

9 ACKNOWLEDGMENT

We sincerely thank our shepherd Dushyanth Narayanan and the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 61772335, 61925206), and the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

REFERENCES

[1] Amazon Aurora FAQs: High Availability and Replication. https://aws.amazon.com/rds/aurora/faqs/.

[2] AWS Glue. https://aws.amazon.com/glue/.

[3] Hive Transactions. https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.3.0/bk_dataintegration/content/hive-013-feature-transactions.html.

[4] MemSQL. http://memsql.com/.

[5] Microsoft Azure. https://docs.microsoft.com/en-us/azure/.

[6] MonetDB. http://www.monetdb.org/.

[7] SQL Server 2019. https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[8] TiDB. https://pingcap.com/.

[9] TiFlash Overview. https://pingcap.com/docs/stable/reference/tiflash/overview/.

[10] AGRAWAL, N., AND VULIMIRI, A. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, p. 647–664.

[11] ALAGIANNIS, I., IDREOS, S., AND AILAMAKI, A. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14, p. 1103–1114.

[12] ALIBABA CLOUD. Double 11 Real-Time Monitoring System with Time Series Database. https://www.alibabacloud.com/blog/594855, 2019.

[13] APPUSWAMY, R., KARPATHIOTAKIS, M., PAROBIC, D., AND AILAMAKI, A. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research* (2017), CIDR '17.

[14] ARULRAJ, J., PAVLO, A., AND MENON, P. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 583–598.

[15] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed Data Structures over A Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 325–340.

[16] BARBER, R., GARCIA-ARELLANO, C., GROSMAN, R., LOHMAN, G., MOHAN, C., MULLER, R., PIRAHESH, H., RAMAN, V., SIDLE, R., STORM, A., ET AL. WiSer: A Highly Available HTAP DBMS for IoT Applications. In *2019 IEEE International Conference on Big Data (Big Data)* (2019).

[17] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An Introduction to Bε-trees and Write-Optimization. *login; magazine 40*, 5 (2015), 22–28.

[18] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.

[19] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion Concurrency Control Theory and Algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465–483.

[20] BESTA, M., AND HOEFLER, T. Accelerating Irregular Computations with Hardware Transactional Memory and Active Messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), HPDC'15, pp. 161–172.

[21] BINGMANN, T. STX B+ Tree C++ Template Classes. https://panthema.net/2007/stx-btree/, 2013.

[22] BOISSIER, M. Reducing the Footprint of Main Memory HTAP Systems: Removing, Compressing, Tiering, and Ignoring Data. In *Proceedings of the VLDB 2018 PhD Workshop* (2018).

[23] BONCZ, P. A., ZUKOWSKI, M., AND NES, N. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2nd Conference on Innovative Data Systems Research* (2005), vol. 5 of *CIDR '05*, pp. 225–237.

[24] CAO, S., YANG, X., CHEN, C., ZHOU, J., LI, X., AND QI, Y. TitAnt: Online Real-Time Transaction Fraud Detection in Ant Financial. *Proceedings of the VLDB Endowment 12*, 12 (Aug. 2019), 2082—-2093.

[25] CAO, Y., CHEN, C., GUO, F., JIANG, D., LIN, Y., OOI, B., VO, H., WU, S., AND XU, Q. ES2: A cloud data storage system for supporting both OLTP and OLAP. pp. 291–302.

[26] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and General Distributed Transactions using RDMA and HTM. In *Proceedings of the European Conference on Computer Systems* (2016), EuroSys'16, p. 26.

[27] CHISHOLM, S. Adopting medical technologies and diagnostics recommended by NICE: The Health Technologies Adoption Programme, 2014.

[28] COELHO, F., PAULO, J. A., VILAÇA, R., PEREIRA, J., AND OLIVEIRA, R. HTAPBench: Hybrid Transactional and Analytical Processing Benchmark. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (2017), ICPE '17, pp. 293—-304.

[29] COLE, R., FUNKE, F., GIAKOUMAKIS, L., GUY, W., KEMPER, A., KROMPASS, S., KUNO, H., NAMBIAR, R., NEUMANN, T., POESS, M., ET AL. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (2011), p. 8.

[30] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013).

[31] COSTEA, A., IONESCU, A., RĂDUCANU, B., SWITAKOWSKI, M., BÂRCA, C., SOMPOLSKI, J., UNDEFINEDUSZCZAK, A., SZAFRAUNDEFINEDSKI, M., DE NIJS, G., AND BONCZ, P. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, pp. 1105—-1117.

[32] DAGEVILLE, B., CRUANES, T., ZUKOWSKI, M., ANTONOV, V., AVANES, A., BOCK, J., CLAYBAUGH, J., ENGOVATOV, D., HENTSCHEL, M., HUANG, J., LEE, A. W., MOTIVALA, A., MUNIR, A. Q., PELLEY, S., POVINEC, P., RAHN, G., TRIANTAFYLLIS, S., AND UNTERBRUNNER, P. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, p. 215–226.

[33] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13, pp. 1243–1254.

[34] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, pp. 54–70.

[35] DRESELER, M., BOISSIER, M., RABL, T., AND UFLACKER, M. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment 13*, 10 (2020), 1206–1220.

[36] DRESELER, M., KOSSMANN, J., BOISSIER, M., KLAUCK, S., UFLACKER, M., AND PLATTNER, H. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proceedings of the 22nd International Conference on Extending Database Technology* (2019), pp. 313–324.

[37] DZIEDZIC, A., WANG, J., DAS, S., DING, B., NARASAYYA, V. R., AND SYAMALA, M. Column-store and B+ tree - Are Hybrid Physical Designs Important? In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 177–190.

[38] ESSERTEL, G., TAHBOUB, R., DECKER, J., BROWN, K., OLUKOTUN, K., AND ROMPF, T. Flare: Optimizing Apache Spark with Native Compilation for Scale-up Architectures and Medium-Size Data. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018), pp. 799–815.

[39] GU, J., YU, Q., WANG, X., WANG, Z., ZANG, B., GUAN, H., AND CHEN, H. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of 2019 USENIX Annual Technical Conference* (2019), pp. 913–928.

[40] GUPTA, A., AGARWAL, D., TAN, D., KULESZA, J., PATHAK, R., STEFANI, S., AND SRINIVASAN, V. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), pp. 1917–1923.

[41] HERLIHY, M., AND MOSS, J. E. B. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), ISCA'93, pp. 289–300.

[42] HONG, C., ZHOU, D., YANG, M., KUO, C., ZHANG, L., AND ZHOU, L. KuaFu: Closing the parallelism gap in database replication. In *Proceedings of 2013 IEEE 29th International Conference on Data Engineering* (2013), pp. 1186–1195.

[43] HUANG, D., LIU, Q., CUI, Q., FANG, Z., MA, X., XU, F., SHEN, L., TANG, L., ZHOU, Y., HUANG, M., WEI, W., LIU, C., ZHANG, J., LI, J., WU, X., SONG, L., SUN, R., YU, S., ZHAO, L., CAMERON, N., PEI, L., AND TANG, X. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow. 13*, 12 (Aug. 2020), 3072–3084.

[44] IDREOS, S., GROFFEN, F., NES, N., MANEGOLD, S., MULLENDER, K. S., AND KERSTEN, M. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull. 35* (2012), 40–45.

[45] JOHN PIEKOS. Measuring real-time. https://www.infoworld.com/article/3220430/measuring-real-time.html, 2017.

[46] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI'16, pp. 185–201.

[47] KAUFMANN, M., MANJILI, A. A., VAGENAS, P., FISCHER, P. M., KOSSMANN, D., FÄRBER, F., AND MAY, N. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1173–1184.

[48] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of 2011 IEEE 27th International Conference on Data Engineering* (2011), pp. 195–206.

[49] KEMPER, A., NEUMANN, T., FUNKE, F., LEIS, V., AND MÜHE, H. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Eng. Bull. 35*, 1 (2012), 46–51.

[50] KIM, K., WANG, T., JOHNSON, R., AND PANDIS, I. Ermia: Fast Memory-optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1675–1687.

[51] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[52] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical Paxos and Primary-backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (2009), PODC'09, pp. 312–313.

[53] LARSON, P.-Å., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow. 5*, 4 (2011), 298–309.

[54] LARSON, P.-R., BIRKA, A., HANSON, E. N., HUANG, W., NOWAKIEWICZ, M., AND PAPADIMOS, V. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1740–1751.

[55] LEE, J., MOON, S., KIM, K. H., KIM, D. H., CHA, S. K., AND HAN, W.-S. Parallel Replication Across Formats in SAP HANA for Scaling out Mixed OLTP/OLAP Workloads. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1598–1609.

[56] LI, F., ÖZSU, M. T., CHEN, G., AND OOI, B. C. R-store: A Scalable Distributed System for Supporting Real-time Analytics. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering* (2014), pp. 40–51.

[57] LI, J., MICHAEL, E., AND PORTS, D. R. K. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 104–120.

[58] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably Fast Multi-core In-memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 21–35.

[59] LOCKERMAN, J., FALEIRO, J. M., KIM, J., SANKARAN, S., ABADI, D. J., ASPNES, J., SEN, S., AND BALAKRISHNAN, M. The FuzzyLog: A Partially Ordered Shared Log. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018), OSDI'18, pp. 357—-372.

[60] MAKRESHANSKI, D., GICEVA, J., BARTHELS, C., AND ALONSO, G. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 37–50.

[61] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys'12, pp. 183–196.

[62] MARTIN, D., KOETH, O., KERN, J., AND IVANOVA, I. Near Real-Time Analytics with IBM DB2 Analytics Accelerator. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), EDBT '13, pp. 579–588.

[63] MARY SHACKLETT. See real-time big data analytics in milliseconds with IMDG technology. https://www.techrepublic.com/article/see-real-time-big-data-analytics-in-milliseconds-with-imdg-technology/, 2014.

[64] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[65] MÜHLBAUER, T., RÖDIGER, W., REISER, A., KEMPER, A., AND NEUMANN, T. ScyPer: Elastic OLAP Throughput on Transactional Data. In *Proceedings of the Second Workshop on Data Analytics in the Cloud* (2013), DanaC '13, p. 11–15.

[66] MYSQL. MySQL 8.0 Reference Manual: Chapter 17 Replication. https://dev.mysql.com/doc/refman/8.0/en/replication-semisync.html.

[67] NEUMANN, T., AND FREITAG, M. J. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the 10th Conference on Innovative Data Systems Research* (2020), CIDR '20.

[68] NEUMANN, T., MÜHLBAUER, T., AND KEMPER, A. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 677–689.

[69] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 29–41.

[70] ÖZCAN, F., TIAN, Y., AND TÖZÜN, P. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1771–1775.

[71] PAREEK, A., KHALADKAR, B., SEN, R., ONAT, B., NADIMPALLI, V., AGARWAL, M., AND KEENE, N. Striim: A Streaming Analytics Platform for Real-time Business Decisions. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (2017), BIRTE '17, pp. 4:1–4:8.

[72] PAREEK, A., KHALADKAR, B., SEN, R., ONAT, B., NADIMPALLI, V., AND LAKSHMINARAYANAN, M. Real-time ETL in Striim. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (2018), BIRTE '18, pp. 3:1–3:10.

[73] PAVLO, A., ANGULO, G., ARULRAJ, J., LIN, H., LIN, J., MA, L., MENON, P., MOWRY, T. C., PERRON, M., QUAH, I., ET AL. Self-Driving Database Management Systems. In *Proceedings of the 8th Conference on Innovative Data Systems Research* (2017), CIDR '17.

[74] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1816–1827.

[75] PEZZINI, M., FEINBERG, D., RAYNER, N., AND EDJLALI, R. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. *Gartner* (2014).

[76] PUBNUB. How Fast is Realtime? Human Perception and Technology. https://www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/, 2015.

[77] QIU, X., CEN, W., QIAN, Z., PENG, Y., ZHANG, Y., LIN, X., AND ZHOU, J. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow. 11*, 12 (2018), 1876–1888.

[78] QUAH, J. T., AND SRIGANESH, M. Real-time Credit Card Fraud Detection Using Computational Intelligence. *Expert systems with applications 35*, 4 (2008), 1721–1732.

[79] RAMAN, V., ATTALURI, G., BARBER, R., CHAINANI, N., KALMUK, D., KULANDAISAMY, V., LEENSTRA, J., LIGHTSTONE, S., LIU, S., LOHMAN, G. M., ET AL. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow. 6*, 11 (2013), 1080–1091.

[80] RAMNARAYAN, J., MOZAFARI, B., WALE, S., MENON, S., KUMAR, N., BHANAWAT, H., CHAKRABORTY, S., MAHAJAN, Y., MISHRA, R., AND BACHHAV, K. SnappyData: A Hybrid Transactional Analytical Store Built On Spark. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, pp. 2153—-2156.

[81] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating Mapreduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007), pp. 13–24.

[82] RAZA, A., CHRYSOGELOS, P., ANADIOTIS, A. C., AND AILAMAKI, A. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), SIGMOD '20, pp. 2043—2054.

[83] RÖDIGER, W., MÜHLBAUER, T., KEMPER, A., AND NEUMANN, T. High-speed Query Processing over High-speed Networks. *Proc. VLDB Endow. 9*, 4 (2015), 228–239.

[84] SADOGHI, M., BHATTACHERJEE, S., BHATTACHARJEE, B., AND CANIM, M. L-Store: A Real-time OLTP and OLAP System. In *Proceedings of the 21th International Conference on Extending Database Technology* (2018), EBDT '18.

[85] SAHAY, B., AND RANJAN, J. Real Time Business Intelligence in Supply Chain Analytics. *Information Management & Computer Security 16*, 1 (2008), 28–48.

[86] SEWALL, J., CHHUGANI, J., KIM, C., SATISH, N., AND DUBEY, P. PALM: Parallel Architecture-friendly Latch-free Modifications to B+ trees on Many-core Processors. *Proc. VLDB Endowment 4*, 11 (2011), 795–806.

[87] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data* (2019), SIGMOD '19, p. 433–448.

[88] SHARMA, A., SCHUHKNECHT, F. M., AND DITTRICH, J. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 245–258.

[89] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI '16, pp. 317–332.

[90] SIKKA, V., FÄRBER, F., LEHNER, W., CHA, S. K., PEH, T., AND BORNHÖVD, C. 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), SIGMOD '12, pp. 731–742.

[91] SOMPOLSKI, J., ZUKOWSKI, M., AND BONCZ, P. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware* (2011), pp. 33–40.

[92] STEVE ABRAHAM. Creating a proof of concept using Amazon Aurora. https://aws.amazon.com/blogs/database/creating-a-proof-of-concept-using-amazon-aurora/, 2019.

[93] TA, V.-D., LIU, C.-M., AND NKABINDE, G. W. Big Data Stream Computing in Healthcare Real-time Analytics. In *Proceedings of the 2016 IEEE International Conference on Cloud Computing and Big Data Analysis* (2016), pp. 37–42.

[94] TAI, A., WEI, M., FREEDMAN, M. J., ABRAHAM, I., AND MALKHI, D. Replex: A Scalable, Highly Available Multi-index Data Store. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 337–350.

[95] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[96] THE TRANSACTION PROCESSING COUNCIL. TPC-H Benchmark V2.17.3. http://www.tpc.org/tpch/.

[97] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD'12, pp. 1–12.

[98] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13.

[99] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation* (2004), vol. 4 of *OSDI '04*, pp. 91–104.

[100] VASSILIADIS, P., AND SIMITSIS, A. Near real time ETL. In *New Trends in Data Warehousing and Data Analysis*. 2009, pp. 1–31.

[101] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, p. 1041–1052.

[102] WAN, Y. S. G. E. B., LIM, S., AND PAVLO, A. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes. *Proc. VLDB Endow. 13*, 2 (2019).

[103] WANG, T., JOHNSON, R., AND PANDIS, I. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow. 11*, 4 (2017), 406–419.

[104] WANG, X., ZHANG, W., WANG, Z., WEI, Z., CHEN, H., AND ZHAO, W. Eunomia: Scaling Concurrent Search Trees under Contention Using HTM. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), PPoPP '17, p. 385–399.

[105] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using Restricted Transactional Memory to Build a Scalable In-memory Database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, pp. 26:1–26:15.

[106] WEI, X., CHEN, R., CHEN, H., WANG, Z., GONG, Z., AND ZANG, B. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *Proceedings of 18th USENIX Symposium on Networked Systems Design and Implementation* (Apr. 2021).

[107] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18, pp. 233–251.

[108] WEI, X., SHEN, S., CHEN, R., AND CHEN, H. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), USENIX ATC '17, pp. 335–347.

[109] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, pp. 87–104.

[110] WU, Y., ARULRAJ, J., LIN, J., XIAN, R., AND PAVLO, A. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. VLDB Endow. 10*, 7 (Mar. 2017), 781–792.

[111] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving Graph Traversal with Split Live Migration. In *Proceedings of the 2019 USENIX Annual Technical Conference* (2019), USENIX ATC '19, pp. 723–738.

[112] YANG, J., RAE, I., XU, J., SHUTE, J., YUAN, Z., LAU, K., ZENG, Q., ZHAO, X., MA, J., CHEN, Z., GAO, Y., DONG, Q., ZHOU, J., WOOD, J., GRAEFE, G., NAUGHTON, J., AND CIESLEWICZ, J. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow. 13*, 12 (Aug. 2020), 3313–3325.

[113] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.

[114] ZHANG, H., ANDERSEN, D. G., PAVLO, A., KAMINSKY, M., MA, L., AND SHEN, R. Reducing the Storage Overhead of Main-memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1567–1581.

[115] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-Millisecond Stateful Stream Querying over Fast-Evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, p. 614–630.

[116] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 465–477.

[117] ZHOU, J., LI, X., ZHAO, P., CHEN, C., LI, L., YANG, X., CUI, Q., YU, J., CHEN, X., DING, Y., AND QI, Y. A. KunPeng: Parameter Server Based Distributed Learning Systems and Its Applications in Alibaba and Ant Financial. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), KDD '17, pp. 1693–1702.

A ARTIFACT APPENDIX

Abstract

This artifact provides the prototype of VEGITO, including the document, source code and scripts to execute the main experiments and reproduce the experimental results. VEGITO is a fast distributed in-memory HTAP system, which retrofits high availability mechanism to tame hybrid transaction/analytical processing. An open-source version of VEGITO is available at https://github.com/SJTU-IPADS/vegito.

Scope

This artifact (including the document, source code and scripts) is used for artifact evaluation, which can reproduce the main experimental results in VEGITO. To use VEGITO in your research, we recommend using the `master` branch of the public repository, which would be maintained by members of the Institute of Parallel and Distributed Systems.

Contents

- **README and document:** A detailed description of the artifacts, including the steps of environment building, installation, usage of scripts and configuration files, and how to conduct experiments. Please read the `README.md` at first.
- **Source code:** We provide the prototype of VEGITO with the HTAP benchmark called CH-benCHmark and some micro-benchmarks.
- **Configuration files:** We record different configurations in some XML format files. The detailed format is described in the `README.md`.
- **Scripts:** We run the VEGITO by using the Python scripts and Shell scripts. These scripts use SSH for cluster deployment and management.

Hosting

- **Program:** `vegito`.
- **Compilation:** `g++` and `cmake`.
- **Hardware:** Intel CPU with RTM and Mellanox NIC with RDMA.
- **Execution:** Python scripts, Shell scripts, SSH.
- **Metrics:** Throughput, latency, and time lag (freshness).
- **Public link:** https://github.com/SJTU-IPADS/vegito.
- **Code licenses:** Apache License 2.0.

Requirements

Hardware Dependencies. At least three machines are used to reproduce the experimental results for distributed configurations. Each machine must have:

- **CPU:** Intel processors with 2 sockets and Restricted Transactional Memory (RTM) (e.g., Xeon E5-2650 v4).
- **NIC:** At least one (two is better) Mellanox RDMA network card (e.g., Mellanox ConnectX-4 100Gbps InfiniBand NIC).

Software Dependencies.

- **Operating system:** `Ubuntu` $\geq$ `16.04`.
- **Compile toolchain:** `g++` $\geq$ `5.4.4` and `cmake` $\geq$ `3.5.1`.
- **Libraries:** `Mellanox OFED`, `boost 1.61.0`, `ssmalloc`.

AE Methodology

Submission, reviewing and badging methodology is specified at https://www.usenix.org/conference/osdi21/call-for-artifacts.

The nanoPU: A Nanosecond Network Stack for Datacenters

Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen,
Muhammad Shahbaz*, Changhoon Kim, and Nick McKeown
Stanford University **Purdue University*

Abstract

We present the nanoPU, a new NIC-CPU co-design to accelerate an increasingly pervasive class of datacenter applications: those that utilize many small Remote Procedure Calls (RPCs) with very short (μs-scale) processing times. The novel aspect of the nanoPU is the design of a *fast path* between the network and applications—bypassing the cache and memory hierarchy, and placing arriving messages directly into the CPU register file. This fast path contains programmable hardware support for low latency transport and congestion control as well as hardware support for efficient load balancing of RPCs to cores. A hardware-accelerated thread scheduler makes sub-nanosecond decisions, leading to high CPU utilization and low tail response time for RPCs.

We built an FPGA prototype of the nanoPU fast path by modifying an open-source RISC-V CPU, and evaluated its performance using cycle-accurate simulations on AWS FPGAs. The wire-to-wire RPC response time through the nanoPU is just 69ns, an order of magnitude quicker than the best-of-breed, low latency, commercial NICs. We demonstrate that the hardware thread scheduler is able to lower RPC tail response time by about 5× while enabling the system to sustain 20% higher load, relative to traditional thread scheduling techniques. We implement and evaluate a suite of applications, including MICA, Raft and Set Algebra for document retrieval; and we demonstrate that the nanoPU can be used as a high performance, programmable alternative for one-sided RDMA operations.

1 Introduction

Today, large online services are typically deployed as multiple tiers of software running in data centers. Tiers communicate with each other using Remote Procedure Calls (RPCs) of varying size and complexity [7, 28, 57]. Some RPCs call upon microservices lasting many milliseconds, while others call remote (serverless) functions, or retrieve a single piece of data and last only a few *micro*seconds. These are important workloads, and so it seems feasible that small messages with microsecond (and possibly nanosecond) service times will become more common in future data centers [7, 28]. For example, it is reported that a large fraction of messages communicated in Facebook data centers are for a single key-value memory reference [4, 7], and a growing number of papers describe fine-grained (typically cache-resident) computation based on very small RPCs [22, 23, 28, 57].

Three main metrics are useful when evaluating an RPC system's performance: (1) the *median response time* (i.e., time from when a client issues an RPC request until it receives a response) for applications invoking many sequential RPCs; (2) the *tail response time* (i.e., the longest or 99th %ile RPC response time) for applications with large fanouts (e.g., map-reduce jobs), because they must wait for all RPCs to complete before continuing [17]; and (3) the *communication overhead* (i.e., the communication-to-computation ratio). When communication overhead is high, it may not be worth farming out the request to a remote CPU at all [57]. We will sometimes need more specific metrics for portions of the processing pipeline, such as the *median wire-to-wire latency*, the time from when the first bit of an RPC request arrives at the server NIC until the last bit of the response departs.

Many authors have proposed exciting ways to accelerate RPCs by reducing the message processing overhead. These include specialized networking stacks, both in software (e.g., DPDK [18], ZygOS [51], Shinjuku [27], and Shenango [49]), and hardware (e.g., RSS [43], RDMA [9], Tonic [2], NeBuLa [57], and Optimus Prime [50]). Each proposal tackles one or more components of the RPC stack (i.e., network transport, congestion control, core selection, thread scheduling, and data marshalling). For example, DPDK removes the memory copying and network transport overhead of an OS and lets a developer handle them manually in user space. ZygOS implements a scheme to efficiently load balance messages across multiple cores. Shenango efficiently shares CPUs among services requiring RPC messages to be processed. eRPC [28] cleverly combines many software techniques to reduce median RPC response times by optimizing for the common case (i.e., small messages with short RPC handlers). These systems have successfully reduced the message-processing overhead from 100s of microseconds to 1–2 microseconds.

NeBuLa [57] is a radical hardware design that tries to further minimize response time by integrating the NIC with the CPU (bypassing PCIe), and dispatching RPC requests *directly into the L1 cache*. The approach effectively reduces the minimum wire-to-wire response time below 100ns.

Put another way, these results suggest that with the right hardware and software optimizations, it is practical and useful to remotely dispatch functions as small as a few microseconds. The goal of our work is to enable even smaller functions, with computation lasting less than 1μs, for which we need to minimize communication overhead. We call these very short RPCs *nanoRequests*.

The nanoPU, presented and evaluated here, is a combined NIC-CPU optimized to process nanoRequests very quickly. When designing nanoPU, we set out to answer two questions.

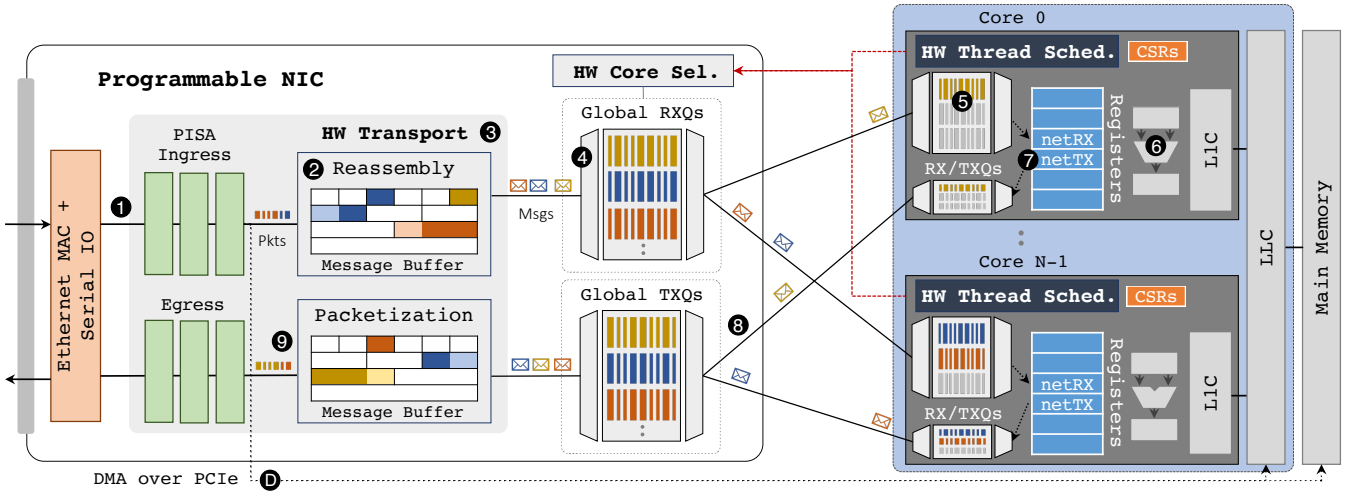


Figure 1: The nanoPU design. The NIC includes ingress and egress PISA pipelines as well as a hardware-terminated transport and a core selector with global RX queues; each CPU core is augmented with a hardware thread scheduler and local RX/TX queues connected directly to the register file.

The first is, **what is the absolute minimum communication overhead we can achieve for processing nanoRequests?** NanoRequests are simply very short-lived RPCs marked by the client and server NICs for special treatment. In nanoPU, nanoRequests follow a new low-overhead path through the NIC, bypassing the OS and the memory-cache hierarchy and arriving directly into running threads' registers. All message reassembly functions, transport and congestion control logic are moved to hardware, as are thread scheduling and core selection decisions. Incoming nanoRequests pass through only hardware before reaching application code. Our nanoPU prototype can deliver an arriving nanoRequest into a running application thread in less than 40ns (less than 15ns if we bypass the Ethernet MAC)—an order of magnitude faster than the fastest commercial NICs [20] and faster than the quickest reported research prototype [57]. For compatibility with existing applications, nanoPU allows all other network traffic (e.g., larger RPCs) to traverse a regular path through a DMA NIC, OS, and memory hierarchy.

Our second question is, **can we minimize tail response time by processing nanoRequests in a deterministic amount of time?** The answer is a qualified yes. Because nanoRequests are processed by a fixed-latency hardware pipeline, if a single-packet request arrives at a waiting core, its thread will always start processing the message in less than 40ns. On the other hand, if the core is busy, or another request is queued ahead, then processing can be delayed. In Section 2.2, we show how our novel hardware thread scheduler can bound the tail response time in this case too, under specific assumptions (e.g., that a nanoRequest can bound its CPU processing time, else its priority is downgraded). We believe nanoPU is the first system to bound the response time of short-lived requests.

In summary, the main contributions of the nanoPU are:

1. The nanoPU's median *wire-to-wire* response time for nanoRequests, from the wire through the header-processing pipeline, transport layer, core selection, and thread scheduling, plus a simple loopback application and back to the wire is just 69ns, an order of magnitude lower latency than the best commercial NICs [20]. Without the MAC and serial I/O, loopback latency is only 17ns.
2. Our prototype's hardware thread scheduler continuously monitors processing status for nanoRequests and makes decisions in less than 1ns. The nanoPU sustains 20% higher load than existing approaches, while maintaining close to 1μs 99th %ile tail response times.
3. Our complete RISC-V based prototype is available open-source,[1] and runs on AWS F1 FPGAs using Firesim [31].
4. We evaluate a suite of applications including: the MICA key-value store [38], Raft consensus [47], set algebra and high dimensional search inspired from the μ-Suite benchmark [56].
5. We demonstrate that the nanoPU can be used to implement one-sided RDMA operations with lower latency and more flexibility than state-of-the-art commercial RDMA NICs.

The nanoPU ideas could be deployed in a variety of ways: by adding the low latency path to a conventional CPU, or by designing new RPC-optimized CPUs with only the low-latency path, or by adding the new path to embedded CPUs on smartNICs.

2 The nanoPU Design

The nanoPU is a new NIC-CPU co-design that adds a new fast path for nanoRequest messages requiring ultra-low and predictable network communication latency. Figure 1 depicts

[1] nanoPU Artifact: https://github.com/l-nic/chipyard/wiki

the key design components. The nanoPU has two independent network paths: (1) the traditional (unmodified) DMA path to/from the host's last-level [16] or L1 cache [57], and (2) an accelerated fast path for nanoRequests, directly into the CPU register file.

The traditional path can be any existing path through hardware and software; hence all network applications can run on the traditional path of the nanoPU unchanged, and perform at least as well as they do today. The fast path is a nanosecond-scale network stack optimized for nanoRequests. Applications should (ideally) be optimized to efficiently process nanoRequest messages directly out of the register file to fully harness the benefits of the fast path.

Each core has its own hardware thread scheduler (HTS), two small FIFO memories for network ingress and egress data, and two reserved general-purpose registers (GPRs): one as the tail of the egress FIFO for sending nanoRequest data, and the other as the head of the ingress FIFO for receiving. CPU cores are statically partitioned into two groups: those running normal applications and those running nanoRequest applications. Cores running regular applications use standard OS software thread scheduling [27, 49, 51]; however, the OS delegates scheduling of nanoRequest threads to HTS.

To understand the flow of the nanoPU fast path, consider the numbered steps in Figure 1. In ❶, a packet arrives and enters the P4-programmable PISA pipeline. In addition to standard header processing (e.g., matching IP addresses, checking version and checksum, and removing tunnel encapsulations), the pipeline examines the destination layer-4 port number in the transport header using a match-action table[2] to decide if the message should be delivered along the fast path. If so, it proceeds to ❷, else it follows the usual DMA processing path Ⓓ. In ❷, packets are reassembled into messages; a buffer is allocated for the entire message and packet data is (potentially) re-sequenced into the correct order. In ❸, the transport protocol ensures reliable message arrival; until all data has arrived, message data and signaling packets are exchanged with the peer depending on the protocol (e.g., NDP and Homa are both receiver driven using different grant mechanisms) (Section 2.3). When a message has arrived, in ❹ it is placed in a per-application receive queue where it waits to be assigned to a core by the core-selection logic (Section 2.3). When its turn comes, in ❺, the message is sent to the appropriate per-thread ingress FIFO on the assigned core, where it waits for HTS (Section 2.2) to alert the core to run the message's thread and place the first word in the `netRX` register (Section 2.1). In ❻, the core processes the data and, if running a server application, will typically generate a response message for the client. The application transmits a message by issuing instructions that write one "word" at a time to the `netTX` register in ❼, where the word size is defined by the size of a CPU register, typically 64-bits (8B). These message words then flow into the global transmit queues in ❽. Messages are split into packets in ❾, before departing through the egress PISA pipeline.

Next, we detail the design of the main, novel components of the fast path: the thread-safe register file network interface, the hardware thread scheduler (HTS), and the programmable NIC pipeline, including transport and core selection.

2.1 Thread-Safe Register File Interface

Recent work [45] showed that PCIe latency contributes about 90% of the median wire-to-wire response time for small packets (800–900ns). Several authors have proposed integrating the NIC with the CPU, to bring packets directly into the cache [12, 46, 57].

The nanoPU takes this one step further and connects the network fast path directly to the CPU core's register file. The high-level idea is to allow applications to send and receive network messages by writing/reading one word (8B) at a time to/from a pair of dedicated CPU registers.

There are several advantages to bringing packet data directly into the register file:

Message data bypasses the memory and cache hierarchy, minimizing the time from when a packet arrives on the wire until it is available for processing. In Section 5.2.1, we show that this reduces median wire-to-wire response time to 69ns, 50% lower than the state-of-the-art.

Reduces variability in processing time and therefore minimizes tail response time. For example, there is no variable waiting time to cross PCIe, no cache misses for message data (messages do not enter or leave through memory) and no IO-TLB misses (which lead to an expensive 300ns access to the page table [45]). And because nanoRequests are buffered in dedicated FIFOs, separate from the cache, nanoRequest data does not compete for cache space with other application data, further reducing cache misses for applications. Cache misses can be expensive: A LLC miss takes about 50-100ns to resolve and creates extra traffic on the (shared) DRAM memory bus. DRAM access can be a bottleneck for a multicore CPU, and when congested, memory access times can increase by more than 200% [60]. Furthermore, contention for cache space and DRAM bandwidth is worse at network speeds above 100Gb/s [21].

Less software overhead per message because software does not need to manage DMA buffers or perform memory-mapped IO (MMIO) handshakes with the NIC. In a conventional NIC, when an application sends a message, the OS first places the message into a DMA buffer and passes a message descriptor to the NIC. The NIC interrupts or otherwise notifies software when transmission completes, and software must step in again to reclaim the DMA buffer. The register file message interface has much lower overhead: When an application thread sends a message it simply writes the message directly into the `netTX` register, with no additional work. Section 5.2.1 shows how this leads to a much higher throughput interface.

[2] It is the responsibility of the the host software to configure this table with entries for all nanoRequest processing applications.

2.1.1 How an application uses the interface

The J-Machine [13] first used the register file in 1989 for very low latency inter-core communication, followed by the Cray T3D [33]. The approach was abandoned because it proved difficult to protect messages from being read/written by other threads sharing the same core; both machines required atomic message reads and writes [14]. As we show below, our design solves this problem. We believe ours is the first design to add a register file interface to a regular CPU for use in data centers.

The nanoPU reserves two general-purpose registers (GPRs) in the register file for network IO, which we call `netRX` and `netTX`. When an application issues an instruction that reads from `netRX`, it actually reads a message word from the head of the network receive queue. Similarly, when an application issues an instruction that writes to `netTX`, it actually writes a message word to the tail of the network transmit queue. The network receive and transmit queues are stored in small FIFO memories that are connected directly to the register file.[3] In addition to the reserved GPRs, a small set of control & status registers (CSRs, described in Section 3.4) are used for the core and NIC hardware to coordinate with each other.

Delimiting messages. Each message that is transmitted and received by an application begins with a fixed 8B "application header". On arriving messages, this header indicates the message length (as well as the source IP address and layer-4 port number), which allows software to identify the end of the message. Similarly, the application header on departing messages contains the message length (along with the destination IP address and layer-4 port number) so that the NIC can detect the end of the outgoing message. The programmable NIC pipeline replaces the application header with the appropriate Ethernet, IP, and transport headers on all transmitted packets.

Inherent thread safety. We need to prevent an errant thread from reading or writing another thread's messages. The nanoPU prevents this using a novel hardware interlock. It maintains a separate ingress and egress FIFO for each thread, and controls access to the FIFOs so that `netRX` and `netTX` are always mapped to the head and tail, respectively, of the FIFOs for the currently running thread only. Note our hardware design ensures this property even when a previous thread does not consume or finish writing a complete message.[4] This turned out to be a key design choice, simplifying application development on the nanoPU; nanoRequest threads no longer need to read and write messages atomically.

Software changes. The register file can be accessed in one CPU cycle, while the L1 cache typically takes three cycles.

[3] We think of these FIFO memories as equivalent to an L1 cache, but for network messages; both are built into the CPU pipeline and sit right next to the register file.

[4] Our interlock logic would have been prohibitively expensive in the early days; but since 1989, Moore's Law lets us put four orders of magnitude more gates on a chip, making the logic quite manageable (Section 5).

Application	Description	Response Time p50 / p99 (μs)
MICA	Implements a fast in-memory key-value store	0.40 / 0.50
Raft	Runs leader-based state machine replication	3.08 / 3.26 *
Chain Repl.	Runs a vertical Paxos consensus algorithm	1.10 / 1.40 *
Set Algebra	Processes data-mining and text-analytics workloads	0.60 / 1.50
HD Search	Analyzes and processes image, video, and speech data	0.80 / 1.20
N-Body Sim.	Computes gravitational force for simulated bodies	0.35 / N/A
INT Processing	Processes network telemetry data (e.g., path latency)	0.13 / N/A
Packet Classifier	Classifies packets for intrusion detection (100K rules)	0.90 / 2.20
Othello Player	Searches the Othello state space	0.90 / 1.70 [26]
One-sided RDMA	Performs one-sided RDMA operations in software	0.68 / N/A *

Table 1: Example applications that have been implemented on the nanoPU. These applications use small network messages, few memory references, and cache-resident function stack and variables (in the common case), and are designed to efficiently process messages out of the register file. Table indicates median and 99th %ile wire-to-wire response time at low load. *Measured at client.

Therefore, an application thread will run faster if it can process data directly from the ingress FIFO by serially reading `netRX`. Ideally, the developer picks a message data structure with data arranged in the order it will be consumed—we did this for the message processing components of the applications evaluated in Section 5.3. If an application needs to copy long messages entirely into memory so that it can randomly access each byte many times during processing, then the register file interface may not offer much advantage over the regular DMA path. Our experience so far is that, with a little practice, it is practical to port latency-sensitive applications to efficiently use the nanoPU register file interface. Table 1 lists applications that have been ported to efficiently use this new network interface and Section 4 further discusses applications on the nanoPU.

A related issue is how, and at which stage of processing, to serialize/deserialize (also known as marshall/unmarshall) message data. In modern RPC applications this processing is typically implemented in libraries such as Protobuf [52] or Thrift [59]. Recent work pointed out that on conventional CPUs, where network data passes through the memory hierarchy, the serialize/deserialize logic is dominated by scatter/gather memory-copy operations and subword-level data transformation operations, suggesting a separate hardware accelerator might help [50]. In the nanoPU, the memory copy overhead involved in serialization and deserialization is little

or none; only a few copies between registers and the L1 cache may be necessary when a working set is larger than the register file. The remaining subword data-transformation tasks can be done either in the applications (in software) or on the NIC (in hardware) using a PISA-like pipeline, but still operating at the message level. We currently take the former approach for the applications we evaluate in Section 5.3, but intend to explore the latter approach in future work.

2.2 Thread Scheduling in Hardware

Current best practice for low-latency applications is to either (1) pin threads to dedicated cores [18, 51], which is very inefficient when a thread is idle, or (2) devote one core to run a software thread scheduler for the other cores [27, 49].

The fastest software-based thread schedulers are not fast enough for nanoRequests. Software schedulers need to run periodically so as to avoid being overwhelmed by interrupts and associated overheads, which means deciding how frequently they should run. If it runs too often, resources are wasted; too infrequently and threads are unnecessarily delayed. The fastest state-of-the-art operating systems make periodic scheduling decisions every 5μs [27, 49], which is too coarse-grained for nanoRequests requiring only 1μs of computation.

We therefore moved the nanoRequest thread scheduler to hardware, which continuously monitors message processing status as well as the network receive queues and makes sub-nanoseconds scheduling decisions. Our new hardware thread scheduler (HTS) is both faster and more efficient; a core never sits on an idle thread when another thread with a pending message could run.

2.2.1 How the hardware thread scheduler works

Every core contains its own scheduler hardware. When a new thread initializes, it must register itself with its core's HTS by binding to a layer-4 port number and selecting a strict priority level (0 is the highest). The layer-4 port number lets the nanoPU hardware distinguish between threads and ensure that `netRX` and `netTX` are always the head and tail of the FIFOs for the currently running thread.

HTS tracks the running thread's priority and its time spent on the CPU core. When a new message arrives, if its destination thread's priority is lower than or equal to the current thread, the new message is queued. If the incoming message is for a higher priority thread, the running thread is suspended and the destination thread is swapped onto the core. Whenever HTS determines that threads must be swapped, it (1) asserts a new, NIC-specific interrupt that traps into a small software interrupt handler (only on the relevant core), and (2) tells the interrupt handler which thread to switch to by writing the target's layer-4 port number to a dedicated CSR. Our current HTS implementation takes about 50ns to swap a previously idle thread onto the core, measured from the moment its first pending message arrives (Section 3.2).

If the thread to switch to belongs to a different process, the software interrupt handler must perform additional work: notably, it must change privilege modes and swap address spaces. A typical context switch in Linux takes about 1μs [27], but most of this time is spent making the scheduling decision [62]. Our HTS design makes this decision entirely in hardware and the software scheduler simply needs to read a CSR to determine which thread to swap to.

The scheduling policy. HTS implements a *bounded strict priority* scheduling policy to ensure that the highest priority thread with pending work is running on the core at all times. Threads are marked `active` or `idle`. A thread is marked `active` if it is eligible for scheduling, which means it has been registered (a port number and RX/TX FIFOs have been allocated) and a message is waiting in the thread's RX FIFO. The thread remains `active` until it explicitly indicates that it is `idle` and its RX FIFO is empty. HTS tries to ensure that the highest priority `active` thread is always running.

Bounded response time. HTS supports a unique feature to bound how long one high-priority application can hold up another. If a priority 0 thread takes longer than t_0 to process a message, the scheduler will immediately downgrade its priority from 0 to 1, allowing it to be preempted by a different priority 0 thread with pending messages. (By default, $t_0 = 1\mu$s.) We define a *well-behaved* application as one that processes all of its messages in less than t_0.

As a consequence, HTS guarantees an upper bound on the response time for well-behaved applications. If a core is configured to run at most k priority 0 application threads, each with at most one outstanding message at a time, then the total message processing time, t_p for well-behaved applications is bounded by: $t_p \leq t_n + kt_0 + (k-1)t_c$, where t_n is the NIC latency, and t_c is the context-switch latency. In practice, this means an application developer who writes a well-behaved application can have full confidence that no other applications will delay it beyond a predetermined bound. If application writers do not wish to use the time-bounded service, they may assign all their application threads priority 1.

Writing well-behaved applications, which are able to process all messages within a short, bounded amount of time, is complicated by cache / TLB misses and CPU power management. Our approach so far has been to empirically verify that certain applications are well-behaved. However, we believe that there is substantial opportunity for future research to determine more systematic ways for developers to write well-behaved applications. One approach may be to propose modifications to the memory hierarchy in order to make access latency more predictable. Another approach may be to develop code verification tools to check whether threads meet execution time bounds. The eBPF [19] compiler, for example, is able to verify that a packet processing program will complete eventually; we believe a similar approach can be used to verify completion within a bounded amount of time.

2.3 The nanoPU NIC Pipeline

The NIC portion of the nanoPU fast path consists of two primary components: the programmable transport layer, and the core-selection algorithm. We describe each in turn.

Programmable transport layer. The nanoPU provides nanoRequest threads a *reliable one-way message* service. To be fast enough, the transport layer needs to be terminated in hardware in the NIC. For example, our prototype hardware NDP implementation (Section 3.3) runs in 7ns (fixed) per packet and at 200Gb/s for minimum size packets (64B). Such low latency means a tight congestion-control loop between end-points, and hence more efficient use of the network. Moreover, moving transport to hardware frees CPU cycles for application logic [2].

We only have space to give a high level overview of our programmable transport layer, leaving details to a follow-on paper. At the heart of our programmable transport layer is an event-driven, P4-programmable PISA pipeline [10, 25]. The pipeline can be programmed to do normal header processing, such as VXLAN, overlay tunnels, and telemetry data [35]. We enhance it for reliable message processing, including congestion control, and have programmed it to implement the NDP [24] and Homa [42] low-latency message protocols. Network operators can program custom message protocols tailored to their specific workloads.

Low-latency, message-oriented transport protocols are well-suited to hardware, compared to connection-oriented, reliable byte-stream protocols such as TCP. The NIC only needs to maintain a small amount of state for partially delivered messages. For example, our NDP implementation, beyond storing the actual message, keeps a per-message bitmap of received packets, and a few bytes for congestion control. This allows our design to be limited only by the number of outstanding messages, rather than the number of open connections, allowing large scale, highly-distributed applications across thousands of servers.

The transport layer (Figure 1) contains buffers to convert between the unreliable IP datagram domain and the reliable message domain. Outbound messages pass through a packetization buffer to split them into datagrams, which may need to be retransmitted out of order due to drops in the network. Inbound datagrams are placed into a reassembly buffer, reordering them as needed to prepare them for delivery to a CPU core.

Selecting a CPU core. If the NIC randomly sends messages to cores, some messages will inevitably sit in a queue waiting for a busy core while another core sits idle. Our NIC therefore implements a core-selection algorithm in hardware. Inspired by NeBuLa [57], our NIC load balances nanoRequest messages across cores using the Join-Bounded-Shortest-Queue or JBSQ(n) algorithm [36].

JBSQ(n) approximates an ideal, work-conserving single queue policy using a combination of a single central queue, and short bounded queues at each core, with a maximum depth of n messages. The centralized queue replenishes the shortest per-core queues first. JBSQ(1) is equivalent to the theoretically ideal single-queue model, but is impractical to implement efficiently at these speeds.

Our nanoPU prototype implements a JBSQ(2) load balancer in hardware *per application*. The NIC is connected to each core using dedicated wires, and the RX FIFOs on each core have space for at least two messages per thread running on the core. We chose JBSQ(2) based on the communication latency between the NIC and the cores as well as the available memory bandwidth for the centralized queues. We evaluate its performance in Section 5.2.3.

3 Our nanoPU Implementation

We designed a prototype quad-core nanoPU based on the open-source RISC-V Rocket core [54]. A block diagram of our prototype is shown in Figure 2.

Our prototype extends the open-source RISC-V Rocket-Chip SoC generator [3], adding 4,300 lines of Chisel [6] to the code base. The Rocket core is a simple five-stage, in-order, single-issue processor. We use the default Rocket core configuration: 16KB L1 instruction and data caches, a 512KB shared L2 cache, and 16GB of external DRAM memory. Everything shown in Figure 2, except the MAC and Serial IO, is included in our prototype and is available as an open-source, reproducible artifact.[5] Our prototype does not include the traditional DMA path between the NIC and memory hierarchy. Instead, we focus our efforts on building the nanoPU fast path for nanoRequests.

To improve simulation speed, we do not run a full operating system on our prototype, but rather just enough to boot the system, initialize one or more threads on the cores, and perform context switches between threads when instructed to do so by the hardware thread scheduler (HTS). In total, this consists of about 1,200 lines of C code and RISC-V assembly instructions. All applications run as bare-metal applications linked with the C standard library.

The nanoPU design is intended to be fabricated as an ASIC, but we use an FPGA to build the initial prototype. As we will discuss further in Section 5, our prototype runs on AWS F1 FPGA instances, using the Firesim [31] framework. Our prototype adds about 15% more logic LUTs to an otherwise unmodified RISC-V Rocket core with a traditional DMA NIC.

3.1 RISC-V Register File Network Interface

The RISC-V Rocket core required surprisingly few changes to add the nanoPU register file network interface. The main change, naturally, involves the register file read-write logic. Each core has 32 GPRs, each 64-bits wide, and we reserve two for network communication (shared by all threads). Applications must be compiled to avoid using the reserved GPRs for

[5] nanoPU Artifact: https://github.com/l-nic/chipyard/wiki

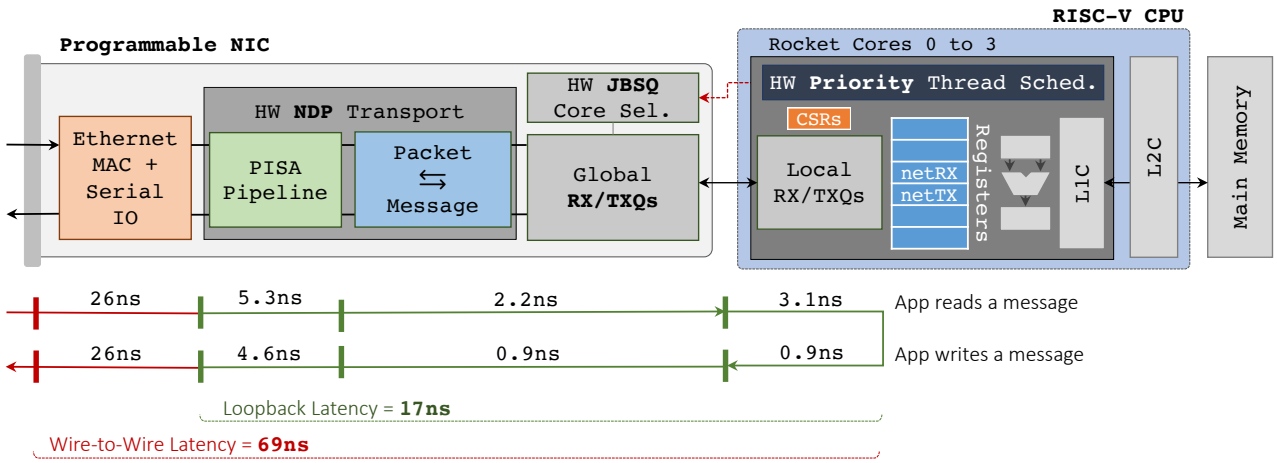


Figure 2: Our nanoPU prototype latency breakdown. Total wire-to-wire latency for an 8B message (72B packet) is 69ns.

temporary storage. Fortunately, `gcc` makes it easy to reserve registers via command-line options [48].

The core also required changes to the control logic that handles pipeline flushes. A pipeline flush can occur for a number of reasons (e.g., a branch misprediction). On a traditional five-stage RISC-V Rocket core, architectural state is not modified until an instruction reaches the write-back stage (Rocket Stage 5). However, with the addition of our network register file interface, reading `netRX` now causes a state modification (FIFO read) in the decode stage (Rocket Stage 2). The destructive read operation must be undone when there is a pipeline flush. The CPU pipeline depth is an upper bound on how many read operations need to be undone; in our case, at most two reads require undoing. It is straightforward to implement a FIFO queue supporting this operation.

3.2 Bounded Thread Scheduling in Hardware

The nanoPU core implements thread scheduling in hardware, as described in Section 2.2. The number of threads that can run on each core is primarily determined by the amount of buffering available for the local RX/TX queues. In order to implement the JBSQ(2) core selection policy, as described in Section 2.3, the local RX queue for each thread must be able to hold at least two maximum size messages. We use a maximum message size of 2KB (two packets)[6] and allocate 16KB of buffer for the local RX queues. Therefore, the prototype supports up to four threads on each core; each thread can be configured with a unique priority value. Priority 0 has a configurable maximum message processing time in order to implement the bounded priority thread scheduling policy. We added a new *thread-scheduling interrupt* to the RISC-V core, along with an accompanying control & status register (CSR) set by HTS to tell the interrupt's trap handler which thread it should run next. When processing nanoRequests, we disable all other interrupts to avoid unnecessary interrupt handling overheads.

We define the context-switch latency to be the time from when the scheduler fires the interrupt to when the first instruction of the target thread is executed. Our prototype has a measured context-switch latency of 160 cycles, or 50ns on a 3.2GHz CPU. This is much faster than a typical Linux context switch, partly because the thread scheduling decision is offloaded to hardware, and partly because the core only runs bare-metal applications in the same address space with the highest privilege mode. Therefore, nanoPU hardware thread scheduling in a Linux environment would be less efficient than our bare-metal prototype.

3.3 Prototype NIC Pipeline

The NIC portion of the nanoPU fast path consists of the programmable transport module and the core selection module. Our prototype implements both.

Transport hardware. We configured our programmable transport module to implement NDP [24] entirely in hardware. We chose NDP because it has promising low-latency performance, and is well-suited to handle small RPC messages (the class of messages we are most interested in accelerating, i.e., nanoRequests). However, the nanoPU does not depend on NDP. As explained in Section 2.3, our NIC transport layer is programmable. It has already been shown to support several other protocols, including Homa [42]. We evaluate our hardware NDP implementation in Section 5.2.3.

JBSQ hardware. As explained in Section 2.3, our NIC implements JBSQ(2) [36] to load balance messages across cores on a per-application basis. JBSQ(2) is implemented using two tables. The first maps the message's destination layer-4 port number to a per-core bitmap, indicating whether or not each core is running a thread bound to the port number. The second maps the layer-4 port number to a count of how many messages are outstanding at each core for the given port number. When a new message arrives, the algorithm checks if any of the cores that are running an application thread bound

[6] The maximum message size is a configurable parameter of the architecture and we have experimented with messages as long as 38 packets.

to the destination port are holding fewer than two of the application's messages. If so, it will immediately forward the message to the core with the smallest message count. If all target cores are holding two or more messages for this port number, the algorithm waits until one of the cores indicates that it has finished processing a message for the destination port. It then forwards the next message to that core. We evaluate our JBSQ implementation in Section 5.2.3.

3.4 The nanoPU HW/SW Interface

To illustrate how software on the nanoPU core interacts with the hardware, Listing 1 shows a simple bare-metal loopback-with-increment program in RISC-V assembly. The program continuously reads 16B messages (two 8B integers) from the network, increments the integers, and sends the messages back to their sender. The program details are described below.

The `entry` procedure binds the thread to a layer-4 port number at the given priority level by first writing a value to both the `lcurport` and `lcurpriority` CSRs, then writing the value 1 to the `lniccmd` CSR. The `lniccmd` CSR is a bit-vector used by software to send commands to the networking hardware; in this case, it is used to tell the hardware to allocate RX/TX queues both in the core and the NIC for port 0 with priority 0. The `lniccmd` CSR can also be used to unbind a port or to update the priority level.

The `wait_msg` procedure waits for a message to arrive in the core's local RX queue by polling the `lmsgsrdy` CSR until it is set by the hardware.[7] While it is waiting, the application tells HTS that it is idle by writing to the `lidle` CSR during the polling loop. The scheduler uses the idle signal to evict idle threads in order to schedule a new thread that has messages waiting to be processed.

The `loopback_plus1_16B` procedure simply swaps the source and destination addresses by moving the RX application header (the first word of every received message, see Section 2.1) from the `netRX` register to the `netTX` register, shown on line 19 (Listing 1), and thus the RX application header becomes the TX application header.[8] Upon writing the TX application header, the hardware ensures that there is sufficient buffer space for the entire message; otherwise, it generates an exception which should be handled by the application accordingly. The procedure then increments every integer in the received message and appends them to the message being transmitted. After the procedure has finished processing the message, it tells HTS it is done by writing to the `lmsgdone` CSR. The scheduler uses this write signal to: (1) reset the message processing timer for the thread, and (2) tell the NIC to dispatch the next message for this application

```
1  // Simple loopback & increment application
2  entry:
3    // Register port number & priority with NIC
4    csrwi lcurport, 0
5    csrwi lcurpriority, 0
6    csrwi lniccmd, 1
7
8  // Wait for a message to arrive
9  wait_msg:
10   csrr  a5, lmsgsrdy
11   bnez  a5, loopback_plus1_16B
12 idle:
13   csrwi lidle, 1 // app is idle
14   csrr  a5, lmsgsrdy
15   beqz  a5, idle
16
17 // Loopback and increment 16B message
18 loopback_plus1_16B:
19   mv netTX, netRX // copy app hdr: rx to tx
20   addi netTX, netRX, 1 // send word one + 1
21   addi netTX, netRX, 1 // send word two + 1
22   csrwi lmsgdone, 1 // msg processing done
23   j wait_msg // wait for the next message
```

Listing 1: Loopback with increment. A nanoPU assembly program that waits for a 16B message, increments each word, and returns it to the sender.

to the core.[9] Finally, the procedure waits for the next message to arrive.

3.5 How It All Fits Together

Next, we walk through a more representative nanoRequest processing application, written in C, to compute the dot product of a vector stored in memory and a vector contained in arriving RPC request messages. Listing 2 is the C code for the routine, based on a small library of C macros (`lnic_*`) we wrote to allow applications to interact with the nanoPU hardware (`netRX` and `netTX` GPRs, and the CSRs). The `lnic_wait()` macro corresponds to the `wait_msg` procedure on lines 9-15 in Listing 1. The `lnic_read()` and `lnic_write_*()` macros generate instructions that either read from or write to `netRX` or `netTX` using either registers, memory, or an immediate; and the `lnic_msg_done()` macro writes to the `lmsgdone` CSR, corresponding to line 22 of Listing 1. Our library also includes other macros as well such as `lnic_branch()` which branches control flow based on the value in `netRX`.

The dot product C application waits for a message to arrive then extracts the application header (the first word of every message), followed by the message type in the second word. It checks that it is a `DATA_TYPE` message, and reads the third word to know how many 8B words the vector contains. The vector identifies the in-memory weight to use for each word

[7] It is the responsibility of the application to ensure that it does not try to read `netRX` when the local RX queue is empty; doing so results in undefined behavior.

[8] Note that this instruction also sets the TX message length to be equal to the RX message length because the message length is included in the TX/RX application headers.

[9] A future implementation may also want to use this signal to flush any unread bytes of the current message from the local RX queue. Doing so would guarantee that the next read to `netRX` would yield the application header of the subsequent message and help prevent application logic from becoming desynchronized with message boundaries.

```
while (1) {
  // Wait for a msg to arrive
  lnic_wait();
  // Extract application header from RX msg
      and check msg type
  app_hdr = lnic_read();
  if (lnic_read() != DATA_TYPE) {
    printf("Expected Data msg.\n");
    return -1;
  }
  // Compute the dot product of the msg
      vector with in-memory data
  uint64_t num_words = lnic_read();
  uint64_t result = 0;
  for (i = 0; i < num_words; i++) {
    uint64_t idx = lnic_read();
    uint64_t word = lnic_read();
    result += word * weights[idx];
  }
  // Send response message
  lnic_write_r((app_hdr & (IP_MASK |
      PORT_MASK)) | RESP_MSG_LEN);
  lnic_write_i(RESP_TYPE);
  lnic_write_r(result);
  lnic_msg_done();
}
```

Listing 2: Example nanoPU application that computes the dot product between a vector in a network message and in-memory weights.

when computing the dot product. Note that the application processes message data directly out of the register file and message data never needs to be copied into memory, allowing it to run faster than on a traditional system. Finally, the application sends a response message back to the sender containing the dot product.

4 The nanoPU Applications

Applications that will benefit most from using the nanoPU fast path exhibit one or both of the following characteristics: (i) strict tail response time requirements for network messages; or (ii) short (μs-scale) on-core service times. It should come as no surprise that applications with strict tail response time requirements will benefit from using the nanoPU fast path. Enabling low tail response time was one of our primary goals that guided many of the design decisions described in Section 2. For the latter, when an application's on-core service time is short, any CPU cycles spent sending or receiving network messages become comparatively more expensive. The nanoPU's extremely low per-message overheads help to ensure that these applications are able to dedicate close to 100% of CPU cycles to performing useful processing and thus achieve their maximum possible message processing throughput. Furthermore, the nanoPU can also help to reduce on-core service times by reducing pressure on the cache-hierarchy and allowing message data to be processed directly out of the register file. Another consequence of having short on-core service times is that the end-to-end completion time of each RPC becomes dominated by communication latency. By moving the entire network stack into hardware and by using the register file interface, the nanoPU fast path efficiently reduces communication latency and, hence, the RPC completion time. Therefore, the relative benefit provided by the nanoPU will increase as on-core service time decreases. An application's on-core service time does not necessarily need to be sub-1μs in order to benefit from using the nanoPU. The following section describes a few specific classes of applications that we believe are well-suited for the nanoPU.

4.1 Example Application Classes

μs-scale (or ns-scale) Services. An increasing number of datacenter applications are implemented as a collection of independent software modules called microservices. It is common for a single user request to invoke microservices across thousands of servers. At such large scale, the *tail* RPC response time dominates the end-to-end performance of these applications [17]. Furthermore, many microservices exhibit very short on-core service times; a key-value store is one such example that has sub-1μs service time. Therefore, these applications exhibit both of the characteristics described in the previous section and are ideal candidates to accelerate with the nanoPU.

Programmable One-sided RDMA. Modern NICs support RDMA for quick read and write access to remote memory. Some NICs support further "one-sided" operations in hardware: a single RDMA request leads to very low latency *compare-and-swap*, or *fetch-and-add*. It is natural to consider extending the set of one-sided operations to further accelerate remote memory operations [40, 55], for example *indirect read* (dereferencing a memory pointer in one round-trip time, rather than two), *scan and read* (scan a small memory region to match an argument and fetch data from a pointer associated with the match), *return max*, and so on. Changing fixed-function NIC hardware requires a new hardware design and fork-lift upgrade, and so, instead, Google Snap [40] implements a suite of custom one-sided operations in software in the kernel. This idea would run much faster on the nanoPU, for example as an embedded core on a NIC, and could implement arbitrary one-sided RDMA operations in software (Section 5.3).

High Performance Computing (HPC) and Flash Bursts. HPC workloads (e.g., N-body simulations [34]) as well as flash bursts [37], a new class of data center applications that utilize hundreds or thousands of machines for a short amount of time (e.g., one millisecond), are both examples of highly parallelizable application classes that are partitioned into fine-grained tasks distributed across many machines. These applications tend to be very communication intensive and spend a significant amount of time sending and receiving small messages [37]. We believe that the nanoPU's extremely low per-message overheads and low communication latency can help to accelerate these applications.

Network Function Virtualization (NFV). NFV is a well-known class of applications with μs-scale on-core service times [60, 66]. The nanoPU's low per-message overhead, register file interface, and programmable PISA pipelines allow it to excel at stream processing network data and thus is an excellent platform for deploying NFV applications.

5 Evaluation

Our evaluations address the following four questions:

1. How does the performance of the nanoPU register file interface compare to a traditional DMA-based network interface (Section 5.2.1)?
2. Is the hardware thread scheduler (HTS) able to provide low tail latency under high load and bounded tail latency for well-behaved applications (Section 5.2.2)?
3. How does our prototype NIC pipeline (i.e., transport and core selection) perform under high incast and service-time variance (Section 5.2.3)?
4. How do real applications perform using the nanoRequest fast path (Section 5.3)?

5.1 Methodology

We compare our nanoPU prototype against an unmodified RISC-V Rocket core with a standard NIC (IceNIC [31]), which we call a *traditional* NIC. The traditional NIC is implemented in the same simulation environment as our nanoPU prototype and performs DMA operations directly with the last-level (L2) cache. The traditional NIC does not support hardware-terminated transport or multi-core network applications, however, an ideal traditional NIC would support both of these. Therefore, for our evaluations, we do not implement transport in software for the traditional NIC baseline; we omit the overhead that would be introduced by this logic.

Our evaluations ignore the overheads of translating addresses because we run bare-metal applications using physical addresses. When using virtual memory, the traditional design would perform worse than reported here, because the message buffer descriptors would need to be translated resulting in additional latency, and more TLB misses. There is no need to translate addresses when processing nanoRequests from the register file.

Benchmark tools. We use two different cycle-accurate simulation tools to perform our evaluations: (1) the Verilator [63] software simulator, and (2) the Firesim [31] FPGA-accelerated simulator. Firesim enables us to run large-scale, cycle-accurate simulations with hundreds of nanoPU cores using FPGAs in AWS F1 [1]. The FPGAs run at 90MHz, and we simulate a target clock rate of 3.2GHz—all reported results are in terms of this target clock rate. The simulated servers are connected by C++ switch models running on the AWS x86 host CPUs.

5.2 Microbenchmarks

5.2.1 Register file interface

Loopback response time. Figure 2 shows a breakdown of the latency through each component for a single 8B nanoRequest message (in a 72B packet) measured from the Ethernet wire through a simple loopback application in the core, then back to the wire (first bit in to last bit out).[10] As shown, the loopback response time through the nanoPU fast path is only 17ns, but in practice we also need an Ethernet MAC and serial I/O, leading to a wire-to-wire response time of 69ns.

For comparison, Figure 3 shows the median loopback response time for both the nanoPU fast path and the traditional design for different messages sizes. For an 8B nanoRequest, the traditional design has a 51ns loopback response time, or about 3× higher than the nanoPU. 12ns (of the 51ns) are spent performing `memcpy`'s to swap the Ethernet source and destination addresses, something that is unnecessary for the nanoPU, because it is handled by the NIC hardware. The speedup of the nanoPU fast path decreases as the message size increases because the response time becomes dominated by store-and-forward delays and message-serialization time.

If instead the traditional NIC placed arriving messages directly in the L1 cache, as NeBuLa proposes [57], the loopback response time would be faster, but the nanoPU fast path would still have 50% lower response time for small nanoRequests.

Loopback throughput. Figure 4 shows the throughput of the simple loopback application running on a single core for both the nanoPU fast path and the traditional NIC. The traditional NIC processes batches of 30 packets, which fit comfortably in the LLC. Batching allows the traditional NIC to overlap computation (e.g., Ethernet address swapping) with NIC DMA send/receive operations.

Throughput is dominated by the software overhead to process each message. For the register file interface, the software overhead is: read the `lmsgsrdy` CSR to check if a message is available for processing, read the message length from the application header, and write to the `lmsgdone` CSR after forwarding the message. For the traditional design, the software overhead is: perform MMIO operations to pass RX/TX descriptors to the NIC and to check for RX/TX DMA completions, and `memcpy`'s to swap the Ethernet source and destination addresses.

Because of lower overheads, the application has 2–7× higher throughput on the nanoPU than on the traditional NIC. For small 8B messages (72B packets), the nanoPU loopback application achieves 68Gb/s, or 118Mrps – 7× higher than the traditional system. For 1KB messages, the nanoPU achieves a throughput of 166Gb/s (83% of the line-rate). When we add the per-packet NDP control packets sent/received by the NIC, the 200Gb/s link is completely saturated.

[10] Our prototype does not include MAC & Serial IO, so we add real values measured on a 100GE switch (with Forward Error Correction disabled).

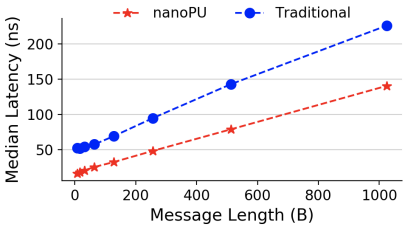


Figure 3: Loopback median response time vs. message length; nanoPU fast path and traditional.

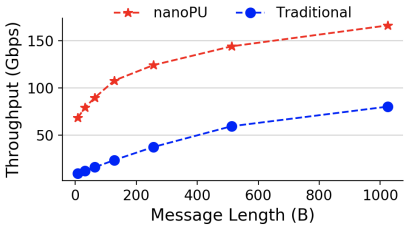


Figure 4: Loopback throughput vs. message length; nanoPU fast path and traditional.

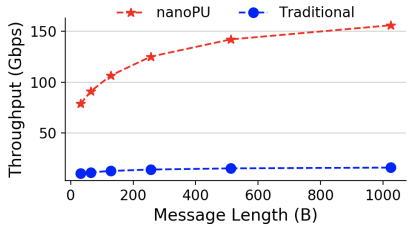


Figure 5: Loopback-with-increment throughput vs. message length; nanoPU fast path and traditional.

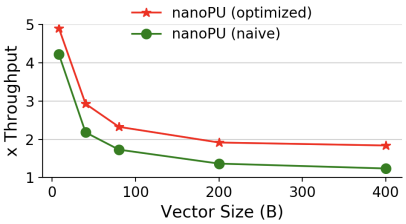


Figure 6: Dot-product throughput speedup for various vector sizes; nanoPU fast path (naive & optimal) relative to traditional NIC.

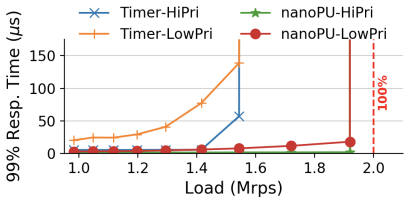


Figure 7: 99th %ile response time vs load; hardware thread scheduler (HTS) vs. traditional timer-interrupt driven scheduler (TIS).

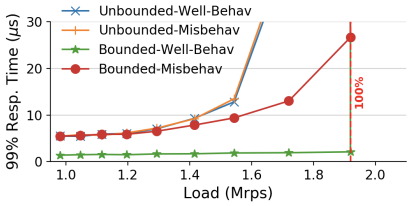


Figure 8: 99th %ile response time vs load for well-behaved and misbehaved threads, with and without bounded message processing time.

Stateless nanoRequest jobs. The nanoPU is well-suited for compute-intensive applications that transform the data carried by self-contained nanoRequests. We use a very simple benchmark application that increments each word of the message by one and forwards the message back into the network; similar to the program described in Section 3.4.

Figure 5 shows that the nanoPU accelerates the throughput of this application by up to 10×. NanoRequest data is read from the register file and passed directly through the ALU; no memory operations are required at all. On the other hand, when using the traditional NIC, each word of the message must be read from the last-level cache (LLC), passed through the ALU, and the final result is written back to memory. If instead the traditional NIC loaded words into the L1 cache, as in [57], we estimate a throughput about 1.3× faster than via the LLC. This would still be 7.5× slower than the nanoPU fast path. In Section 5.3, we will compare more realistic benchmarks for real applications.

Stateful nanoRequest jobs. These are applications that process both message data and local memory data. Similar to the example described in Section 3.5, our simple microbenchmark computes the dot product of two vectors of 64-bit integers, one from the arriving message and a *weight vector* in local memory. The weight vector is randomly chosen from enough vectors to fill the L1 cache (16kB).

There are two ways to implement the application on the nanoPU. The *optimal* method is to process each message word directly from the register file, multiplying and accumulating each word with the corresponding weight value from memory. The *naive* method copies the entire message from `netRX` into memory before computing the dot product with the weight vector. The *traditional* design processes messages in batches of 30, to overlap dot-product computation with DMA operations.

Figure 6 shows the throughput speedup of the *optimal* and *naive* methods relative to the traditional application, for different message lengths.

- *Small messages:* For small messages below 100bytes, the nanoPU is 4–5× faster because of fewer per-message software overheads.
- *Large messages:* For large vectors throughput is limited by the longer dot product computation time. The *optimal* application consistently doubles throughput by keeping message data out of the L1 cache and reducing cache misses. The *naive* application is slowed by the extra copy, and about twice as many L1 data cache misses. The *traditional* application has 10× as many L1 data cache misses as *optimal* because message data must be fetched from the LLC, which pollutes the L1 cache, evicting weight data. If we speed up the traditional NIC by placing message data directly in the L1 cache, as NeBuLa proposes [57], we estimate the traditional design would run 1.5× faster for large messages. *Optimal* would still be 30% faster for large messages.

The benefits are clear when an application processes message data directly from the `netRX` register. While this may seem like a big constraint, we have found that it is generally feasible and natural to design applications this way. We demonstrate example applications in Section 5.3.

5.2.2 Hardware thread scheduling

Next, we evaluate how much the hardware thread scheduler (HTS) can reduce tail response time under high load.

Methodology. We evaluate tail response time under load by connecting a custom (C++) load generator to our nanoPU prototype in Firesim [31]. It generates nanoRequests with Poisson inter-arrival times, and measures the end-to-end response time.

Priority thread scheduling. We compare our hardware thread scheduler (HTS) against a more traditional timer-interrupt driven scheduler (TIS) interrupted by the kernel every 5μs to swap in the highest-priority active thread. We run both schedulers in hardware on our prototype.[11] TIS uses a 5μs timer interrupt to match the granularity of state-of-the-art low-latency operating systems [27, 49].

We evaluate both schedulers when they are scheduling two threads: one with priority 0 (high) and one with priority 1 (low). The load generator issues 10K requests for each thread, randomly interleaved, each with an on-core service time of 500ns (i.e., an ideal system will process 2Mrps).

Figure 7 shows the 99th %ile tail response time vs load for both thread scheduling policies, with a high and low priority thread. HTS reduces tail response time by 4× and 6.5× at high and low load, respectively; and can sustain 96% load.[12]

Bounded message-processing time. HTS is designed to bound the tail response time of well-behaved applications, even when they are sharing a core with misbehaving applications. To test this, we configure a core to run a well-behaved thread and a misbehaving thread, both configured to run at priority 0. All requests have an on-core service time of 500ns, except when a thread misbehaves (once every 100 requests), in which case the request processing time increases to 5μs.

Figure 8 shows the 99th %ile tail response time vs load for both threads with, and without, the bounded message processing time feature enabled. When enabled, if a priority 0 thread takes longer than 1μs to process a request, HTS lowers its priority to 1. When disabled, all requests are processed by the core in FIFO order.

We expect an application with at most one message at a time in the RX queue, to have a tail response time bounded by $2 \cdot 43ns + 17ns + 2 \cdot 1000ns + 50ns = 2.15\mu s$. This matches our experiments: With the feature enabled, the tail response time of the well-behaved thread never exceeds 2.1μs, until the offered load on the system exceeds 100% (1.9 Mrps).[13] HTS lowers the priority of the misbehaving application the first time it takes longer than 1μs to process a request. Hence, the well-behaved thread quickly becomes strictly higher priority and its 500ns requests are never trapped behind a long 5μs one. Note also that by bounding message processing times, shorter requests are processed first, queues are smaller and the system can sustain higher load.

[11] TIS would run in software in practice, likely on a separate core, and would therefore be slower than in hardware.

[12] Our prototype does not currently allocate NIC buffer space per-application, causing high-priority requests to be dropped when the low-priority queue is fill. This will be fixed in the next version.

[13] This is despite our Poisson arrival process occasionally placing more than one message in the RX queue.

5.2.3 Prototype NIC pipeline

Hardware NDP transport. We verify our hardware NDP implementation by running a large 80-to-1 incast experiment on Firesim, with 324 cores simulated on 81 AWS F1 FPGAs. All hosts are connected to one simulated switch; 80 clients send a single packet message to the same server at the same time. The switch has insufficient buffer capacity to store all 80 messages and hence some are dropped. When NDP is disabled, dropped packets are detected by the sender using a timeout and therefore the maximum latency through the network is dictated by the timeout interval. When NDP is enabled, the dropped messages are quickly retransmitted by NDP's packet trimming and NACKing mechanisms, lowering maximum network latency by a factor of three.

Hardware JBSQ core selection. We evaluate our JBSQ implementation using a bimodal service-time distribution: 99.5% of nanoRequests have a service time of 500ns and 0.5% have a service time of 5μs. When using a random core assignment technique, like receive side scaling (RSS), to balance requests across four cores, short requests occasionally get queued behind long requests, resulting in high tail response time. With JBSQ enabled, tail response time is reduced 5× at low load, and can sustain 15% higher load than RSS.

5.3 Application Benchmarks

As shown in Table 1, we implemented and evaluated many applications on our nanoPU prototype. Below, we present the evaluation results for a few of these applications.

MICA. We ported the MICA key-value store [38] and compared it running on the nanoPU and traditional NIC designs. MICA is implemented as a library with an API that allows applications to GET and SET key-value pairs. Traditionally, this API uses in-memory buffers to pass key-value pairs between the MICA library and application code. The *naive* way to port MICA to the nanoPU is to copy key-value pairs in network messages between the register file and in-memory buffers, using the MICA library without modification. However, we find it more efficient to modify the MICA library to read and write the register file directly when performing GET and SET operations. This avoids unnecessary `memcpy`s in the MICA library. Optimizing the MICA library to use the register file only required changes to 36 lines of code.

Our evaluation stores 10k key-value pairs (16B keys and 512B values). The load generator sends a 50/50 mix of read-/write nanoRequest queries with keys picked uniformly. Figure 9 compares the 99th %ile tail response time vs load for the traditional, nanoPU naive, and nanoPU optimized versions of this application. The naive nanoPU implementation outperforms the traditional implementation, likely because it is able to use an L1-cache resident in-memory buffer rather than an LLC-resident DMA buffer. The optimized nanoPU imple-

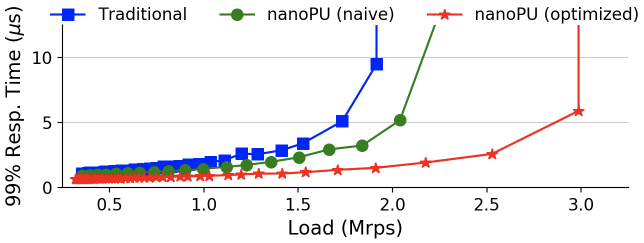


Figure 9: MICA KV store: 99th %ile tail response time for READ and WRITE requests.

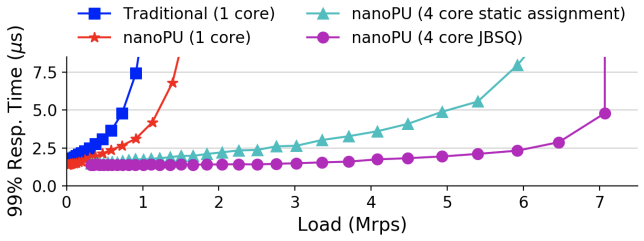


Figure 10: Set intersection: 99th %ile tail response time.

mentation is able to achieve about 30% higher throughput and lower response times by efficiently using the register file interface when processing network messages.

Raft. Raft is a widely-used consensus algorithm for distributed applications [47]. We evaluate a production grade version of Raft [53] using a 16B-key, 64B-value MICA key-value store state machine, with three servers and one client connected to a single switch. The switch has a forwarding latency of 300ns (typical of modern cut-through commercial switch ASICs [58]) and all links have a latency of 43ns. Although our Raft cluster correctly implements leader election, can tolerate server failure, and our client can automatically identify a new Raft leader, we evaluate our Raft cluster in the steady-state, failure-free case, with a single leader and three fully-functioning replicas.

We define the response time to be from when the client issues a three-way replicated write request to the Raft cluster, until the client hears back from the cluster leader that the request has been fully replicated and committed across all three Raft servers. In 10K trials, the median response time was 3.08μs, with a 3.26μs 99th %ile tail response time. eRPC [28], a high performance, highly-optimized RPC library reports a 5.5μs median and 6.3μs 99th %ile tail response time — about a factor of two slower.

Set algebra. In information retrieval systems, set intersections are commonly performed for data mining, text analytics, and search. For example, Lucene [8] uses a reverse index that maps each word to a set of documents that contain the word. Searches yield a document set for each search word, then compute the intersection of these sets.

We created a reverse index of 100 Wikipedia [65] articles with 200 common English words. Our load generator sends search requests with 1-4 words chosen from a Zipf distribution based on word frequency. Porting the set intersection application to the nanoPU was straight forward. The only difference between the nanoPU and traditional versions of the applications is the logic to send and receive network messages (∼50 LOC). We did not need to make any modifications to the application logic that computes the intersection between sets of document IDs.

One-sided RDMA	Latency (ns) Median	90th %ile
Read	678	680
Write	679	686
Compare-and-Swap	687	690
Fetch-and-Add	688	692
Indirect Read	691	715

Table 2: Median and 90th %ile latency of one-sided RDMA operations implemented on the nanoPU. Measurements are made at the client, and the one-way latency through the switch and links is 300ns.

Figure 10 shows the tail response time for searches. The traditional design has a low-load tail response time of 1.7μs, compared to 1.4μs on a single nanoPU core. JBSQ helps to ensure that long running requests do not get stuck behind short ones. With JBSQ enabled for four cores, the 99th %ile tail response time remains low until 7Mrps.

One-sided RDMA operations. As described in Section 4.1, the nanoPU can implement flexible, low latency one-sided RDMA operations. As a baseline, the median end-to-end latency of one-sided operations between two hosts using state-of-the-art RDMA NICs, connected by a single switch with a port-to-port latency of 300ns is about 2μs [28]. [14] Table 2 shows the median and 90% tail latency of several one-sided RDMA operations implemented on the nanoPU, using the same topology as the baseline. The median latency, measured by the nanoPU client, is 680-690ns with a 90% tail latency of approximately 700ns, 65% lower latency than state-of-the-art RDMA NICs. Most of the latency reduction is from eliminating the traversal of PCIe on the client and server.

In addition to the standard one-sided RDMA operations (read, write, compare-and-swap, fetch-and-add) we also implement *indirect read*, in which the server simply dereferences a pointer to determine the actual memory address to read. This operation would require two network round trips on a standard RDMA NIC; on the nanoPU, it takes only a few nanoseconds longer than a standard read.

6 Discussion

nanoPU deployment possibilities. We believe there are a number of ways to deploy nanoPU ideas, in addition to a modified regular CPU. For example, the nanoPU fast path

[14] Note that when using an ARM-based smartNIC, such as the Mellanox BlueField [41], the time to traverse the embedded cores will increase this end-to-end latency by at least a factor of two [39, 61].

could be added to embedded CPUs on smartNICs for the data center [5, 41, 44]. This could be a less invasive way to introduce nanoPU ideas without needing to modify server CPUs. A more extreme approach would be to build a nanoPU domain-specific architecture explicitly for nanoRequests. For example, it would be practical today to build a single chip 512-core nanoPU, similar to Celerity [15], with one hundred 100GE interfaces, capable of servicing RPCs at up to 10Tb/s.

In-order execution. Our prototype is based on a simple 5-stage, in-order RISC-V Rocket core and required only minor modifications to the CPU pipeline. An out-of-order processor would require bigger changes to ensure that words read from `netRX` are delivered to the application in FIFO order.

7 Related Work

Low-latency RPCs (software). Recent work focuses on algorithms to choose a core by approximating a single-queue system using work-stealing (like ZygOS [51]) or preempting requests at microsecond timescales (Shinjuku [27]). However, the overheads associated with inter-core synchronization and software preemption make these approaches too slow and coarse-grained for nanoRequests.

eRPC [28] takes the other extreme to the nanoPU and runs everything in software, and through clever optimizations, achieves impressively low latency on a commodity server for the common case. eRPC has good median response times, but its common-case optimizations sacrifice tail response times, which often dictate application performance. The nanoPU's hardware pipeline makes median and tail RPC response times almost identical.

Low-latency RPCs (hardware). We are not the first to implement core-selection algorithms in hardware. RPCvalet [12] and NeBuLa [57] are both built on the Scale-out NUMA architecture [46]. RPCvalet implements a single queue system, which in theory provides optimal performance. However, it ran into memory bandwidth contention issues, which they later resolve in NeBuLa. Both NeBuLa and R2P2 [36] implement the JBSQ load balancing policy; NeBuLa runs JBSQ on the server whereas R2P2 runs JBSQ in a programmable switch. Like NeBuLa, the nanoPU also implements JBSQ to steer requests to cores.

Many NICs support RDMA in hardware. Several systems (HERD [29], FaSST [30], and DrTM+R [11]) exploit RDMA to build applications on top. As described in Sections 4.1 and 5.3, the nanoPU can be used to implement programmable one-sided RDMA operations while providing lower latency than state-of-the-art commercial NICs.

SmartNICs (NICs with CPUs on them) [5, 41, 44] are being deployed to offload infrastructure software from the main server to CPUs on the NIC. However, these may actually increase the RPC latency, unless they adopt nanoPU-like designs on the NIC.

Transport protocols in hardware. We are not the first to implement the transport layer and congestion control in hardware. Modern NICs that support RDMA over Converged Ethernet (RoCE) implement DCQCN [67] in hardware. In the academic research community, Tonic [2] proposes a framework for implementing congestion control in hardware. The nanoPU's programmable transport layer (and NDP implementation) draws upon ideas in Tonic.

Register file interface. GPRs were first used by the J-machine [13] for low-latency inter-core communication on the same machine, but were abandoned because of the difficulty implementing thread-safety. The idea has reappeared in several designs, including the RAW processor [64], and the SNAP processor for low-power sensor networks [32].

8 Conclusion

Today's CPUs are optimized for load-store operations to and from memory. Memory data is treated as a first-class citizen. But modern workloads frequently process huge numbers of small RPCs. Rather than burden RPC messages with traversing a hierarchy optimized for data sitting in memory, we propose providing them with a new optimized fast path, inserting them directly into the heart of the CPU, bypassing the unnecessary complications of caches, PCIe and address translation. Hence, we aim to elevate network data to the same importance as memory data.

As datacenter applications continue to scale out, with one request fanning out to generate many more, we must find ways to minimize not only the communication overhead, but also the *tail* response time. Long tail response times are inherently caused by resource contention (e.g., shared CPU cores, cache space, and memory and network bandwidths). By moving key scheduling decisions into hardware (i.e., congestion control, core selection, and thread scheduling), these resources can be scheduled extremely efficiently and predictably, leading to lower tail response times.

If future cloud providers can provide bounded, end-to-end RPC response times for very small nanoRequests, on shared servers also carrying regular workloads, we will likely see much bigger distributed applications based on finer grain parallelism. Our work helps to address part of the problem: bounding the RPC response time once the request arrives at the NIC. If coupled with efforts to bound network latency, it might complete the end-to-end story. We hope our results will encourage other researchers to push these ideas further.

Acknowledgements

We would like to thank our shepherd, Yiying Zhang, Amin Vahdat, John Ousterhout, and Kunle Olukotun for their invaluable suggestions throughout the duration of this project. This work was supported by Xilinx, Google, Stanford Platform Lab, and DARPA Contract Numbers HR0011-20-C-0107 and FA8650-18-2-7865.

A Artifact Appendix

Abstract

This artifact contains the Chisel source code of our nanoPU prototype as well as the application code and simulation infrastructure that is required to reproduce the key results presented in this paper. Our prototype is evaluated using both Verilator for cycle-accurate simulations in software, and Firesim for cycle-accurate simulations on FPGAs in AWS. The artifact is packaged as an AWS EC2 image with all of the dependencies pre-installed to make it easy for others to use and build upon our work.

Scope

The artifact can be used to reproduce the key results presented in the following figures:

- **Figure 3** – Loopback latency; nanoPU vs. traditional.
- **Figure 4** – Loopback throughput; nanoPU vs. traditional.
- **Figure 5** – Loopback-with-inc. throughput; nanoPU vs. traditional.
- **Figure 6** – Dot-product throughput speedup.
- **Figure 7** – Tail response time using priority thread scheduling.
- **Figure 8** – Tail response time using bounded message processing time.
- **Figure 9** – MICA tail response time.
- **Figure 10** – Set intersection tail response time.

Additionally, there are a number of ways to use the artifact to build upon our work. For example, you can write new applications for the nanoPU and evaluate them at close to real-time using a custom topology with Firesim. Alternatively, you can modify the nanoPU architecture and use the provided simulation infrastructure to easily test your changes.

Contents

The documentation for the nanoPU artifact can be found at: https://github.com/l-nic/chipyard/wiki. The primary repositories are briefly described below:

- Chipyard – The main top-level repository which contains the others listed below as git submodules. Contains application code as well as Verilator simulation infrastructure.
- Rocket Chip – Contains the chisel source code for our modified RISC-V Rocket core (as well as all of the other components that are needed to create a full SoC).
- L-NIC – Contains the chisel source code for the nanoPU NIC.
- Firesim – Provides all of the infrastructure that is required to run FPGA-accelerated, cycle-accurate simulations on AWS.

Hosting

The nanoPU artifact is hosted on GitHub:
https://github.com/l-nic/chipyard/tree/nanoPU-artifact-v1.0

The development branch is called `lnic-dev` and, at the time of this writing, the latest release is tagged `nanoPU-artifact-v1.0`. In order to make it easy for others to reuse and build upon our work, we have developed a custom Amazon Machine Image (AMI) with the artifact and all required dependencies pre-installed. See the documentation for detailed instructions regarding how to access and use this AMI.

Requirements

In order to use the nanoPU artifact, you will need access to an AWS account and you will need to subscribe to the AWS FPGA developer AMI. Additionally, you will need permission from AWS to launch F1 instances. These requirements are explained in greater detail in the online documentation.

References

[1] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed on 2020-08-10.

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, 2020.

[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Aws nitro system. https://aws.amazon.com/ec2/nitro/. Accessed on 2020-12-10.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[8] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[11] 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*, pages 1–17, 2016.

[12] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[13] William J Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, and John Keen. The j-machine: A fine grain concurrent computer. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER, 1989.

[14] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[15] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.

[16] Intel corporation. intel data direct i/o technology (intel ddio): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. Accessed on 2020-08-17.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] DPDK: Data Plane Development Kit. https://www.dpdk.org/. Accessed on 2020-12-04.

[19] eBPF – extended Berkeley Packet Filter. https://prototype-kernel.readthedocs.io/en/latest/bpf/. Accessed on 2020-12-08.

[20] Cisco Nexus X100 SmartNIC K3P-Q Data Sheet. https://www.cisco.com/c/en/us/products/collateral/interfaces-modules/nexus-smartnic/datasheet-c78-743828.html. Accessed on 2020-12-01.

[21] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689, 2020.

[22] Sadjad Fouladi, Dan Iter, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure, 2017.

[23] 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 *USENIX NSDI*, 2017.

[24] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[25] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 133–140, 2019.

[26] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The case for a network fast path to the cpu. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 52–59, 2019.

[27] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306, 2014.

[30] 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, 2016.

[31] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[32] Clinton Kelly, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor-network asynchronous processor. In *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, pages 24–33. IEEE, 2003.

[33] Richard E Kessler and James L Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Digest of Papers. COMPCON Spring*, pages 176–182. IEEE, 1993.

[34] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 57–64. IEEE, 2016.

[35] 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.

[36] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 863–880, 2019.

[37] Yilong Li, Seo Jin Park, and John Ousterhout. Millisort and milliquery: Large-scale data-intensive computing in milliseconds. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 2021.

[38] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[39] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. Association for Computing Machinery, 2019.

[40] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[41] Mellanox bluefield-2. https://www.mellanox.com/products/bluefield2-overview. Accessed on 2020-12-10.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 18)*, pages 221–235, 2018.

[43] Microsoft: Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. Accessed on 2020-12-07.

[44] Naples dsc-100 distributed services card. https://www.pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf. Accessed on 2020-12-10.

[45] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[46] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[47] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[48] Options for Code Generation Conventions. https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options. Accessed on 2020-11-11.

[49] 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, 2019.

[50] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, 2020.

[51] 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 (SOSP 17)*, pages 325–341, 2017.

[52] Google protocol buffers. https://developers.google.com/protocol-buffers. Accessed on 2020-12-08.

[53] raft GitHub. https://github.com/willemt/raft. Accessed on 2020-08-17.

[54] Rocket-chip github. https://github.com/chipsalliance/rocket-chip. Accessed on 2020-08-17.

[55] 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*, pages 1–16, 2020.

[56] Akshitha Sriraman and Thomas F Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[57] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NEBULA rpc-optimized architecture. Technical report, 2020.

[58] Sx1036 product brief. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1036.pdf. Accessed on 2020-09-12.

[59] Apache thrift. https://thrift.apache.org/. Accessed on 2020-12-08.

[60] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, 2018.

[61] 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} 20)*, pages 33–48, 2020.

[62] User level threads. http://pdxplumbers.osuosl.org/2013/ocw//system/presentations/1653/original/LPC%20-%20User%20Threading.pdf. Accessed on 2020-12-08.

[63] Verilator. https://www.veripool.org/wiki/verilator. Accessed on 2020-01-29.

[64] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[65] Wikipedia:database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed on 2020-12-08.

[66] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. Halo: accelerating flow classification for scalable packet processing in nfv. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 601–614. IEEE, 2019.

[67] 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. *ACM SIGCOMM Computer Communication Review (CCR)*, 45(4):523–536, 2015.

Beyond `malloc` efficiency to fleet efficiency: a hugepage-aware memory allocator

A.H. Hunter
Jane Street Capital[*]

Chris Kennelly
Google

Paul Turner
Google

Darryl Gove
Google

Tipp Moseley
Google

Parthasarathy Ranganathan
Google

Abstract

Memory allocation represents significant compute cost at the warehouse scale and its optimization can yield considerable cost savings. One classical approach is to increase the efficiency of an allocator to minimize the cycles spent in the *allocator code*. However, memory allocation decisions also impact overall application performance via data placement, offering opportunities to improve *fleetwide productivity* by completing more units of application work using fewer hardware resources. Here, we focus on hugepage coverage. We present TEMERAIRE, a hugepage-aware enhancement of TCMALLOC to reduce CPU overheads in the application's code. We discuss the design and implementation of TEMERAIRE including strategies for hugepage-aware memory layouts to maximize hugepage coverage and to minimize fragmentation overheads. We present application studies for 8 applications, improving requests-per-second (RPS) by 7.7% and reducing RAM usage 2.4%. We present the results of a 1% experiment at fleet scale as well as the longitudinal rollout in Google's warehouse scale computers. This yielded 6% fewer TLB miss stalls, and 26% reduction in memory wasted due to fragmentation. We conclude with a discussion of additional techniques for improving the allocator development process and potential optimization strategies for future memory allocators.

1 Introduction

The *datacenter tax* [23, 41] within a warehouse-scale computer (WSC) is the cumulative time spent on common service overheads, such as serialization, RPC communication, compression, copying, and memory allocation. WSC workload diversity [23] means that we typically cannot optimize single application(s) to strongly improve total system efficiency, as costs are borne across many independent workloads. In contrast, focusing on the components of datacenter tax can realize substantial performance and efficiency improvements in aggregate as the benefits can apply to entire classes of application. Over the past several years, our group has focused on minimizing the cost of memory allocation decisions, to great effect; realizing whole system gains by dramatically reducing the time spent in memory allocation. But it is not only the cost of these components we can optimize. Significant benefit can also be realized by improving the efficiency of application code by changing the allocator. In this paper, we consider how to optimize application performance by improving the hugepage coverage provided by memory allocators.

Cache and Translation Lookaside Buffer (TLB) misses are a dominant performance overhead on modern systems. In WSCs, the memory wall [44] is significant: 50% of cycles are stalled on memory in one analysis [23]. Our own workload profiling observed approximately 20% of cycles stalled on TLB misses.

Hugepages are a processor feature that can significantly reduce the number, and thereby the cost, of TLB misses [26]. The increased size of a hugepage enables the same number of TLB entries to map a substantially larger range of memory. On the systems under study, hugepages also allow the total stall time for a miss+fill to be reduced as their page-table representation requires one fewer level to traverse.

While an allocator cannot modify the amount of memory that user code accesses, or even the pattern of accesses to objects, it can cooperate with the operating system and control the placement of new allocations. By optimizing hugepage coverage, an allocator may reduce TLB misses. Memory placement decisions in languages such as C and C++ must also deal with the consequence that their decisions are final: Objects cannot be moved once allocated [11]. Allocation placement decisions can only be optimized at the point of allocation. This approach ran counter to our prior work in this space, as we can potentially increase the CPU cost of an allocation, *increasing* the datacenter tax, but make up for it by *reducing* processor stalls elsewhere. This improves application metrics[1] such as requests-per-second (RPS).

[*]Work performed while at Google.

[1]While reducing stalls can improve IPC, IPC alone is a poor proxy [3] for how much useful application work we can accomplish with a fixed amount

Our contributions are as follows:

- The design of TEMERAIRE, a hugepage-aware enhancement of TCMALLOC to reduce CPU overheads in the rest of the application's code. We present strategies for hugepage-aware memory layouts to maximize hugepage coverage and to minimize fragmentation overheads.
- An evaluation of TEMERAIRE in complex real-world applications and scale in WSCs. We measured a sample of 8 applications running within our infrastructure observed requests-per-second (RPS) increased by 7.7% and RAM usage decreased by 2.4%. Applying these techniques to all applications within Google's WSCs yielded 6% fewer TLB miss stalls, and 26% reduction in memory wasted due to fragmentation.
- Strategies for optimizing the development process of memory allocator improvements, using a combination of tracing, telemetry, and experimentation at warehouse-scale.

2 The challenges of coordinating Hugepages

Virtual memory requires translating user space addresses to *physical* addresses via caches known as Translation Lookaside Buffers (TLBs) [7]. TLBs have a limited number of entries, and for many applications, the entire TLB only covers a small fraction of the total memory footprint using the default page size. Modern processors increase this coverage by supporting *hugepages* in their TLBs. An entire aligned hugepage (2MiB is a typical size on x86) occupies just one TLB entry. *Hugepages* reduce stalls by increasing the effective capacity of the TLB and reducing TLB misses [5, 26].

Traditional allocators manage memory in page-sized chunks. Transparent Huge Pages (THP) [4] provide an opportunity for the kernel to opportunistically cover consecutive pages using hugepages in the page table. A memory allocator, superficially, need only allocate hugepage-aligned and -sized memory blocks to take advantage of this support.

A memory allocator that *releases* memory back to the OS (necessary at the warehouse scale where we have long running workloads with dynamic duty cycles) has a much harder challenge. The return of non-hugepage aligned memory regions requires that the kernel use smaller pages to represent what remains, defeating the kernel's ability to provide hugepages and imposing a performance cost for the remaining used pages. Alternatively, an allocator may wait for an entire hugepage to become free before returning it to the OS. This preserves hugepage coverage, but can contribute significant amplification relative to true usage, leaving memory idle. DRAM is a significant cost the deployment of WSCs [27]. The management of *external fragmentation*, unused space in blocks too small to be used for requested allocations, by the allocator is important in this process. For example consider the allocations in Figure 1. After this series of allocations there are 2 units of free space. The choice is to either use small pages, which result in lower fragmentation but less efficient use of TLB entries, or hugepages, which are TLB-efficient but have high fragmentation.

of hardware. A busy-looping spinlock has extremely high IPC, but does little useful work under contention.

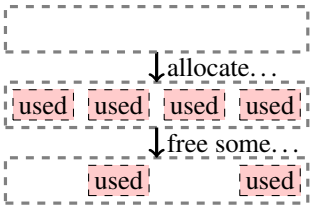


Figure 1: Allocation and deallocation patterns leading to fragmentation

A user-space allocator that is aware of the behavior produced by these policies can cooperate with their outcomes by densely aligning the packing of allocations with hugepage boundaries, favouring the use of allocated hugepages, and (ideally) returning unused memory at the same alignment[2]. A *hugepage-aware allocator* helps with managing memory contiguity at the user level. The goal is to maximally pack allocations onto nearly-full hugepages, and conversely, to minimize the space used on empty (or emptier) hugepages, so that they can be returned to the OS as complete hugepages. This efficiently uses memory and interacts well with the kernel's transparent hugepage support. Additionally, more consistently allocating and releasing hugepages forms a positive feedback loop: reducing fragmentation at the kernel level and improving the likelihood that future allocations will be backed by hugepages.

3 Overview of TCMALLOC

TCMALLOC is a memory allocator used in large-scale applications, commonly found in WSC settings. It shows robust performance [21]. Our design builds directly on the structure of TCMALLOC.

Figure 2 shows the organization of memory in TCMALLOC. Objects are segregated by size. First, TCMALLOC partitions memory into *spans*, aligned to page size[3].

TCMALLOC's structure is defined by its answer to the same two questions that drive any memory allocator.

1. How do we pick object sizes and organize metadata to

[2] This is important as the memory backing a hugepage must be physically contiguous. By returning complete hugepages we can actually assist the operating system in managing fragmentation.

[3] Confusingly, TCMALLOC's "page size" parameter is not necessarily the system page size. The default configuration is to use an 8 KiB TCMALLOC "page", which is two (small) virtual memory pages on x86.

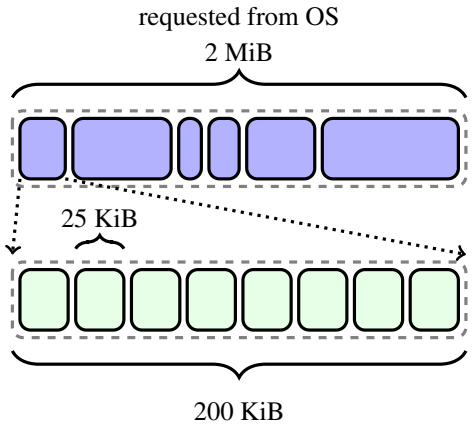


Figure 2: Organization of memory in TCMALLOC. System-mapped memory is broken into (multi-)page *spans*, which are sub-divided into objects of an assigned, fixed *sizeclass*, here 25 KiB.

minimize space overhead and fragmentation?
2. How do we scalably support concurrent allocations?

Sufficiently large allocations are fulfilled with a span containing only the allocated object. Other spans contain multiple smaller objects of the same size (a *sizeclass*). The "small" object size boundary is 256 KiB. Within this "small" threshold, allocation requests are rounded up to one of 100 sizeclasses. TCMALLOC stores objects in a series of caches, illustrated in

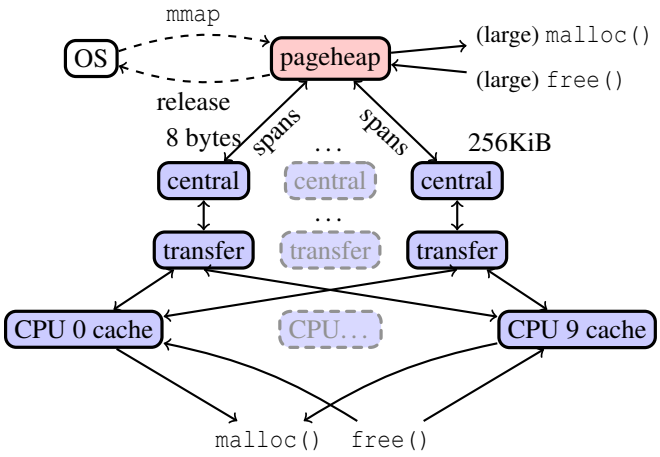


Figure 3: The organization of caches in TCMALLOC; we see memory allocated from the OS to the pageheap, distributed up into spans given to the central caches, to local caches. This paper focuses on a new implementation for the pageheap.

Figure 3. Spans are allocated from a simple *pageheap*, which keeps track of all unused pages and does best-fit allocation.

The pageheap is also responsible for returning no-longer-needed memory to the OS when possible. Rather than doing this on the `free()` path, a dedicated release-memory method is invoked periodically, aiming to maintain a configurable, steady rate of release in MB/s. This is a heuristic. TCMALLOC wants to simultaneously use the least memory possible in steady-state, avoiding expensive system allocations that could be elided by using previously provisioned memory. We discuss handling this peak-to-trough allocation pattern in more detail in Section 4.3.

Ideally, TCMALLOC would return all memory that user code will not need *soon*. Memory demand varies unpredictably, making it challenging to return memory that will go unused while simultaneously retaining memory to avoid syscalls and page faults.. Better decisions about memory return policies have high value and are discussed in section 7.

TCMALLOC will first attempt to serve allocations from a "local" cache, like most modern allocators [9,12,20,39]. Originally these were the eponymous per-**T**hread **C**aches, storing a list of free objects for each sizeclass. To reduce stranded memory and improve re-use for highly threaded applications, TCMALLOC now uses a per-hyperthread local cache. When the local cache has no objects of the appropriate sizeclass to serve a request (or has too many after an attempt to `free()`), requests route to a single *central cache* for that sizeclass. This has two components–a small fast, mutex-protected *transfer cache* (containing flat arrays of objects from that sizeclass) and a large, mutex-protected *central freelist*, containing every span assigned to that sizeclass; objects can be fetched from, or returned to these spans. When all objects from a span have been returned to a span held in the central freelist, that span is returned to the *pageheap*.

In our WSC, most allocations are small (50% of allocated space is objects $\leq$ 8192 bytes), as depicted in Figure 4. These are then aggregated into spans. The pageheap primarily allocates 1- or 2-page spans, as depicted in Figure 5. 80% of spans are smaller than a hugepage.

The design of "stacked" caches make the system usefully modular, and there are several concomitant advantages:

- Clean abstractions are easier to understand and test.
- It's reasonably direct to replace any one level of the cache with a totally new implementation.
- When desired, cache implementations can be selected at runtime, with benefits to operational rollout and experimentation.

TCMALLOC's pageheap has a simple interface for managing memory.

- `New(N)` allocates a span of N pages
- `Delete(S)` returns a `New`'d span (`S`) to the allocator.
- `Release(N)` gives >= N unused pages cached by the page heap back to the OS

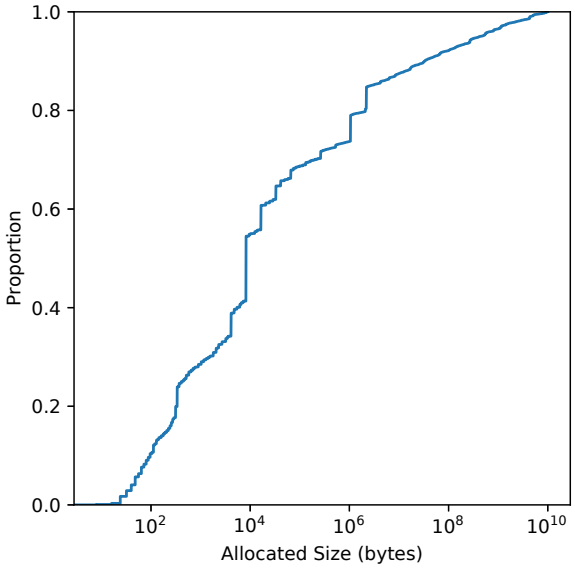


Figure 4: CDF of allocation sizes from WSC applications, weighted by bytes.

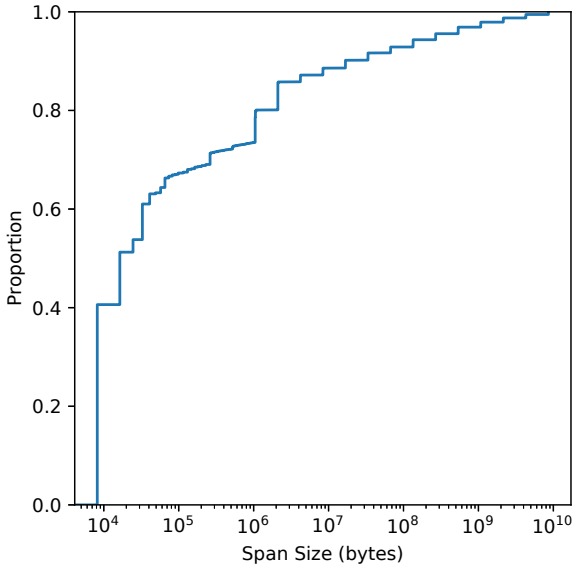


Figure 5: CDF of TCMALLOC span sizes from WSC applications, weighted by bytes.

4 TEMERAIRE's approach

TEMERAIRE, this paper's contribution to TCMALLOC, replaces the pageheap with a design that attempts to maximally fill (and empty) hugepages. The source code is on Github (see Section 9). We developed heuristics that pack allocations densely onto highly-used hugepages and simultaneously form entirely unused hugepages for return to the OS.

We refer to several definitions. *Slack* is the gap between an allocation's requested size and the next whole hugepage. Virtual address space allocated from the OS is *unbacked* without reserving physical memory. On use, it is *backed*, mapped by the OS with physical memory. We may release memory to the OS once again making it *unbacked*. We primarily pack within hugepage boundaries, but use *regions* of hugepages for packing allocations *across* hugepage boundaries.

From our telemetry of `malloc` usage and TCMALLOC internals, and knowledge of the kernel implementation, we developed several key principles that motivate TEMERAIRE's choices.

1. **Total memory demand varies unpredictably with time, but not every allocation is released.** We have no control over the calling code, and it may rapidly (and repeatedly) modulate its usage; we must be hardened to this. But many allocations on the pageheap are immortal (and it is difficult to predict which they are [30]); any particular allocation might disappear instantly or live forever, and we must deal well with both cases.

2. **Completely draining hugepages implies packing memory at hugepage granularity**. Returning hugepages that aren't nearly-empty to the OS is costly (see section 2). Generating empty/nearly-empty hugepages implies densely packing the *other* hugepages in our binary. Our design must enable densely packing allocations into as few, saturated, bins as possible.

   While we aim to use exclusively hugepage-sized bins, `malloc` must support allocation sizes larger than a single hugepage. These can be allocated normally, but we place smaller allocations into the *slack* of the allocation to achieve high allocation density. Only when small allocations are dominated by *slack* do we need to place large allocations end on end in *regions*.

3. **Draining hugepages gives us new release decision points.** When a hugepage becomes completely empty, we can choose whether to retain it for future memory allocations or return it to the OS. Retaining it until released by TCMALLOC's background thread carries a higher memory cost. Returning it reduces memory usage, but comes at a cost of system calls and page faults if reused. Adaptively making this decision allows us to return memory to the OS faster than the background thread while simultaneously avoiding extra system calls.

4. **Mistakes are costly, but work is not.** Very few allocations directly touch the pageheap, but *all* allocations are backed via the pageheap. We must only pay the cost of allocation once; if we make a bad placement and fragment

a hugepage, we pay either that space or the time-cost of breaking up a hugepage for a long time. It is worth slowing down the allocator, if doing so lets it make better decisions.

Our allocator implements its interface by delegating to several subcomponents, mapped in Figure 6. Each component is built with the above principles in mind, and each specializes its approximation for the type of allocation it handles best. As per principle #4, we emphasize smart placement over speed[4].

While the particular implementation of TEMERAIRE is tied to TCMALLOC internals, most modern allocators share similar large backing allocations of page (or higher) granularity, like TCMALLOC's spans: compare `jemalloc`'s "extents" [20], `Hoard`'s "superblocks" [9], and `mimalloc`'s "pages" [29]. `Hoard`'s 8KB superblocks are directly allocated with 'mmap', preventing hugepage contiguity. Those superblocks could instead be densely packed onto hugepages. `mimalloc` places its 64KiB+ "pages" within "segments," but these are maintained per-thread which hampers dense packing across the segments of the process. Eagerly returning pages to the OS minimizes the RAM cost here, but breaks up hugepages. These allocators could also benefit from a TEMERAIRE-like hugepage aware allocator[5].

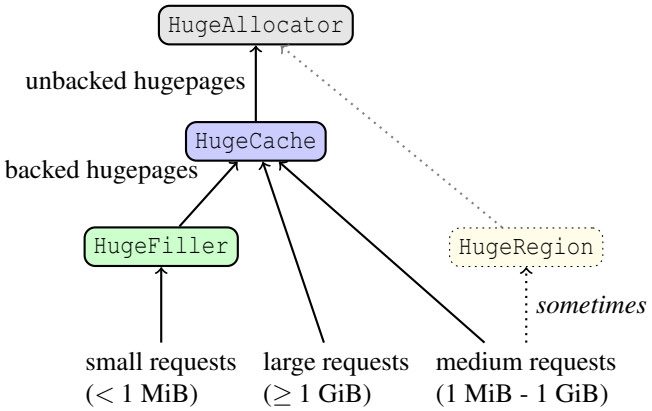


Figure 6: TEMERAIRE's components. Arrows represent the flow of requests to interior components.

4.1 The overall algorithm

We will briefly sketch the overall approach and each component's role, then describe each component in detail. Our goal is to minimize generated slack, and if we do generate slack, to reuse it for other allocations (as with any page-level fragmentation.)

[4]As each operation holds an often-contended mutex, we do maintain reasonable efficiency: most operations are $O(1)$, with care taken to optimize constant factors.

[5]Indeed, `jemalloc` is doing so, based on TEMERAIRE.

```
Span New(N) {
    // Slack is too small to matter
    if (N >= 1 GiB) return HugeCache.New(N);
    // Help bin-pack a single hugepage
    if (N <= 1 MiB) return HugeFiller.New(N);

    if (N < 2 MiB) {
       // If we can reuse empty space, do so
       Span s = HugeFiller.TryAllocate(N);
       if (s != NULL) return s;
    }

    // If we have a region, use it
    Span s = HugeRegion.TryAllocate(N);
    if (s != NULL) return s;

    // We need a new hugepage.
    s = HugeCache.New(N);
    HugeFiller.DonateTail(s);

    return s;
}
```

Figure 7: Allocation flow for subcomponents. Hugepage size is 2 MiB.

Behind all components is the `HugeAllocator`, which deals with virtual memory and the OS. It provides other components with *unbacked* memory that they can back and pass on. We also maintain a cache of *backed*, fully-empty hugepages, called the `HugeCache`.

We keep a list of partially filled single hugepages (the `HugeFiller`) that can be densely filled by subsequent small allocations. Where binpacking the allocations along hugepage boundaries would be inefficient, we implement a specialized allocator (the `HugeRegion`).

TEMERAIRE directs allocation decisions to its subcomponents based on request size with the algorithm in Figure 7. Each subcomponent is optimized for different allocation sizes.

Allocations for an exact multiple of hugepage size, or those sufficiently large that slack is immaterial, we forward directly to the `HugeCache`.

Intermediate sized allocations (between 1MiB and 1GiB) are typically also allocated from the `HugeCache`, with a final step of *donation* for slack. For example, a 4.5 MiB allocation from the `HugeCache` produces 1.5 MiB of slack, an unacceptably high overhead ratio. TEMERAIRE donates that slack to the `HugeFiller` by pretending that the last hugepage of the request has a single "leading" allocation on it (Figure 8).

When such a large span is deallocated, the allocator also marks the fictitious leading allocation as free. If the slack is unused, it is returned to the tail hugepage along with the rest. Otherwise the tail hugepage is left behind in the `HugeFiller` and

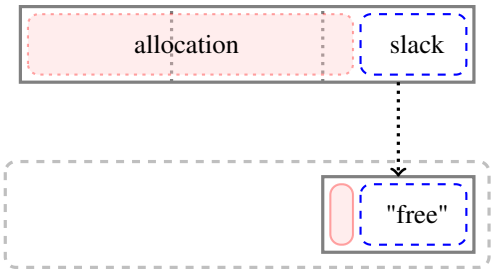


Figure 8: The slack from a large allocation spanning 3 hugepages is "donated" to the `HugeFiller`. The larger allocation's tail is treated as a fictitious allocation.

only the first $N-1$ hugepages are returned to the `HugeCache`.

For certain allocation patterns, intermediate-size allocations produce more slack than we can fill with smaller allocations in strict 2MiB bins. For example, many 1.1MiB allocations will produce 0.9MiB of slack per hugepage (see Figure 12). When we detect this pattern, the `HugeRegion` allocator places allocations across hugepage boundaries to minimize this overhead.

Small requests (<= 1MiB) are always served from the `HugeFiller`. For allocations between 1MiB and a hugepage, we evaluate several options:

1. We *try* the `HugeFiller`: if we have available space there we use it and are happy to fill a mostly-empty page.

2. If the `HugeFiller` can't serve these requests, we next consider `HugeRegion`; if we have regions allocated which can serve the request, we do so. If no region exists (or they're all too full) we consider allocating one, but only, as discussed below, if we've measured high ratios of slack to small allocations.

3. Otherwise, we allocate a full hugepage from the `HugeCache`. This generates *slack*, but we anticipate that it will be filled by future allocations.

We make a design choice in TEMERAIRE to care about external fragmentation up to the level of a hugepage, but essentially not at all past it (but see Section 4.5 for an exception.) For example, a system with a single 1 GiB free range and one with 512 discontiguous free hugepages is handled equally well by TEMERAIRE. In either case, the allocator will (typically) return all of the unused space to the OS; a fresh allocation of 1 GiB will require faulting in memory in either case. In the fragmented scenario, we will need to do so on fresh virtual memory. Waste of virtual address range unoccupied by live allocations and not consuming physical memory is not a concern, since with 64-bit address spaces, virtual memory is practically free.

```
while (true) {
  Delete(New(512KB))
}
```

Figure 9: Program which repeatedly drains a single hugepage.

4.2 HugeAllocator

`HugeAllocator` tracks mapped virtual memory. All OS mappings are made here. It stores hugepage-aligned *unbacked* ranges (i.e. those with no associated physical memory.) Virtual memory is nearly free, so we aim for simplicity and reasonable speed. Our implementation tracks unused ranges with a treap [40]. We augment subtrees with their largest contained range, which lets us quickly select an approximate best-fit.

4.3 HugeCache

The `HugeCache` tracks *backed* ranges of memory at full hugepage granularity. A consequence of the `HugeFiller` filling and draining whole hugepages is that we need to decide when to return empty hugepages to the OS. We will regret returning memory we will need again, and equally regret *not* returning memory that will languish in the cache. Returning memory eagerly means we make syscalls to return the memory and take page faults to reuse it. Releasing memory only at the rate requested by TCMALLOC's periodic release thread means memory is held unused.

Consider the artificial program in Figure 9 with no additional heap allocations. On each iteration of the loop, 'New' requires a new hugepage and places it with the `HugeFiller`. 'Delete' removes the allocation and the hugepage is now completely free. Returning eagerly would require a syscall every iteration for this simple, but pathological program.

We track periodicity in the demand over a 2-second sliding window and calculate the minimum and maximum seen $(demand_{min}, demand_{max})$. Whenever memory is returned to the `HugeCache`, we return hugepages to the OS if the cache would be larger than $demand_{max} - demand_{min}$. We also tried other algorithms, but this one is simple and suffices to capture the empirical dynamics we've seen. The cache is allowed to grow as long as our windowed demand has seen a need for the new size. In oscillating usage, this will (incorrectly) free memory once, then (correctly) keep it from then on. Figure 10 shows our cache size for a Tensorflow workload which rapidly oscillates usage by a large fraction; we track the actually needed memory tightly.

4.4 HugeFiller

The `HugeFiller` satisfies smaller allocations that each fit within a single hugepage. This satisfies the majority of allocations (78% of the pageheap is backed by the `HugeFiller`

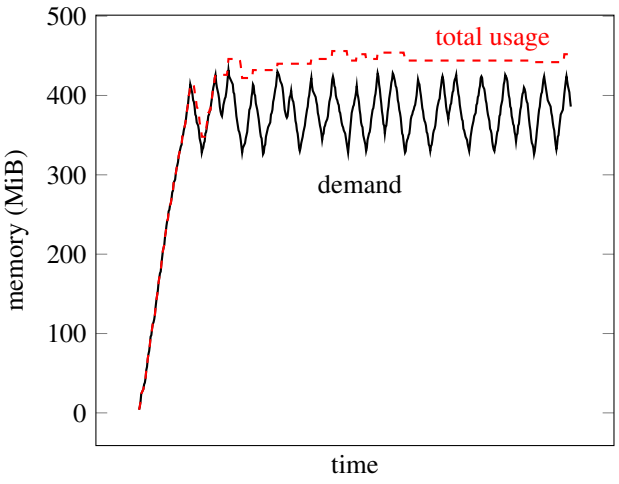


Figure 10: Tensorflow's demand on the `HugeCache` over time, plotted with the cache limit (+demand). Notice that we tightly track their saw-toothed demand the first time it drops. After that, we recognize the pattern and keep the peak demand in cache.

on average across the fleet) and is the most important–and most optimized–component of our system. *Within* a given hugepage, we use a simple (and fast) best-fit algorithm to place an allocation; the challenging part is deciding *which* hugepage to place an allocation on.

This component solves our binpacking problem: our goal is to segment hugepages into some that are kept maximally full, and others that are empty or nearly so. The emptiest hugepages can be reclaimed (possibly breaking up a hugepage as needed) while minimizing the impact on hugepage coverage as the densely-filled pages cover most used memory with hugepages. But it is challenging to empty out hugepages, since we cannot rely on any particular allocation disappearing.

A secondary goal is to minimize fragmentation *within* each hugepage, to make new requests more likely to be served. If the system needs a new K-page span and no free ranges of $\geq K$ pages are available, we require a hugepage from the `HugeCache`. This creates slack of $(2MiB - K * pagesize)$, wasting space.

These give us two goals to prioritize. Since we want to maximize the probability of hugepages becoming totally free, nearly-empty hugepages are precious. Since we need to minimize fragmentation, hugepages with long free ranges are also precious. Both priorities are satisfied by preserving hugepages with the longest free range, as longer free ranges must have fewer in-use blocks. We organize our hugepages into ranked lists correspondingly, leveraging per-hugepage statistics.

Inside each hugepage, we track a bitmap of used pages; to fill a request from some hugepage we do a best-fit search from that bitmap. We also track several statistics:

- the *longest free range* (L), the number of contiguous pages not already allocated,
- the total *number of allocations* (A),
- the total *number of used pages* (U).

These three statistics determine a *priority order* of hugepages to place allocations. We choose the hugepage with the lowest sufficient L and the highest A. For an allocation of K pages, we first consider only hugepages whose longest free range is sufficient ($L \geq K$). This determines whether a hugepage is a *possible* allocation target. Among hugepages with the minimum $L \geq K$, we prioritize by fullness. Substantial experimentation led us to choose A, rather than U.

This choice is motivated by a *radioactive decay-type allocation model* [16] where each allocation, of any size, is equally likely to become free (with some probability p). In this model a hugepage with 5 allocations has a probability of becoming free of $p^5 << p$; so we should very strongly avoid allocating from hugepages with very few allocations. In particular, this model predicts A is a much better model of "emptiness" than U: one allocation of size M is more likely to be deallocated than M allocations of size 1.

The decay model isn't perfectly true in real applications, but it is an effective approximation, and experimentation backs up its primary claim: prioritizing by A empties substantially more pages than prioritizing by U. (In practice, using U produces acceptable results, but meaningfully worse ones.)

In some more detail, A is used to compute a *chunk index* C, given by $min(0, C_{\max} - log_2(A))$. We compute our chunk index so that our fullest pages have $C = 0$ and the emptiest have $C = C_{\max} - 1$. In practice, we have found that $C_{\max} = 8$ chunks are sufficient to avoid allocation from almost-empty pages. Distinguishing hugepages with large counts is less important: For example, we predict a hugepage with 200 allocations and one with 150 as both very unlikely to completely drain. This scheme prioritizes distinguishing gradations among pages that might become empty.

We store hugepages in an array of lists, where each hugepage is stored on the list at index $I = C_{\max} * L + C$. Since a K-page allocation is satisfiable from any hugepage with $L >= K$, the hugepages which can satisfy an allocation are exactly those in lists with $I >= C_{\max} * K$. We pick an (arbitrary) hugepage from the least such nonempty list, accelerating that to constant time with a bitmap of nonempty lists.

Our strategy differs from best fit. Consider a hugepage X with a 3 page gap and a 10 page gap and another hugepage Y with a 5 page gap. Best fit would prefer X. Our strategy prefers Y. This strategy works since we are looking to allocate on the most fragmented page, since fragmented pages are less likely to become entirely free. If we need, say, 3 pages, then pages which contain at most a gap of 3 available pages are more likely to be fragmented and therefore good candidates for allocation. Under the radioactive-decay model, allocations

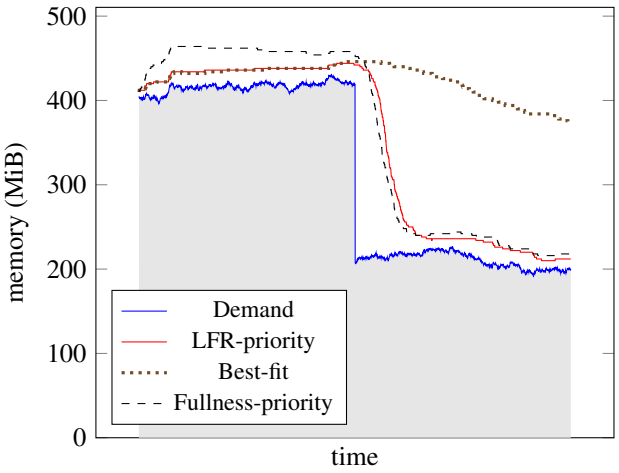


Figure 11: `HugeFiller` with various bin-packing strategies. Best fit is outperformed by prioritizing either fullness or longest free range (LFR); LFR dominates fullness.

near large gaps are as likely as any other to become free, which can cause those gaps to substantially grow; they can then be used for large allocations. We treat that 10-page gap as precious and avoid allocating near it unless nothing else works, which allows it to grow.

Figure 11 demonstrates this in a simple case. We plot the demand on the `HugeFiller` from a synthetic trace (see Section 6.1). We also show the total used memory from three approaches: `HugeFiller`'s actual search, a search that prioritizes fullness over fragmentation (*A* over *L*), and a global best fit. Note that the trace includes a substantial one-time drop, to go with random fluctuations in usage. Our LFR-priority algorithm beats both other approaches. In particular, we see that after the usage drop, best-fit barely recovers any total memory, and finishes with close to 100% overhead, whereas both other algorithms closely match the actual demand.

Surprisingly, this simple strategy substantially outperforms a *global* best fit algorithm–placing a request in the single gap in any hugepage that is closest to its size. Best-fit would be prohibitively expensive—we cannot search 10-100K hugepages for every request, but it's quite counter-intuitive that it also produces higher fragmentation. Best-fit being far from optimal for general fragmentation problems is not a new result [36], but it's interesting to see how poor it can be here.

A last important detail is that donated hugepages are less desirable allocation targets than any non-donated hugepage. Consider the pathological program looping:

```
while (true) {
  // Reserve 51 hugepages + donate tail of last
  L = New(100 MiB + 1 page);
  // Make a small allocation
  S = New(1);
  // Delete large allocation
  Delete(L);
}
```

Each iteration only allocates 1 (net) page, but if we always use the slack from `L` to satisfy `S`, we will end up placing each `S` on its own hugepage. In practice, simply refusing to use donated pages if others are available prevents this, while effectively using slack where it's needed.

4.5 HugeRegion

`HugeCache` (and `HugeAllocator` behind it) suffices for large allocations, where rounding to a full hugepage is a small cost. `HugeFiller` works well for small allocations that can be packed into single hugepages. `HugeRegion` helps those between.

Consider a request for 1.1 MiB of memory. We serve it from the `HugeFiller`, leaving 0.9 MiB of unused memory from the 2MiB hugepage: the *slack* space. The `HugeFiller` assumes that slack will be filled by future small (<1MiB) allocations, and typically it is: our observed byte ratio of fleet-wide small allocations to slack is 15:1. In the limit we can imagine a binary that requests literally nothing but 1.1 MiB spans in Figure 12.

The `HugeRegion` deals with this problem, which is to some extent *caused* by our own choices. We focus heavily on packing allocations into hugepage-sized bins with the `HugeFiller`, and our desire to do that with donated slack is catastrophic with some allocation patterns. Most normal binaries are of course fine without it, but a general purpose memory allocator needs to handle diverse workloads, even those dominated by slack-heavy allocations. Clearly, we must be able to allocate these lying across hugepage boundaries. `HugeRegion` neatly eliminates this pathological case.

A `HugeRegion` is a large fixed-size allocation (currently 1 GiB) tracked at small-page granularity with the same kind of bitmaps used by individual hugepages in the `HugeFiller`. As with those single hugepage ranges, we best-fit any request across all pages in the region. We keep a list of these regions, ordered by longest free range, for the same reason as `HugeFiller`. Allocating from these larger bins immediately allows large savings in wasted space: rather than losing 0.9 MiB/hugepage in our pessimal load, we lose 0.9 MiB per

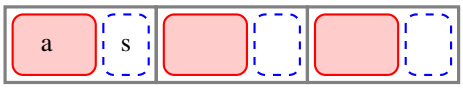


Figure 12: Slack ("s") can accumulate when many allocations ("a") are placed on single hugepages. No single slack region is large enough to accommodate a subsequent allocation of size "a."

HugeRegion, only about 0.1%. (This motivates the large size of each region.)

Most programs don't need regions at all. We do not allocate any region until we've accumulated large quantities of slack that are larger than the total of the program's small allocations. Fleetwide, only 8.8% of programs trigger usage of regions, but the feature is still important: 53.2% of allocations in those binaries are served from regions. One such workload is a key-value store that loads long-lived data in large chunks into memory and makes a small number of short-lived small allocations for serving requests. Without regions, the request-related allocations are unable to fill the slack generated by the larger allocations. This technique prevents this slack-heavy uncommon allocation pattern from bloating memory use.

4.6 Memory Release

As discussed above, `Release(N)` is invoked periodically by support threads at a steady trickle.

To implement our interface's `Release(N)` methods, TEMERAIRE typically just frees hugepage ranges from `HugeCache` and possibly shrinks its limit as described above. Releasing more than the hinted N pages is not a problem; the support threads use the actual released amount as feedback, and adjust future calls to target the correct overall rate.

If the `HugeCache` cannot release N pages of memory, the `HugeFiller` will subrelease just the free (small) pages on the emptiest hugepage.

Returning small pages from partially filled hugepages ("subreleasing" them) is the last resort for reducing memory footprints as the process is largely irreversible[6]. By returning some but not all small pages on a hugepage, we cause the OS to replace the single page table entry spanning the hugepage with small entries for the remaining pages. This one-way operation, through increased TLB misses, slows down accesses to the remaining memory. The Linux kernel will use small pagetable entries for the still-used pages, even if we re-use the released address space later. We make these return decisions in the `HugeFiller`, where we manage partially filled hugepages.

The `HugeFiller` treats the subreleased hugepages separately: we do not allocate from them unless no other hugepage is usable. Allocations placed on this memory will not benefit from hugepages, so this helps performance and allows these partially released hugepages to become completely empty.

5 Evaluation of TEMERAIRE

We evaluated TEMERAIRE on Google's WSC workloads. The evaluation was concerned with several metrics, including both CPU and memory savings. We present evaluations of TEMERAIRE on several key services, measuring 10% of cycles and 15% of RAM usage in our WSC. In section 6.4 we discuss workload diversity; in this evaluation we examine data across all workloads using our experimental framework and fleetwide-profiler telemetry. We've argued for prioritizing workload efficiency over the attributable cost of `malloc`; we therefore examine IPC metrics (as a proxy for user throughput) and where possible, we obtained application-level performance metrics to gauge workload productivity (e.g., requests-per-second per core) on our servers. We present longitudinal data from the rollout of TEMERAIRE to all TCMALLOC users in our fleet.

Overall, TEMERAIRE proved a significant win for CPU and memory.

5.1 Application Case Studies

We worked with performance-sensitive applications to enable TEMERAIRE in their production systems, and measure the effect. We summarize the results in Table 1. Where possible, we measured each application's user-level performance metrics (throughput-per-CPU and latency). These applications use roughly 10% of cycles and 15% of RAM in our WSC.

Four of these applications (`search1`; `search2`; `search3`; and `loadbalancer`) had previously turned off the periodic memory release feature of TCMALLOC. This allowed them to have good hugepage coverage, even with the legacy pageheap's hugepage-oblivious implementation, at the expense of memory. We did not change that setting with TEMERAIRE. These applications maintained their high levels of CPU performance while reducing their total memory footprint.

With the exception of Redis, all of these applications are multithreaded. With the exception of `search3`, these workloads run on a single NUMA domain with local data.

- Tensorflow [1] is a commonly used machine learning application. It had previously used a high periodic release rate to minimize memory pressure, albeit at the expense of hugepages and page faults.
- `search1`, `search2`, `ads1`, `ads2`, `ads4`, `ads5` receive RPCs and make subsequent RPCs of their own other services.
- `search3`, `ads3`, `ads6` are leaf RPC servers, performing read-mostly retrieval tasks.
- Spanner [17] is a node in a distributed database. It also includes an in-memory cache of data read from disk which adapts to the memory provisioned for the process and unused elsewhere by the program.
- `loadbalancer` receives updates over RPC and periodically publishes summary statistics.

[6] While the THP machinery may reassemble hugepages, it is non-deterministic and dependent on system utilization. There is a negative feedback loop here where high-utilization scenarios actually compete with and impede THP progress that might benefit them the most.

Application	Throughput	Mean Latency	RSS (GiB)	RAM change	IPC		dTLB Load Walk (%)		malloc (% of cycles)		Page Fault (% of cycles)	
					Before	After	Before	After	Before	After	Before	After
Tensorflow [1]	+26%		8.4	-8.7%	1.33±0.04	1.43±0.02	9.5±0.6	9.0±0.6	5.9±0.09	5.9±0.12	0.005±0.003	0.131±0.071
search1 [6, 18]†			3.7	-20%	1.28±0.01	1.29±0.01	10.3±0.2	10.2±0.1	4.37±0.05	4.38±0.02	0.003±0.003	0.032±0.002
search2†			234	-7%	1.64±0.02	1.67±0.02	8.9±0.1	8.9±0.3	3.2±0.02	3.3±0.04	0.001±0.000	0.005±0.001
search3 †			4.8	-6.9%	0.77±0.02	0.84±0.01	38.1±1.3	15.9±0.3	2.3±0.04	2.7±0.05	0.012±0.003	0.011±0.002
ads1	+2.5%	-14%	5.6	-6.5%	1.12±0.01	1.22±0.01	27.4±0.4	10.3±0.2	2.7±0.03	3.5±0.08	0.022±0.001	0.047±0.001
ads2	+3.4%	-1.7%	50.6	-0.8%	1.36±0.01	1.43±0.01	27.1±0.5	11.6±0.2	2.9±0.04	3.2±0.03	0.067±0.002	0.03±0.003
ads3	+0.5%	-0.2%	2.5	-1.7%	0.87±0.01	0.93±0.01	28.5±0.9	11.1±0.3	4.2±0.05	4.9±0.04	0.022±0.001	0.008±0.001
ads4	+6.6%	-1.1%	10.0	-1.1%	1.16±0.02	1.16±0.02	21.9±1.2	16.7±2.4	3.6±0.08	3.8±0.15	0.018±0.002	0.033±0.007
ads5	+1.8%	-0.7%	53.5	-2.3%	1.40±0.02	1.59±0.03	33.6±2.4	17.8±0.4	13.5±0.48	9.9±0.07	0.037±0.012	0.048±0.067
ads6	+15%	-10%	7.0		1.55±0.30	1.70±0.14	31.0±4.3	15.7±1.8	3.1±0.88	3.0±0.24	0.025±0.08	0.024±0.01
Spanner [17]	+6.3%		1.4	-40%	1.38±0.12	1.39±0.28	19.6±1.2	9.5±4.5	11.5±0.60	10.7±0.46	0.094±0.06	0.057±0.062
loadbalancer†				-7.9%	1.26	1.33	23.3	12.4	5.2	5.0	0.058	0.112
Average (all WSC apps)	+5.2%											
Redis†	+0.75%											
Redis	+0.44%											

Table 1: Application experiments from enabling TEMERAIRE. Throughput is normalized for CPU. †: Applications' periodic memory release turned off. dTLB load walk (%) is the fraction of cycles spent page walking, not accessing the L2 TLB. malloc (% of cycles) is the *relative* amount of time in allocation and deallocation functions. 90%th confidence intervals reported.

- Redis is a popular, open-source key-value store. We evaluated the performance of Redis 6.0.9 [42] with TEMERAIRE, using TCMALLOC's legacy page heap as a baseline. These experiments were run on servers with Intel Skylake Xeon processors. Redis and TCMALLOC were compiled with LLVM built from Git commit 'cd442157cf' using '-O3'. In each configuration, we ran 2000 trials of 'redis-benchmark', with each trial making 1000000 requests to push 5 elements and read those 5 elements.

For the 8 applications with periodic release, we observed a mean CPU improvement of 7.7% and a mean RAM reduction of 2.4%. Two of these workloads did not see memory reductions. TEMERAIRE's HugeCache design handles Tensorflow's allocation pattern well, but cannot affect its bursty demand. Spanner maximizes its caches up to a certain memory limit, so reducing TCMALLOC's overhead meant more application data could be cached within the same footprint.

5.2 Fleet experiment

We randomly selected 1% of the machines distributed throughout our WSCs as an experiment group and a separate 1% as a control group (see section 6.4). We enabled TEMERAIRE on all applications running on the experiment machines. The applications running on control machines continued to use the stock pageheap in TCMALLOC.

Our fleetwide profiler lets us correlate performance metrics against the groupings above. We collected data on memory usage, hugepage coverage, overall IPC, and TLB misses. At the time of the experiment, application-level performance metrics (throughput-per-CPU, latency) were not collected. In our analysis, we distinguish between applications that periodically release memory to the OS and those that turn off this feature to preserve hugepages with TCMALLOC's prior non-hugepage-aware pageheap. Figure 13 shows that TEMERAIRE improved hugepage coverage, increasing the percentage of heap memory backed by hugepages from 11.8% to 23% for applications periodically releasing memory and from 44.3% to 67.3% for applications not periodically releasing memory.

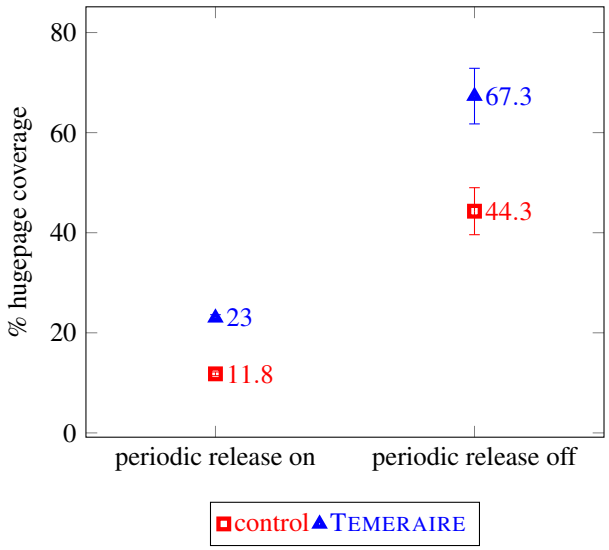


Figure 13: Percentage of heap memory backed by hugepages during fleet experiment and 90%th confidence interval. (Error bars in "release on" condition are too small to cleanly render.)

We observed a strong improvement even in the case that periodic release was disabled. Since these binaries do not break up hugepages in either configuration, the benefit is derived from increased system-wide availability of hugepages (due to reduced fragmentation in other applications). TEMERAIRE improves this situation in two ways: since we aggressively release empty hugepages (where the traditional pageheap does not), we consume fewer hugepages that we do not need, allowing other applications to more successfully request them, and other co-located applications are no longer breaking up hugepages at the same rate. Even if we map large aligned regions of memory and do not interfere with transparent hugepages, the kernel cannot always back these with hugepages [26, 33]. Fragmentation in physical memory can limit the number of available hugepages on the system.

We next examine the effect this hugepage coverage had

Periodic Release	Walk Cycles (%)		MPKI	
	Control	Exp.	Control	Exp.
On	12.5	11.9 (**-4.5%**)	1.20	1.14 (**-5.4%**)
Off	14.1	13.4 (**-5%**)	1.36	1.29 (**-5.1%**)

Table 2: dTLB load miss page walk cycles as percentage of application usage and dTLB misses per thousand instructions (MPKI) without TEMERAIRE (Control) TEMERAIRE enabled (Exp.)

on TLB misses. Again, we break down between apps that enable and disable periodic memory release. We measure the percentage of total cycles spent in a dTLB load stall[7].

We see reductions of 4.5-5% of page walk miss cycles (Table 2). We see in the experiment data that apps not releasing memory (which have better hugepage coverage) have higher dTLB stall costs, which is slightly surprising. Our discussions with teams managing these applications is that they turn off memory release because they *need* to guarantee performance: on average, they have more challenging memory access patterns and consequently greater concerns about microarchitectural variance. By disabling this release under the prior implementation, they observed better application performance and fewer TLB stalls. With TEMERAIRE, we see our improved hugepage coverage leads to materially lower dTLB costs for *both* classes of applications.

For our last CPU consideration, we measured the overall impact on IPC[8]. Fleetwide overall IPC in the control group was $0.796647919 \pm 4e{-}9$; in the experiment group, $0.806301729 \pm 5e{-}9$ instructions-per-cycle. This 1.2% improvement is small in relative terms but is a large absolute savings (especially when considered in the context of the higher individual application benefits discussed earlier).

For memory usage, we looked at *pageheap overhead*: the ratio of backed memory in the pageheap to the total heap memory in use by the application. The experiment group decreased this from 15.0% to 11.2%, again, a significant improvement. The production experiments comprise thousands of applications running continuously on many thousands of machines, conferring high confidence in a fleetwide benefit.

5.3 Full rollout trajectories

With data gained from individual applications and the 1% experiment, we changed the default[9] behavior to use TEMERAIRE. This rolled out to 100% of our workloads gradually [10, 38].

Over this deployment, we observed a reduction in cycles stalled on TLB misses (L2 TLB and page walks) from 21.6% to 20.3% (6% reduction) and a reduction in pageheap overhead from 14.3% to 10.6% (26% reduction). Figure 14 shows the effect on TLB misses over time: at each point we show the total percentage of cycles attributable to TLB stalls (load and store), broken down by pageheap implementation. As TEMERAIRE rolled out fleetwide, it caused a noticeable downward trend.

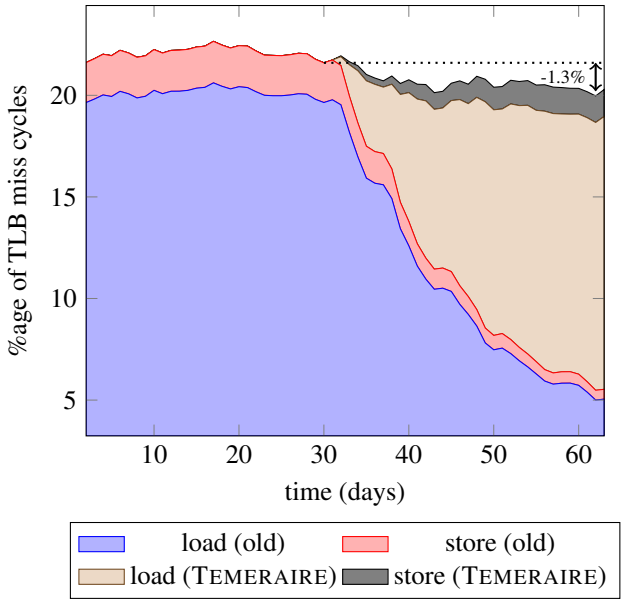


Figure 14: Stacked line graph showing effect of TEMERAIRE rollout on TLB miss cycles. We see an overall downward trend from 21.6% to 20.3% as TEMERAIRE became a larger fraction of observed usage in our WSC.

Figure 15 shows a similar plot of pageheap overhead. We see another significant improvement. Hugepage optimization has a natural tradeoff between space and time here; saving the maximum memory possible requires breaking up hugepages, which will cost CPU cycles. But TEMERAIRE outperforms the previous design in *both* space and time. We highlight several conclusions from our data:

Application productivity outpaced IPC. As noted above and by Alameldeen et al. [3], simple hardware metrics don't always accurately reflect application-level benefits. By all indication, TEMERAIRE improved application *metrics* (RPS, latencies, etc.) by *more* than IPC.

Gains were not driven by reduction in the cost of `malloc`. Gains came from accelerating user code, which was sometimes drastic–in both directions. One application (`ads2`) saw an increase of `malloc` cycles from 2.7% to 3.5%, an apparent regression, but they reaped *improvements* of 3.42% RPS, 1.7% latency, and 6.5% peak memory usage.

There is still considerable headroom, and small percentages matter. Even though TEMERAIRE has been successful, hugepage coverage is still only 67% when using TEMERAIRE

[7] More precisely cycles spent page walking, not accessing the L2 TLB.

[8] Our source of IPC data is not segmented by periodic background memory release status.

[9] This doesn't imply, quite, that every binary uses it. We allow opt outs for various operational needs.

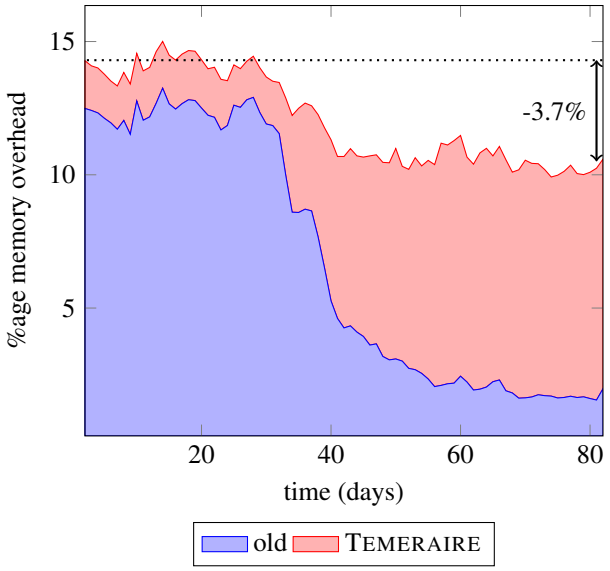


Figure 15: Stacked line graph showing effect of TEMERAIRE rollout on pageheap overhead. Total memory overhead goes from 14.3% to 10.6%, as TEMERAIRE became a larger fraction of observed usage in our WSC by growing from a handful of applications (section 5.1) to nearly all applications.

without subrelease due to physical memory contiguity limitations. Increasing to 100% would significantly improve application performance.

6 Strategies used in building TEMERAIRE

It is difficult to predict the best approach for a complex system a priori. Iteratively designing and improving a system is a commonly used technique. Military pilots coined the term "OODA (Observe, Orient, Decide, Act) loop" [13] to measure a particular sense of reaction time: seeing incoming data, analyzing it, making choices, and acting on those choices (producing new data and continuing the loop). Shorter OODA loops are a tremendous tactical advantage to pilots and accelerate our productivity as well. Optimizing *our* own OODA loop–how quickly we could develop insight into a design choice, evaluate its effectiveness, and iterate towards better choices–was a crucial step in building TEMERAIRE.

While our final evaluation was driven by execution on our production servers, this was both too disruptive and too risky to test intermediate ideas; however, `malloc` microbenchmarks are also not particularly interesting at the page level. To address these challenges, we generated traces to drive the development of TCMALLOC in two ways.

6.1 "Empirical" distribution sampling

Our production fleet implements a fleet wide profiler [35]. Among the data collected by this profiler are fleet-wide samples of `malloc` tagged with request size and other useful properties. We collect a sample of currently-live data in our heap and calls to `malloc`. From these samples we can infer the empirical distribution of size both for live objects and malloc calls. Our *empirical driver* generates calls to `malloc` and `free` as a Poisson process[10] that replicates these distributions, while also targeting an arbitrary (average) heap size. That target size can be changed over simulated time, reproducing factors such as diurnal cycles, transient usage, or high startup costs. We have made this driver and its inputs available on Github (see Section 9).

Despite the name "empirical driver," this remains a *highly* unrealistic workload: every allocation (of a given size) is equally likely to be freed at any timestep, and there is no correlation between the sizes of consecutive allocation. Neither does it reproduce per-thread or per-CPU dynamics. Nevertheless, the empirical driver is a fast, efficient way to place `malloc` under an extremely challenging load that successfully replicates many macro characteristics of real work.

6.2 Heap tracing

Tracing every call to `malloc` without the instrumentation overhead perturbing the workload itself is extremely difficult, even infeasible over long timescales. Typical applications can make millions of calls to `malloc` per second. Even if tracing was accomplished non-disruptively, replaying these traces back accurately into a memory allocator in real time *or faster* is similarly intractable: it's difficult to force the right combinations of threads to allocate, access, and free the right buffers on the right CPU at the right (relative) time.

Fortunately, tracing the *pageheap* is considerably easier. It is a single-threaded allocator, only invoked by a small fraction of requests. Playback is also simple–our abstractions allow directly instantiating and manipulating our pageheap representation, rather than going through `malloc()` itself. Traces taken from both real binaries and, surprisingly, the empirical driver itself, played a major role in developing TEMERAIRE.

TEMERAIRE's components serve a request for K pages with memory at address $[p, p+K)$, but never read or write that memory range. We built this for unit testing–allowing the test of corner cases such as 64 GiB of allocations without actually needing 64 GiB of memory–but this is also crucial to accelerating simulations. What might take hours with the empirical driver can be played back in minutes.

[10] Little's law tells us that the average number of live objects L is equal to the product of the arrival rate λ and average lifetime W. To replicate a given distribution of live/allocation object sizes where p_a of live objects have size a, we set $W_a = \frac{c \cdot p_a}{\lambda_a}$. ($c$ is a scaling parameter that determines the total heap size.)

6.3 Telemetry

Beyond producing numbers motivating and evaluating our work, our fleetwide profiler is itself a powerful tool for *designing* allocators. It reveals patterns of allocation we can use to derive heuristics, it allows validation of hypotheses about typical (or even possible) behavior, it helps identify which patterns we can safely ignore as unimportant and which we must optimize. Besides being used in obvious ways–such as tuning cache sizes to fit typical use or determining thresholds for "small" allocations based on the CDF of allocations–querying the profiler was our first step *whenever* we were unsure of useful facts. We gained confidence that our approach to filling slack (see section 4.5) worked on diverse workloads by querying the profiler for ratios of page allocation sizes. Providing large scale telemetry that can be consumed by data analysis tools makes it easy to test and eliminate hypotheses. Such "tiny experiments" [8] lead to better designs.

This reflects a cultivated mindset in identifying new telemetry. Our first question for any new project is "What metrics should we add to our fleetwide profiler?" We continually expose more of the allocator's internal state and derived statistics, such as cache hit rates. While we can form some hypotheses using traditional loadtests, this technique helps validate their generality.

6.4 Experiment framework

We have also developed an experiment framework allowing us to A/B test implementations or tuning choices across our fleet at scale. We can enable or disable experiment groups across a small percentage of *all* our machines, without requiring product teams running services on those machines to take any action. A/B testing is not a new approach, but enabling it at the scale of our WSC is a powerful development tool.

As discussed above, our A/B experiment for TEMERAIRE demonstrated improved hugepage coverage, even for jobs that never released memory. This is an example of an effect–against neighboring, collocated services–that might go unnoticed during the test of an individual service.

We've observed two noteworthy advantages to A/B experimentation:

- Reduced cost and uncertainty associated with major behavioral changes. Small 1% experiments can uncover latent problems well before we roll new defaults, at far less cost [10, Appendix B].
- Reduced likelihood of overfitting to easily tested workloads. Tuning for production-realistic loadtests, while great for the applications they represent, can result in non-ideal results for other workloads. Instead, we can be confident our optimization is good on average for everyone, and detect (and fix) applications that see problems.

Experiments allow us to evaluate changes on diverse workloads. Kanev, et. al. [24] proposed prefetching the next object $i+1$ when `malloc` is returning object i from its freelists. Effective prefetches need to be *timely* [28]. Too early and data can be evicted from the cache before use. Too late and the program waits. In this case, prefetching object i when returning it, turns out to be too late: User code will write to the object within a few cycles, far sooner than the prefetch's access to main memory can complete. Prefetching object $i+1$ gives time for the object to be loaded into the cache by the time the next allocation occurs. Independent of the experiments to develop TEMERAIRE, we added this *next* object prefetch for TCMALLOC usage in our WSC despite the contrarian evidence that it appears to slowdown microbenchmarks and increases apparent `malloc` cost. We were able to still identify this benefit thanks to the introspective techniques described here, allowing us to prove that application performance was improved at scale in our WSC; both unlocking important performance gains and proving the generality of these macro approaches.

7 Future Work

Peak vs. average. A job quickly oscillating between peak and trough demand cannot be usefully binpacked against its average. Even if the allocator could instantaneously return unused memory, job schedulers could not make use of it before it was required again. Thus transient overhead is not a practical opportunity [43]. This guides us to measure how overhead changes over time, which can motivate slower release rates [31] or application of compaction techniques (such as Mesh [34]).

Intermediate caches / exposed free spans. TCMALLOC's design of stacked caches makes for direct optimization and is highly scalable, but hides useful cross-layer information. A good example comes from Bigtable at Google [14]. Cached ranges are 8 KiB `malloc`'d segments (i.e. one TCMALLOC page) to avoid fragmentation. Meaning, most freed buffers won't make it past the local cache or central freelist; only when a full span's worth is simultaneously freed (and somehow pushed out of TCMALLOC's local cache) do these freed buffers get returned to the pageheap. If every alloc/free of these chunks were visible to the pageheap, we'd be able to reduce fragmentation–we'd have a much more precise estimate of available space within each hugepage. Of course, if every `malloc(8192)/free` went to the pageheap, we would also eliminate all scalability! There must be a middle ground. Can we expose the contents of frontline caches to the pageheap and reduce fragmentation?

Upfront costs / amortization / prediction. The fact we cannot anticipate what `Delete()` calls will come in the future is the hardest part of building a hugepage-friendly algorithm. We try to generate empty hugepages through heuristics and hope: we aim to have mostly-empty things stay that way and hope that the final allocations will quickly get freed. But some allocations are likely immortal–common data structures that

are used throughout the program's run, or frequently used pages that will bounce in and out of local caches.

We can improve allocation decisions when we know–immortal or not–they will be hot and see frequent access. Ensuring these allocations are placed onto hugepages provides larger marginal performance benefit. TLB misses occur on access, so it may be preferable to save memory rather than improve access latency to colder allocations.

Far memory cooperation "Far memory" [27] allows us to move data to slower, but less expensive memory, reducing DRAM costs. Clustering rarely accessed allocations can make far memory more effective. More overhead can be afforded on those decisions since they can't happen very often. Avenues like machine learning [30] or profile directed optimization [15, 37] show promise for identifying these allocations.

Userspace-Kernel Cooperation TEMERAIRE places memory in a layout designed to be compatible with kernel hugepage policy (Section 2), but this is only an implicit cooperation. Kernel APIs which prioritize the allocation of hugepages within an address space or across processes would enable proactive management of which regions were hugepage-backed, versus the current best-effort reactive implementation.

In developing TEMERAIRE, we considered but did not deploy an interface to request a memory region be immediately repopulated with hugepages. TEMERAIRE primarily tries to avoid breaking up hugepages altogether as the existing THP machinery is slow to reassemble them (Section 4.6). Being able to initiate on-demand repopulation would allow an application to resume placing allocations in that address space range without a performance gap.

A common problem today is that the first applications to execute on a machine are able to claim the majority of hugepages, even if higher priority applications are subsequently assigned. We ultimately imagine that such a management system might execute as an independent user daemon, cooperating with individual applications. Kernel APIs could allow hugepages to be more intelligently allocated against a more detailed gradient of priority, benefit, and value.

8 Related work

Some work has optimized malloc for cache efficiency of user-level applications. To minimize L1 conflicts, Dice [19] proposed jittering allocation sizes. Similarly, a cache-index-aware allocator [2] reduces conflict misses by changing relative placement of objects inside pages. `mimalloc` [29] tries to give users objects from the same page, increasing the locality.

Addressing this at the kernel level alone would face the same fragmentation challenges and be more difficult to handle because we have less control over application memory usage. The kernel can back the memory region with a hugepage, but if the application does not densely allocate from that hugepage, memory is wasted by fragmentation. Prior work has examined the kernel side of this problem: Kwon et. al. [26] proposed managing memory contiguity as a resource at the kernel level. Panwar et. al. [32] observed memory bloat from using the Linux's transparent hugepage implementation, due to insufficient userspace level packing.

Optimization of TLB usage in general has been discussed extensively; Basu [7] suggested resurrecting segments to avoid it entirely, addressing TLB usage at the architectural level. CoLT [33] proposed variable-size hugepages to minimize the impact of fragmentation. Illuminator [5] improves page decisions in the kernel to reduce physical memory fragmentation. Ingens [26] attempts to fairly distribute a limited supply of kernel-level hugepages and HawkEye [32] manages kernel allocation of hugepages to control memory bloat. Kernel-based solutions can be defeated by hugepage-oblivious user allocators that return partial hugepages to the OS and fail to densely pack allocations onto hugepages.

At the `malloc` level, SuperMalloc [25] considers hugepages, but only for very large allocations. MallocPool [22] uses similar variable-sized TLBs as CoLT [33] but does not attempt to used fixed-size hugepages. LLAMA [30] studies a possible solution using lifetime predictions, but solutions with practical costs remain open problems.

9 Conclusion

In warehouse scale computers, TLB lookup penalties are one of the most significant compute costs facing large applications. TEMERAIRE optimizes the whole WSC by changing the memory allocator to make hugepage-conscious placement decisions while minimizing fragmentation. Application case studies of key workloads from Google's WSCs show RPS/CPU increased by 7.7% and RAM usage decreased by 2.4%. Experiments at fleet scale and longitudinal data during the rollout at Google showed a 6% reduction in cycles spent in TLB misses, and 26% reduction in memory wasted due to fragmentation. Since the memory system is the biggest bottleneck in WSC applications, there are further opportunities to accelerate application performance by improving how the allocator organizes memory and interacts with the OS.

Acknowledgments

Our thanks to our shepherd Tom Anderson for his help improving this paper. We also thank Atul Adya, Sanjay Ghemawat, Urs Hölzle, Arvind Krishnamurthy, Martin Maas, Petros Maniatis, Phil Miller, Danner Stodolsky, and Titus Winters, as well as the OSDI reviewers, for their feedback.

Availability

The code repository at https://github.com/google/tcmalloc includes TEMERAIRE. It also includes the empirical driver (6.1) and its input parameters (CDF of allocation sizes).

References

[1] Martin 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, 2016.

[2] Yehuda Afek, Dave Dice, and Adam Morrison. Cache Index-Aware Memory Allocation. *SIGPLAN Not.*, 46(11):55–64, June 2011.

[3] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, 2006.

[4] Andrea Arcangeli. Transparent hugepage support. 2010.

[5] Aravinda Prasad Ashish Panwar and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, 2018.

[6] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[7] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 237–248, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Jon Bentley. Tiny Experiments for Algorithms and Life. In *Experimental Algorithms*, pages 182–182, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[9] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[10] Jennifer Petoff Betsy Beyer, Chris Jones and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc, 2016.

[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 25–36, New York, NY, USA, 2004. Association for Computing Machinery.

[12] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 15–33, USA, 2001. USENIX Association.

[13] John R. Boyd. Patterns of Conflict. 1981.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, New York, NY, USA, 2016.

[16] William D. Clinger and Lars T. Hansen. Generational Garbage Collection and the Radioactive Decay Model. *SIGPLAN Not.*, 32(5):97–108, May 1997.

[17] 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)*, Hollywood, CA, 2012.

[18] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 1–1, New York, NY, USA, 2009.

[19] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. The Influence of Malloc Placement on TSX Hardware Transactional Memory. *CoRR*, abs/1504.04640, 2015.

[20] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*, 2006.

[21] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 92–98, 2011.

[22] M. Jägemar. Mallocpool: Improving Memory Performance Through Contiguously TLB Mapped Memory. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1127–1130, 2018.

[23] Svilen Kanev, Juan Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2014.

[24] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating Memory Allocation. *SIGARCH Comput. Archit. News*, 45(1):33–45, April 2017.

[25] Bradley C. Kuszmaul. Supermalloc: A Super Fast Multithreaded Malloc for 64-Bit Machines. *SIGPLAN Not.*, 50(11):41–55, June 2015.

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[27] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization - TACO*, 9:1–29, 03 2012.

[29] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free List Sharding in Action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[30] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[31] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 258–269, USA, 2012. IEEE Computer Society.

[34] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 333–346, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, pages 65–79, 2010.

[36] John Robson. Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies. *Comput. J.*, 20:242–244, 08 1977.

[37] Joe Savage and Timothy M. Jones. HALO: Post-Link Heap-Layout Optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 94–106, New York, NY, USA, 2020. Association for Computing Machinery.

[38] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous Deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International*

Conference on Software Engineering Companion (ICSE-C), pages 21–30, 2016.

[39] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable Locality-Conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, page 84–94, New York, NY, USA, 2006. Association for Computing Machinery.

[40] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

[41] 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*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Redis Team. Redis 6.0.9 and 5.0.10 are out.

[43] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[44] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

Scalable Memory Protection in the PENGLAI Enclave

Erhu Feng[1†‡], *Xu Lu*[1†‡], *Dong Du*[†‡], *Bicheng Yang*[†‡], *Xueqiang Jiang*[†‡], *Yubin Xia*[†§‡], *Binyu Zang*[†§‡], *Haibo Chen*[†§‡]

[†]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

[§]*Shanghai AI Laboratory*

[‡]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Abstract

Secure hardware enclaves have been widely used for protecting security-critical applications in the cloud. However, existing enclave designs fail to meet the requirements of scalability demanded by new scenarios like serverless computing, mainly due to the limitations in their secure memory protection mechanisms, including static allocation, restricted capacity and high-cost initialization. In this paper, we propose a software-hardware co-design to support dynamic, fine-grained, large-scale secure memory as well as fast-initialization. We first introduce two new hardware primitives: 1) *Guarded Page Table* (GPT), which protects page table pages to support page-level secure memory isolation; 2) *Mountable Merkle Tree* (MMT), which supports scalable integrity protection for secure memory. Upon these two primitives, our system can scale to thousands of concurrent enclaves with high resource utilization and eliminate the high-cost initialization of secure memory using fork-style enclave creation without weakening the security guarantees.

We have implemented a prototype of our design based on PENGLAI [24], an open-sourced enclave system for RISC-V. The experimental results show that PENGLAI can support 1,000s enclave instances running concurrently and scale up to 512GB secure memory with both encryption and integrity protection. The overhead of GPT is 5% for memory-intensive workloads (e.g., Redis) and negligible for CPU-intensive workloads (e.g., RV8 and Coremarks). PENGLAI also reduces the latency of secure memory initialization by three orders of magnitude and gains 3.6x speedup for real-world applications (e.g., MapReduce).

1 Introduction

There has been a surge of interest in using enclaves like Intel SGX [77], AMD SEV [12] and ARM TrustZone [58] to host security-critical applications in cloud with minimal reliance on the trust of cloud providers [28,29,37,40,46,59,84,87,96]. Meanwhile, microservice [80] and serverless computing [1, 3–5] have become emerging paradigms of cloud, which use *single-purpose service* or *function* as a basic computation unit and achieve high scalability. Since the frameworks of both are also managed by the cloud providers, it is natural to use enclaves to protect serverless-like and microservice-like applications in the cloud [36,93].

However, existing enclave systems cannot well fit some inherent characteristics of these cloud applications, including resilient memory allocation [86], high resource utilization [43, 76], auto-scaling [43] and ephemeral execution time [34,56, 64], mainly due to three scalability limitations of their secure memory protection mechanisms:

Limitation-1. Non-scalable memory partition/isolation: Most existing enclave systems use static or almost-static partition for region-based memory isolation, like fixed-sized PRM (Processor Reserved Memory) in SGX [15], limited secure world memory regions in ARM TrustZone [2][2], 16 protected memory regions in Keystone [13,68], etc. It is hard to dynamically adjust the boundaries of partitions and scale to a large amount of secure memory. Region-based isolation also violates the scalability requirement of fined-grained memory management in the cloud.

Limitation-2. Non-scalable memory integrity protection: Using a traditional Merkle hash tree (or its variants) to protect memory integrity is hard to scale. For example, Intel SGX only supports 128/256MB EPC (Enclave Page Cache)[3]. Although SGX has no restriction on the number of enclaves, running thousands of enclaves may either cause little available EPC for each instance or frequent EPC swapping, which fails to meet the scalability requirement of serverless computing.

Limitation-3. Non-scalable secure memory initialization: High-cost secure memory initialization causes long startup latency for enclaves, which significantly affects the performance of auto-scaling. For example, SGX needs seconds to create an enclave [40,69] by adding memory to EPC and measuring the contents (by EADD and EEXTEND instructions) for every page. On the contrary, serverless functions usually have a very short life cycle (<1s) [50,88].

In this paper, we propose scalable secure memory protection mechanisms for enclaves with three metrics: (1) size and granularity of secure memory, (2) number of enclaves, (3)

[1]The two authors contributed equally to this work and should be considered co-first authors.

[2]The number depends on specific implementations, and is typically 8.

[3]The latest SGX platforms (Ice Lake Server) support larger EPC sizes (e.g., 1TB) [22], but they do so by giving up on integrity protection.

startup latency of enclaves. We first introduce two new architectural primitives, *Guarded Page Table* (GPT) and *Mountable Merkle Tree* (MMT): GPT protects page table pages and enables memory isolation with page-level granularity, and MMT is a new abstraction to achieve on-demand and scalable memory encryption and integrity protection. Leveraging these two primitives, a lightweight *secure monitor* running in the most privileged mode is in charge of enclave management and maintaining security guarantees. To mitigate the high overhead of enclave creation due to costly secure memory initialization, we propose a new type of enclave, called *shadow enclave*, to support fork-style fast enclave creation.

We have implemented a prototype of our design based on PENGLAI [24], an open-source RISC-V enclave system using a secure monitor to manage all the enclaves. We extend the secure monitor to support scalable secure memory protection with two hardware primitives, as well as fast enclave creation. The evaluation results show that PENGLAI can host 1,000s of concurrent enclave instances and support secure memory up to 512GB. PENGLAI incurs negligible overhead for CPU-intensive benchmarks (e.g., RV8 and Coremarks), and incurs 5% overhead for memory-intensive benchmarks (e.g., Redis). For startup latency, PENGLAI leverages the shadow enclave to boost enclave creation by three orders of magnitude (with 16MB enclave memory). We also evaluate PENGLAI with real-world scalable applications. The results show that PENGLAI can significantly reduce the execution time of MapReduce (3.6x speedup with shadow fork) and achieve near-native performance for a serverless application. We have implemented all the architectural features of PENGLAI on RISC-V platform, including an FPGA board, QEMU and the Gem5 simulator, and implemented the software monitor with 6,399 LoCs, including enclave/hardware management and encryption libraries. The hardware costs are also minor, i.e., 0.81% (without MMT or memory encryption) in LUT and 0.73% in FF on a Xilinx VC707 FPGA board.

PENGLAI is open-sourced at https://github.com/Penglai-Enclave.

2 Motivation

This section analyzes the scalability and security of prior enclave systems through several metrics, as shown in Table 1.

2.1 State-of-the-art Enclaves

Intel SGX [57, 77] can protect both confidentiality and integrity of enclave memory, but it has a restriction on secure memory size, i.e., 128/256MB. Also, the secure memory must reside in a contiguous region (PRM) reserved by the CPU in advance. Recently, Intel has released a scalable version of SGX [22], which extends the secure memory size to TB level but weakens the memory integrity protection, and still requires a static reserved secure memory region. AMD SEV [12, 27, 61] protects virtual machines (VMs) without memory size restriction. However, the number of secure VMs is restricted to 16 (509 in EPYC Generation 2 [14]). Intel TDX [16] is also designed to isolate secure VMs from other software, including the hypervisor. However, TDX only provides basic integrity protection and cannot defend against hardware-based memory replay attacks. The number of secure VMs is limited by hardware to 64 private keys in MKTME (Multi-Key Total Memory Encryption). ARM TrustZone-based enclaves, e.g., Komodo [52] and Sanctuary [35], have no restriction on enclave number or memory size. However, the secure memory can only reside in a few fixed memory regions and has no encryption or integrity guarantees.

Keystone [68] implements enclave memory isolation by leveraging the PMP (Physical Memory Protection) mechanism of RISC-V [100], which includes a set of paired registers to indicate physical memory regions as well as their access permissions. Thus, the number of memory regions in Keystone is restricted by the number of PMP registers (up to 16). In order to defend against physical attacks, Keystone leverages on-chip computing, which is costly due to the restricted on-chip RAM [68]. CURE [21] adopts enclave ID-based access control for customizable enclaves. It utilizes a hardware arbiter to record contiguous physical memory regions of enclaves, which can only support 13 enclaves. Similarly, the enclave number of Sanctum [45] is also restricted by the number of isolated DRAM regions. TIMBER-V [102] extends the RISC-V ISA to run unlimited number of enclaves, but it incurs non-trivial overhead (25.2% on average) and does not consider memory integrity protection.

2.2 State-of-the-art Fall Short

Fine-grained memory isolation. Prior art [27,91,92] achieves fine-grained and flexible memory isolation by introducing additional metadata like bitmap [27,92] or tags [102] to identify whether a page belongs to an enclave and check each memory access. However, due to the capacity restriction of in-SoC RAM, most of the metadata has to be stored in the main memory, which requires an extra memory load when metadata is out of SoC and incurs a high performance penalty in TIMBER-V [102].

Large-scale memory integrity protection. Several schemes [38, 39, 85, 91, 92] have been proposed to provide integrity protection for more memory. For example, VAULT [92] extends SGX and optimizes the integrity tree node structure to increase the node's fan-out, which can protect larger memory region given the same tree depth. However, these schemes need to take up extra memory space even when no enclaves are running (e.g., 14.1% in VAULT without MAC optimization), and the size of protected memory (e.g., 64GB in VAULT) is still insufficient for cloud applications.

Boosting startup latency. Some researchers add a new software layer to manage enclaves, like the secure OS in ARM TrustZone [26] and the libOS in Occlum [89], which can

Systems		Scalability Metrics				Security Metrics			
Name	Arch	Enclave number	Mem size	Fast startup	Mem granu.	No PT channel	No Cache channel	Mem enc.	Mem inte.
SGX [57, 77]	Intel	Unrestricted	128/256MB	✗	Region	✗	✗	✓	✓
Scalable SGX [22]	Intel	Unrestricted	All	✗	Region	✗	✗	✓	✗
TDX [16]	Intel	64	All	✗	Page	✓	✗	✓	Partial
SEV [12, 61]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
SEV-ES [61]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
SEV-SNP [27]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
Trustzone [26]	ARM	Unrestricted	All	✓	Region	✓	✗	✗	✗
Komodo [52]	ARM	Unrestricted	All	✓	Region	✓	✗	✗	✗
Sanctuary [35]	ARM	Unrestricted	All	✗	Region	✓	✓	✗	✗
Sanctum [45]	RISC-V	DRAM regions	All	✗	Region	✓	✓	✗	✗
TIMBER-V [102]	RISC-V	Unrestricted	All	✗	Page	✓	✗	✗	✗
Keystone [13, 68]	RISC-V	PMPs	All	✗	Region	✓	✓	On-chip	On-chip
CURE [21]	RISC-V	13	All	✗	Region	✓	✓	✗	✗
PENGLAI	**RISC-V**	**Unrestricted**	**All**	✓	**Page**	✓	✓	✓	✓

Table 1: A comparison on enclave systems. *Mem granu.* means the granularity of secure and non-secure memory. Region means secure memory can only reside in a few contiguous memory regions. *Mem enc.* means memory encryption. *Mem inte.* means memory integrity protection. *Unrestricted* means the number of enclaves is unrestricted, but when secure memory is exhausted, the performance will decline. *Partial* means the physical memory replay attack is out of scope. *16/509* means EPYC Generation 2 (Rome) processors [14] can support 509 keys for SEV VM. PENGLAI is the only system that can achieve both high-security and scalability.

create a new enclave with less overhead. However, these systems do not consider the process of attestation during enclave creation. Clemmys [93] leverages the dynamic memory management of SGX2 as well as batching for EPC augmentation to improve the startup latency of enclaves. However, creating an enclave still takes hundreds of milliseconds, and it needs to add redundant pages into EPC when creating the same enclave multiple times.

3 System Overview

We first present our design goals of memory protection.

- **G1: Scalability.** The design shall not have restrictions on (1) the number of concurrent enclave instances, and (2) the size of secure memory of enclaves. It shall also consider the characteristics of scalable applications, e.g., fine-grained memory management and auto-scaling.
- **G2: Performance.** The design shall not incur high performance overhead.
- **G3: Security.** The design shall achieve the previous two goals with security guarantees. It should consider privileged software attacks, off-chip hardware attacks (e.g., hardware-based memory replay attacks), cache-based side-channel attacks, etc.

3.1 Architecture

As shown in Figure 1, PENGLAI is a software-hardware co-design enclave system. We introduce a small software component called *secure monitor*, which runs in the most privileged mode (e.g., machine mode in RISC-V) and several new hardware extensions to provide the enclave abstractions. Each enclave runs in the user space and is isolated from an untrusted host and other enclaves.

Secure monitor. The secure monitor runs in the most privileged level and separates OS and userspace software into two worlds: one for the OS and normal applications, the other

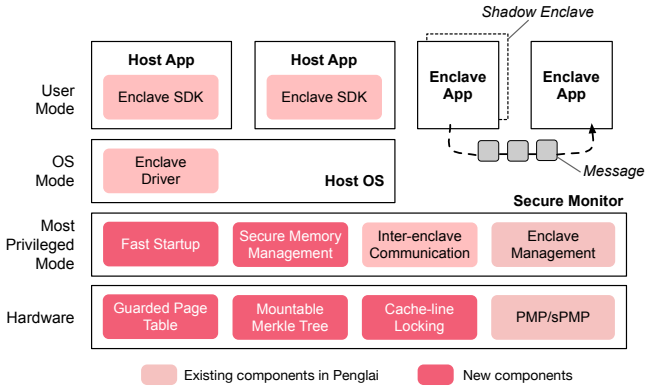


Figure 1: Overview of PENGLAI architecture. PENGLAI is composed of the software monitor, driver, SDK and hardware extensions. The red components are additional to realize the scalable memory protection.

for enclaves. The secure monitor manages all enclaves and provides APIs for users to deploy enclaves. To achieve scalability, we add two components in the secure monitor: one for fine-grained and large-scale secure memory management, and the other for enclave fast startup. Furthermore, to minimize the size of the secure monitor, we separate resource protection from management [51]: the secure monitor only configures privileged hardware resources (e.g., GPT and MMT configurations), and the managements of other hardware are done by the untrusted host OS.

During system boot, the secure monitor is loaded and verified by the boot ROM (aka. secure boot). It then takes control of the system and protects itself with hardware-supported memory isolation (e.g., RISC-V PMP). It also leverages encryption and Mountable Merkle Tree to protect itself from physical memory attacks (details in 4.2).

Hardware primitives. We propose new hardware primitives to assist the secure monitor and achieve scalable memory

protection. We briefly introduce their purposes here. Guarded Page Table (GPT) is the basis of fine-grained memory isolation (§4.1). Mountable Merkle Tree (MMT) is a new physical memory protection abstraction to achieve scalable memory integrity and encryption protection (§4.2). Cache line locking is a cache partition extension to defend against cache-based side-channel attacks (§4.4).

3.2 Threat Model

The TCB of our system only contains the CPU and the secure monitor. Other hardware (e.g., off-chip DRAM and peripherals) and software (e.g., the host OS) are untrusted and could be compromised by an attacker.

We consider three classes of attacks in our threat model:

- **Privileged software attacks:** An attacker may have full control of the untrusted OS and applications and launch adversarial enclaves.
- **Physical attacks:** An attacker may intercept and tamper with any messages between CPU and other hardware (e.g., DRAM), and issue attacks from any off-chip hardware.
- **Side-channel attacks:** An attacker may learn information by observing access patterns during enclaves' execution. Our system aims to solve the controlled channel attacks [81] and cache-based side-channel attacks [74, 82, 107, 109, 110]. Other side-channel attacks, like the ones based on TLB or speculative execution, are out of the scope. The potential defense mechanisms of these attacks are orthogonal to our design.

Our system does not consider DoS attacks performed by untrusted software or hardware.

4 Design

This section focuses on how the secure monitor and the hardware extensions achieve the design goals. We discuss other security issues in §7.

4.1 Fine-grained Flexible Memory Isolation

For fine-grained and flexible memory isolation, the secure monitor maintains an ownership bitmap to record the status of each physical page: *secure* for monitor and enclaves, *non-secure* for untrusted OS and applications, and *TreeNode* for SubTrees (details in §4.2). It achieves 4KB page granularity to isolate memory between enclaves and host. To allocate secure pages, the secure monitor needs to update the ownership of them in the bitmap. The ownership bitmap is protected by hardware (e.g., RISC-V PMP). Any memory access to the ownership bitmap issued by OS or user-level programs will trap into the monitor for a security check.

Several similar approaches [27, 91, 92, 102] also use the ownership bitmap to achieve fine-grained memory isolation. However, these approaches check the page ownership during the memory access, which needs double memory access for a single address. Prior work demonstrates that the double memory access may introduce the 25.2% average runtime overhead [102].

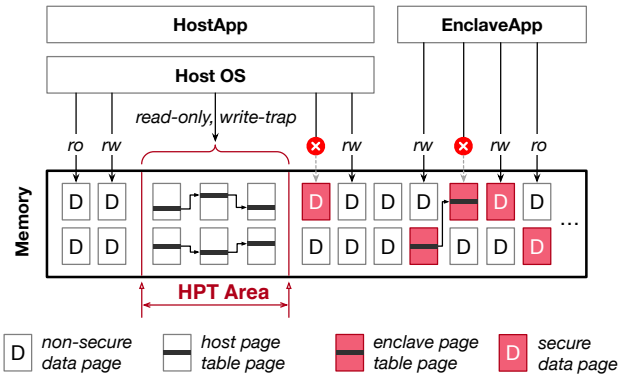


Figure 2: The design of Guarded Page Table with Host Page Table Area (HPT Area). PENGLAI maintains two kinds of page tables: the host page table is used by untrusted software, and the enclave page table is used by enclave.

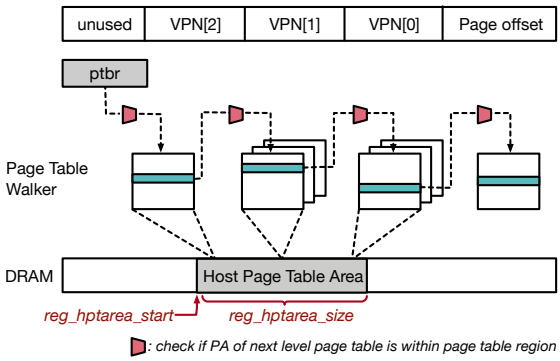


Figure 3: The MMU extension of Guarded Page Table. Extensions are marked as red. MMU will check the location of each page table. VPN means virtual page number.

We propose a new hardware primitive, Guarded Page Table, to shift the ownership checking overhead to the mapping phase, which is based on an observation that mapping operations are far less frequent than memory accesses.

Guarded Page Table. Guarded Page Table implements and optimizes ownership checking with bitmap. The design is based on our insight that if there is no page table of untrusted software containing any mapping to secure pages, then the checking during memory access can be avoided. To this end, we put all host page tables (Figure 2) in a protected memory region: Host Page Table Area (HPT Area), and trap any modification in HPT Area to ensure that *no secure page will be mapped by any page tables of untrusted software*.

HPT Area is indicated by two new registers, *reg_hptarea_start* (start physical address of HPT Area) and *reg_hptarea_size* (size of HPT Area), as shown in Figure 3. To guarantee all the host page tables are located in HPT Area, we extend page table walker (PTW). When a TLB miss occurs, the PTW will walk into each level of page table page (PT page) according to the virtual address to get a corresponding physical address. Our extension to PTW will check whether each PT page is

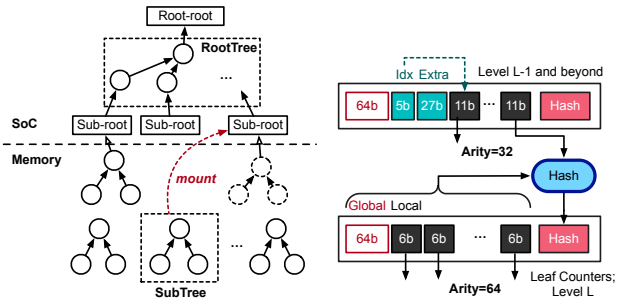
located in HPT Area, as shown in Figure 3. If the address of any PT page is out of HPT Area and the software currently running is not an enclave, the CPU will raise an exception to the monitor for further check. Furthermore, a CPU mode status register, *reg_ms*, is introduced to indicate whether the current CPU is running an enclave. Therefore, we can enforce the untrusted OS to only use pages in HPT Area as Guarded Page Table.

Protecting Host Page Table Area (HPT Area). Similar to the ownership bitmap, HPT Area is also protected by the secure monitor with hardware support. When the OS updates address mappings, the request will be redirected to the secure monitor, which ensures the new page table entry does not point to a secure page. Also, we need to prevent the OS from bypassing such checking via stale TLB entries or disabling page table. Firstly, the secure monitor will flush the corresponding TLB entries during enclave switching and page ownership changing. Directly writing TLB entries is out of scope as there is no such instruction in the prevailing ISA (e.g., X86, RISC-V). Secondly, the hardware will raise an exception if address translation is disabled while the *reg_hptarea_start* and *reg_hptarea_size* registers are non-zero (these two registers can only be modified by monitor). HPT Area is also protected from hardware attacks such as the PThammer attack [113] via hardware integrity protection (details in §4.2).

Memory isolation among enclaves. As all enclaves are running in user mode, PENGLAI utilizes an enclave page table for memory isolation among enclaves. Enclave page tables are marked as secure memory and separated from HPT Area. The secure monitor maintains all the enclave page tables, and each enclave can map its own secure memory as well as non-secure memory shared with the OS.

Huge page support. To support huge pages, we further partition the HPT Area into several sub-areas and assign different levels of page table entries to the corresponding sub-area, including one sub-area for PMDs (huge page entry) and one for PTEs. The extended PTW will check whether each level of page table entry lies in the corresponding sub-area during page table walk. In this way, the secure monitor can distinguish a huge page table entry via its address and perform different security checks.

Summary. We summarize the benefits of the ownership-based design with Guarded Page Table. First, the hardware modification is minor, and the hardware maintains no in-memory metadata like SGX EPCM. Second, it achieves fine granularity since any physical memory page can be used as secure or non-secure. The only contiguous range, Host Page Table Area region, has little impact on scalability as the page table pages are much less than data pages. Last, the design introduces no overhead of checking during memory access. The only costs come from the page table mapping operations. Compared with other page table-based isolation, e.g., shadow PT [94], the mapping overhead is minor.



(a) Forest structure. **(b) Counter layout.**

Figure 4: Mountable merkle tree. (a) The top panel is the mounted SubTree root and RootTree. The bottom panel is the hash forest constituted by multi-SubTrees. (b) The arity of SubTree is 32, 32 and 64, while 32, 32, 32 for RootTree. The "Idx" points to a local counter, which can use the "Extra" to avoid frequently global counter updating.

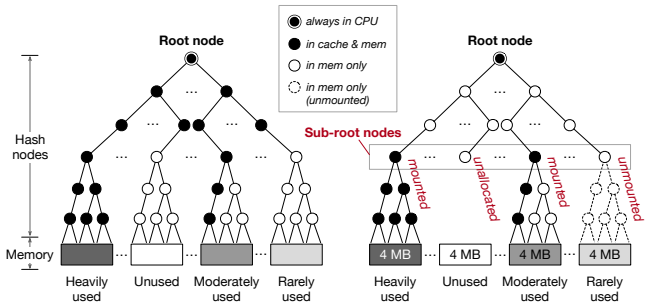
4.2 Scalable Memory Integrity Protection

We propose a new physical memory protection **abstraction**: *Mountable Merkle Tree (MMT)*. MMT promises a stable tree depth that will not increase along with total memory size and minimizes the memory space overheads by storing integrity metadata (tree nodes) on demand. Furthermore, the secure monitor can manage MMT to achieve large-scale, fine-grained memory integrity protection. In our prototype, MMT can support up to 512GB of secure memory.

Challenge. It is very challenging to achieve scalable integrity protection, as shown in Figure 5 (a). First, protecting integrity for large-scale secure memory turns to a deep integrity tree, which requires additional bandwidth to load tree nodes. Second, to boost memory integrity checking, a wise memory integrity engine may cache topmost tree nodes in the CPU cache. However, it only increases the amount of secure memory linearly. Third, the integrity engine needs to pre-allocate extra memory to store all the tree nodes, even if there is no secure memory being used. Lastly, the state-of-the-art memory protection schemes can only protect a fixed range of memory, and software cannot manage secure memory at all. These coarse-grained and fixed memory protection schemes have scalability issues which cannot be solved by just adding hardware resources (e.g., enlarging SoC storage).

MMT introduces a mountable SubTree structure for integrity tree scheme and can reduce both on-die and in-memory storage overhead. Also, MMT allows the software to take part in memory protection management and allocate secure memory with integrity protection on demand.

MMT forest organization. MMT introduces a new concept, *hash forest*, which is composed of a set of SubTrees, as shown in Figure 4 (a). The SubTree is the mountable and manageable unit in *hash forest* and can protect a physical memory region (4MB/3-level in PENGLAI) alone. To save on-die space, MMT stores all SubTrees in a specific memory region, *MMT meta-*



(a) Static Merkle Tree. (b) Mountable Merkle Tree.

Figure 5: Comparison between static Merkle tree and MMT.

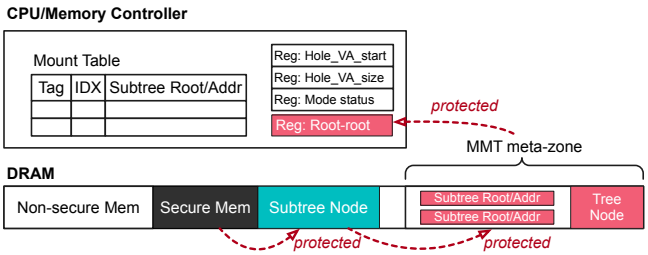


Figure 6: Hardware extension for MMT. The flow shows the integrity protection relation: secure memory is protected by SubTree nodes, while SubTree nodes are protected by MMT meta-zone.

zone. MMT utilizes *RootTree* to protect the integrity of all SubTree roots. The ancestor of RootTree (*Root-root* in the figure) is fixed in SoC to prevent attacks. The Root-root, RootTree, and SubTrees, together form *hash forest*.

SubTree Allocation. Besides normal memory allocation, the software (secure monitor) can also allocate SubTree and secure memory with two new interfaces: *ALLOC_SECURE_MEM*, *REVOKE_SECURE_MEM*. Each SubTree protects a range of secure memory. Unlike traditional memory protection schemes, the CPU can only protect a fixed range of physical memory, and the whole checking procedure is transparent for software. MMT allows the privileged software (e.g., the secure monitor) to allocate secure memory at runtime, and both secure memory and SubTree can reside in anywhere of physical memory, not a fixed range.

Mounting. Like TLB accelerating virtual address translation, MMT accelerates integrity checking by mounting SubTrees into SoC, which avoids the memory access to retrieve SubTree root. MMT extends the memory controller to support mounting operations. As shown in Figure 6, the mounted SubTree root is stored in the *Mount table*, which records counter and address. However, the storage space for *Mount table* in SoC is restricted, e.g., 32 subtrees simultaneously. When space is exhausted, MMT unmounts an inactive SubTree root out of SoC and stores it in the MMT meta-zone, which is isolated with host memory (e.g., PMP-protected). Meanwhile, MMT needs to update the value of Root-root if MMT meta-zone is changed, which ensures the integrity of all inactive SubTree roots.

Bootstrap. Figure 6 demonstrates the hardware extension and memory layout of MMT. Besides the non-secure memory, there are three protected regions, *secure memory* (enclaves and monitors), *SubTree nodes* and *MMT meta-zone*. The MMT meta-zone is the only fixed region configured by bootrom and protected by hardware (e.g., RISC-V PMP). MMT meta-zone contains the SubTree root entries (address and counter) and RootTree nodes (Root-root is reserved in SoC). It incurs minor memory costs — about 2MB, which can support up to 512GB of integrity-protected memory. The integrity protection relation of these three regions is shown in the figure.

During system boot, the CPU bootrom will configure the MMT meta-zone range in physical memory, initialize all SubTree root entries, construct RootTree nodes and allocate the first subtree to protect the memory of the secure monitor. After this, the secure monitor takes control of subsequent booting stages.

Software management. The secure monitor is the only privileged software that can manage secure memory and record memory status (secure, normal and SubTree node) in the bitmap. As the secure monitor cannot directly allocate memory, the host kernel will allocate free memory (used as secure memory and SubTree node later) and transfer it to the secure monitor. The secure monitor configures secure memory and its corresponding SubTree in this memory. After allocation, the corresponding SubTree root is filled in the *MMT meta-zone* (i.e., *ALLOC_SECURE_MEM*). The monitor also ensures that secure memory is zero-filled and SubTree nodes are in the initialized state. So the integrity check will not fail at the first access. If the monitor needs to reclaim secure memory, it can change the memory status, clear the SubTree root in the *MMT meta-zone* (i.e., *REVOKE_SECURE_MEM*), and return memory back to the host kernel.

MMT tree structure. MMT extends the counter-based integrity tree node [55, 85, 92] to *hybrid-counter*. As shown in Figure 4 (b), each tree node is composed of $global\_counter(64b)$, $extra\_idx(5b)$, $extra\_counter(27b)$, $local\_counter(32 \times 11b)$ and $hash(64b)$ (however, leaf node only contains global, local counters and hash). The hash in the tree node is calculated with all other metadata (448b) in the same node and hybrid-counter in its parent node. The *extra counter* in a hybrid-counter remains zero unless *extra_idx* refers to itself. This design is based on an observation that there is almost one active counter in a single tree node. In our implementation, both SubTree and RootTree are 3-level, and each RootTree leaf node can contain 4 SubTree roots.

Integrity checking. For each secure memory access, MMT will first check whether the corresponding SubTree root is in SoC. If it is, MMT checks the integrity with the SubTree. Otherwise, MMT uses *mounting* mechanism to mount the target SubTree into SoC. As for the integrity checking procedure, the MMT engine will compare the hash stored in the tree

node with the hash it computes level by level, until the hash is stored in SoC. A write request will increase hybrid-counter in the tree node, while a read request will not. If a local counter is exhausted or *extra_idx* is changed, MMT will re-hash the relevant tree nodes.

Integrity enabling. PENGLAI will always enable integrity protection when executing in the monitor. Nevertheless, as an enclave can access both secure and non-secure memory, we cannot just impose integrity protection on all its pages.

PENGLAI introduces a pair of *hole registers* (*reg_hole_va_start* and *reg_hole_va_size*) to configure the integrity checking. The effects are shown on the right. Hardware shall perform integrity checking when *reg_ms* is *ENCL_MODE* and disable it when *reg_ms* is *NON_ENCL_MODE*. Hole registers indicate a dedicated memory region (hole region) in the virtual address that handles exceptional cases in the memory protection strategy, e.g., integrity checking is enabled for the memory located in the hole region when *reg_ms* is *NON_ENCL_MODE*, and disable when *reg_ms* is *ENCL_MODE*. The hole region can be used for sharing untrusted memory between enclaves and OS. Besides, it cannot be used by host and enclave for other purposes. To avoid disabling the integrity protection by the malicious kernel, PENGLAI only permits the secure monitor to configure the hole registers when switching between enclave and host.

	MS=ENCL	MS=NON_ENCL
Hole mem.	Disable	Enable
Non-Hole mem.	Enable	Disable

Secure memory granularity. As integrity enabling is controlled by the combination of CPU mode (*reg_ms*) and virtual memory address (*hole register*), the SubTree is not the granularity of secure memory. A SubTree can contain both secure memory and non-secure memory, and only secure memory needs integrity and encryption protection. When considering Guarded Page Table for memory isolation, the combined granularity of secure memory is still 4KB size.

Security analysis. As shown in Figure 6, the integrity of secure memory is guaranteed by SubTree root. Each SubTree root has a backup in the MMT meta-zone, which is protected by Root-root. As Root-root resides in SoC, physical attackers cannot compromise the integrity check. As for software attackers, only the secure monitor can configure secure memory, and any other privileged software cannot tamper memory status and disable integrity protection.

Summary. We summarize the benefits of MMT against prior art, as shown in Figure 5. (1) Save both on-die and in-memory storage. Prior art reserves intermediate tree nodes for all secure memory, and the topmost level tree nodes may be stored in SoC to boost the integrity check [15]. It can only protect a small region of contiguous memory due to the high storage overhead in memory and SoC. However, MMT merely reserves the hot set of SubTree roots and Root-root in the SoC. What's more, SubTree nodes can be lazily allocated with corresponding secure memory (zero memory overhead if there is no secure memory). (2) Improve flexibility in secure memory management. With the help of allocating and mounting operations for SubTrees, MMT can support fine-grained secure memory. The software can manage secure memory, dynamically change memory status, and allocate secure memory on-demand. (3) Boost the integrity check. MMT can provide a fast path (a mounted SubTree) to boost the integrity check with fixed tree depth and save memory bandwidth.

4.3 Secure Memory initialization with Shadow Fork

Auto-scaling and fast startup are key features for cloud computing but still missing in the enclave due to high-cost enclave memory initialization. Prior systems need to create a new enclave instance from scratch even with the same codebase, which consumes more memory with redundant content and incurs high startup latency. PENGLAI follows the idea of recently proposed init-less startup [50] that leverages *fork* to skip the initialization costs, but faces two challenges: 1) memory sharing is not secure in enclave systems, and 2) attestation costs still remain even with fork. PENGLAI proposes *Shadow Fork* as well as *Shadow Enclave* to overcome them both.

Fork with the shadow enclave. Shadow Fork is based on a special kind of enclave (not runnable), *shadow enclave*, which is a clean template used to boost startup by forking a new instance. Shadow enclave is the only entity that can be forked and only contains code and data segments. During fork, PENGLAI monitor will share the read-execute code and read-only segments, copy other writable parts, initialize the stack of a new instance based on *Shadow Enclave*. As the major costs of startup come from enclave memory initialization (hash measurement), memory copying on writable data is acceptable. After fork, the created enclave can dynamically allocate memory from untrusted OS as heap or mmap region.

Lightweight attestation. Mitigating the costs of attestation during startup is based on an observation: calculating the measurement of memory takes up the majority of time in attestation (e.g., >90%), as shown in Figure 11 (b). To mitigate this overhead, the monitor will calculate the measurement for a *shadow enclave* in advance (creation phase). A user can leverage *enclave_fork* with a manifest containing the sealed enclave measurement and the user's public key (similar to SGX [23]). Later, the monitor will unseal the enclave measurement (using the user's public key) and check it with the *shadow enclave*'s measurement. If the measurement is matched, the monitor will fork a new instance based on the *shadow enclave*. Otherwise, the monitor will deny the request. Therefore, we can mitigate the attestation costs during the boot critical path.

4.4 On-demand Cache Line Locking

PENGLAI proposes an on-demand cache line locking mechanism to defend against cache-based side-channel attacks by

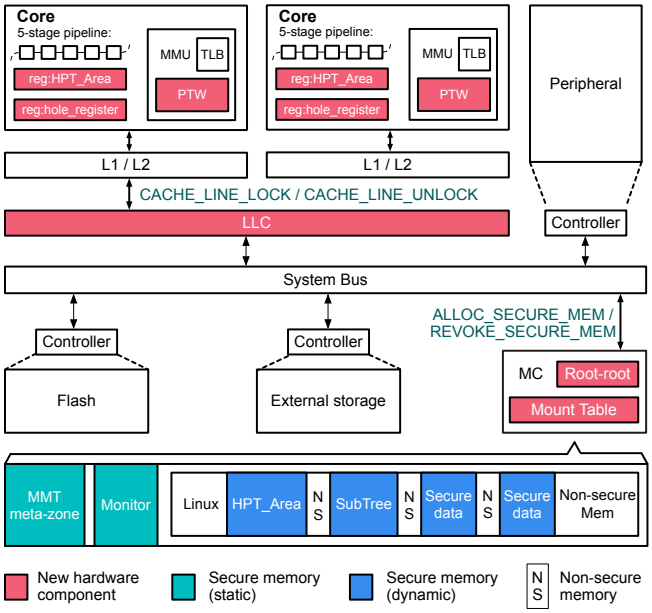


Figure 7: The hardware modification and physical memory layout of PENGLAI.

cache partition. Although cache partition designs are well explored [9, 73, 99, 112], they usually incur non-trivial performance costs [66, 83, 106], and cache partitions also restrict the number of the enclave [45]. PENGLAI optimizes the performance with a new abstraction, *section-based protection*, that can enable on-demand side-channel protection when an enclave enters a security-sensitive section (e.g., encryption) and disable the protection when it leaves. As the sections are short in a program [105, 108], an enclave can run with the best performance (no cache locking) most time.

Security-sensitive sections are decided by enclaves. When an enclave needs cache isolation protection, it issues requests to monitor. Two privileged instructions: *CACHE_LINE_LOCK* and *CACHE_LINE_UNLOCK*, assist monitor in managing cache line locking status. Specifically, the cache line locking mechanism specifies the cache lines to each CPU core. The CPU core can only evict its cache lines during cache miss, and other cores cannot evict these cache lines anymore.

Scalability discussion. Compared with the Intel's CAT [9, 73], PLCache [99] or cache coloring [112], on-demand cache line locking is more scalable. The cache partitions limit the number of protected enclaves in the prior art. However, as for on-demand cache line locking, the cache line is not assigned to an enclave, but a CPU core. In other words, if there are more cache ways than CPU cores, the cache line locking mechanism can support unlimited enclaves. Monitor holds the cache locking status in each enclave's context. If an enclave is scheduled out, the monitor will release the cache lines locked by the current core.

5 Implementation

Figure 7 shows the overall architecture of the PENGLAI. As for hardware extensions, we modify both in-core (Rocket Core) and off-core (Memory Controller) hardware resources to support Guarded Page Table and Mountable Merkle Tree. As for software extensions, we implement a tiny and secure monitor, an extended Linux kernel supporting the HPT Area allocator, an enclave driver and some requisite libraries for running enclaves.

5.1 Hardware Implementation

In-core extensions. We implement the Guarded Page Table extension in FPGA, based on the open-sourced RocketChip RISC-V core [10]. Overall, we add several new registers (e.g., *reg_ms*, *reg_hptarea_start[0..3]*, *reg_hptarea_size*, *reg_hole_va_start* and *reg_hole_va_size*, etc.), which are implemented as CSRs (Control and Status Registers) and can only be accessed by monitor via *csrr* (CSR read) and *csrw* (CSR write) instructions. *Reg_hptarea_start[0...3]* and *reg_hptarea_size* registers partition the physical memory range of HPT Area into several sub-areas, and guarantee that any write access cannot directly modify the content in this area, unless issued by M mode routines. *Reg_ms*, *reg_hole_va_start* and *reg_hole_va_size* registers act as the control switch for memory integrity and encryption checks. Besides these new CSRs, we also extend the MMU module to check the validity of memory access during page table walking. A modified page table walker can guarantee that any PTE entry must be located in the HPT Area (precisely, corresponding sub-area), but the target physical address **will not** reside in the HPT Area.

We only simulate the cache line locking mechanism on the L1 cache, as there is no LLC in our FPGA board.

Off-core extension. We extend the memory controller (MC) to integrate the MMT engine. MMT engine supports three new commands: *ALLOC_SECURE_MEM*, *REVOKE_SECURE_MEM* and *INIT_MMT_METAZONE*; two extended components: Mount Table, Root-root, and the logic of memory encryption and integrity checks. We also extend memory access with the secure/non-secure flag. If it is the secure memory access, the MMT engine will perform integrity and encryption checks. Otherwise, it accesses physical memory and reads/writes the data directly.

The secure monitor allocates/reclaims secure or SubTree node memory with new commands. When the MMT engine receives an *ALLOC_SECURE_MEM* command, it will parse the secure memory address and initialize the SubTree root in the MMT meta-zone. The software must ensure that the memory must be zero-filled before it changes into the secure or SubTree node memory. The *INIT_MMT_METAZONE* command initializes the MMT meta-zone in the booting phase.

Mount Table and Root-root reside in SoC. The fabric of Mount Table is similar to cache — several sets with n-way Mount Table entries. Each Mount Table entry consists of a

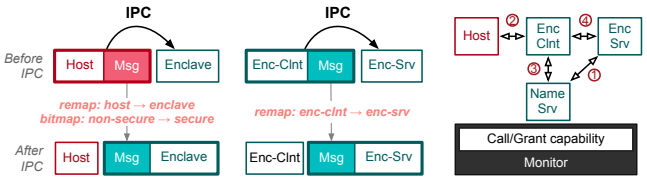
tag, index, SubTree root and LRU bit. Mount Table also uses the LRU strategy (clock-like algorithm) to mount/unmount the subtree roots. If SubTree root does not exist in the Mount Table, the MMT engine will choose an inactive SubTree and unmount it into the MMT meta-zone. Before mounting the requested SubTree, the MMT engine will first check the integrity of the SubTree root in the MMT meta-zone, and update the Root-root if necessary. Root-root cannot be evicted out of SoC, as it is the root trust for integrity protection.

5.2 Software Implementation

Monitor. We implement the secure monitor on both OpenSBI [18] and Berkeley Boot Loader (BBL) [17] in the machine mode in RISC-V. The secure monitor includes enclave management, hardware extensions management, memory checking as well as encryption library, which adds 6,399 LoC. We follow Sanctum [45] to implement the secure boot using the tamper-proof software approach (bootloader as the root of trust). As Figure 7 shows, just after the machine is turning on, the bootloader will first derive the attestation key and initialize the MMT engine. The MMT meta-zone will also be initialized in this phase. After all these early configurations, the bootloader will load and calculate the measurement of the secure monitor. The MMT engine performs integrity and encryption protection for monitor's memory as well. After this, the secure monitor takes control and loads the Linux kernel as its payload.

The monitor provides both host-side and enclave-side interfaces for enclave management and runtime supporting. As for the host-side interfaces, they consist of *create_enclave*, *run_enclave*, *attest_enclave*, etc. As for enclave-side interfaces, they are mainly used as ocall functions (e.g., *enclave_mmap*, *enclave_sys_write*) and inter-enclave calls (e.g., *enclave_call*, *asyn_enclave_call*). Besides the enclave management, the monitor also takes control of configuring Guarded Page Table and Mountable Merkle Tree (e.g., *reg_hptarea_start*, *reg_hptarea_size*), setting page status bitmap and allocating secure memory. Monitor guarantees that all these secure-sensitive configurations are correct and will not be compromised by an attacker.

Linux kernel. We extend the Linux kernel (version: 4.4.0/5.10.2) to support PENGLAI. There are two major modifications: (1) HPT Area allocator, (2) hijacking each PTE settings. Firstly, after memory management is initialized, the kernel will allocate a contiguous physical memory as HPT Area and copy *init_pt* into it. A dedicated allocator will manage all pages in the HPT Area, and is responsible for each page table allocation. To distinguish huge page entries and 4KB page entries, the HPT Area is divided into three sub-areas: PGD, PMD and PTE sub-areas (MMU checks each page table entry's location according to these sub-areas). HPT Area allocator must assign the entries of page tables in the corresponding sub-areas as well. Currently, we have reserved enough memory as HPT Area, which can map all pages at



(a) Host-Enclave IPC (b) Enclave-Enclave IPC (c) Communication Link

Figure 8: Ownership transfer-based communication. Figure (a) and (b) show the message passing in PENGLAI for both host-enclave communication and enclave-enclave communication. Figure (c) presents the communication link established in PENGLAI. The number represents the order of the links.

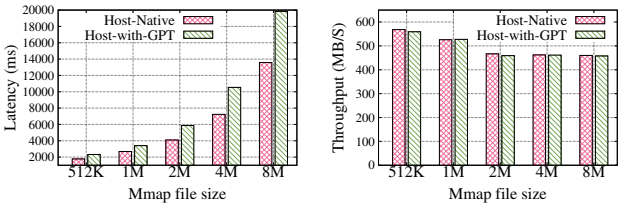
once. Normally, the reserved memory for HPT Area will not affect memory utilization, as page table pages are increased along with used pages. Dynamically adjusting the size of the HPT Area is in our future work.

Secondly, the kernel will redirect the setting operations of each page table entry to the secure monitor. Secure monitor can distinguish 2MB/4KB page entries (according to the sub-areas) and perform the security check of the target address (host cannot map any secure memory).

Server enclave. Despite the enclave and shadow enclave, PENGLAI also implements another type of enclave called *server enclave* to achieve enclave chain, which is common in serverless scenarios. A server enclave does not have running context (e.g., time slice, ocall handler) but inherits it from other enclaves. Hence, the server enclave cannot run alone. When creating a server enclave, it needs to be assigned a unique name as its identification. Other enclaves can acquire the handle of this server enclave with its unique server name. Besides, the server enclave can also perform partial functionalities of OS. For example, we can run a file system server in a server enclave to handle all FS-related requests. Separating the OS functionalities from the untrusted OS to the trusted enclave server can mitigate the risk of Iago attack [41] issued by untrusted privilege.

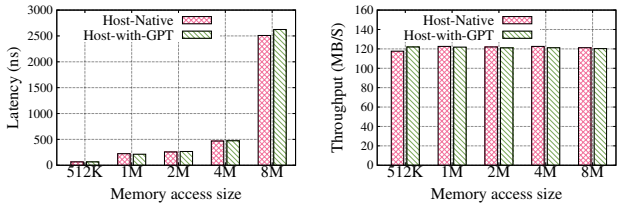
IPC. PENGLAI supports IPC between the enclave and server enclave (E-E), host and enclave (H-E), which is based on two mechanisms: shared memory and relay page [49]. Shared memory is the basic communication method. The secure monitor can map shared memory to both host and enclave, or enclave and server enclave. Relay page is a novel communication mechanism, as shown in Figure 8, and the secure monitor ensures that a page can be mapped for only one owner simultaneously. This mechanism can reduce security issues like TOCTTOU (Time-of-check-to-time-of-use) between E-E and H-E, and achieve zero-copy communication. PENGLAI can also support both synchronous and asynchronous IPC between enclaves. As for synchronous IPC, the caller enclave will wait for the callee enclave to return. As for asynchronous IPC, the caller enclave will return immediately, and arguments will be passed to the callee enclave when it starts to run.

SDK. PENGLAI provides an SDK (i.e., kernel driver, host-



(a) Mmap latency. **(b) Mmap bandwidth.**

Figure 9: GPT performance on mmap. Test the latency and bandwidth of mmap operations, *Host-Native* represents the native linux kernel without GPT extension, *Host-with-GPT* represents the modified linux kernel with GPT extension.



(a) Memory latency. **(b) Memory bandwidth.**

Figure 10: GPT performance on memory access. Test the latency and bandwidth of memory access operations, *Host-Native* represents the native linux kernel without GPT extension, *Host-with-GPT* represents the modified linux kernel with GPT extension.

side library and enclave-side library) to help users manage and develop enclave applications. The driver enables the untrusted host kernel (Linux) to interact with the secure monitor, and manage enclaves via interfaces provided by the monitor. The host side library abstracts ioctl interfaces from the driver and provides APIs to manage the enclave (e.g., create, run and attest enclave). The enclave side library is combined with modified Musl LibC (turns system calls to ocall or redirects to server enclaves) and Eapp library (e.g., IPC interfaces). Hence, PENGLAI can support unmodified POSIX applications. Besides, PENGLAI also integrates some useful libraries into enclave SDK, such as wolfssl [20] and PSA storage API, which are frequently used.

Formal verification. Currently, we are working on formal verification of PENGLAI. The approach is based on symbolic execution and bounded model checking via the state-of-the-art framework, Serval [79]. We have verified the code of the communication module. Verification on others is in progress.

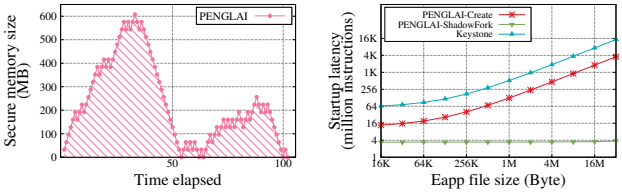
6 Evaluation

6.1 Methodology

We implement PENGLAI based on the open-sourced RISC-V [100] implementation: SiFive Freedom U500 [10] on the Xilinx VC707 FPGA board. We present several microbenchmarks that evaluate the scalability metrics (i.e., startup latency, GPT overhead and the number of concurrent enclaves). We select four benchmarks, SPECCPU, Redis, RV8 and Coremark, to evaluate the overall performance of PENGLAI. We also compare PENGLAI with Keystone (as state of the art) and Linux (as the ideal performance). To evaluate the performance of MMT, we implement the MMT on the GEM5 [32] (RISC-V), and port the state-of-the-art integrity protection schemes: SGX integrity approach (SIT) and VAULT [92] on the GEM5. With the same emulation environment, we can make a fair comparison of these different approaches.

6.2 Microbenchmarks

Guarded Page Table performance. We first evaluate Guarded Page Table on LMbench [7] to gain the overhead for memory-related operations. LMbench can calculate the latency and bandwidth of memory mapping and memory access



(a) Scalability of secure memory. **(b) Startup latency.**

Figure 11: Scalability and startup. Figure (a) shows the variation of secure memory size when creating 100 enclaves concurrently. Figure (b) compares the startup latency with different enclave file sizes.

in the UNIX system. As shown in Figure 9 and Figure 10, Guarded Page Table will introduce 26%∼46% overhead for mapping latency, as each page mapping operation needs to be checked by the secure monitor. However, Guarded Page Table will not sacrifice the bandwidth of the mapping operations. As for memory access, Guarded Page Table will not affect memory access performance (both latency and throughput). The experimental results demonstrate that Guarded Page Table only incurs extra overhead in page mapping (e.g., mmap, munmap, mprotect), and will not impact the performance of memory access. As page mapping operations are infrequent, Guarded Page Table only introduces minor overhead for the whole system in most cases. Indeed, the monitor only inspects the page status with its bitmap during page mapping, so the extra overhead is minor compared with other page table-based isolation techniques [48, 54] (see §6.3).

Scalability of secure memory. To evaluate the scalability and flexibility of the PENGLAI enclave, we construct a test case to randomly create and run enclaves on the FPGA board, and calculate the total secure memory size in the monitor. As shown in Figure 11 (a), secure memory size could be very small when the system does not run any enclaves (2MB and less) and can scale to 600MB (1GB total memory in FPGA) when there are lots of concurrently running enclaves. This is a significant breakthrough over traditional fixed partitioned secure memory design, which usually needs to make a trade-off between host memory size and enclave memory size.

Furthermore, with the same configuration, PENGLAI can achieve up to 1,000 concurrently running enclaves on the

FPGA board (1GB memory). It is possible to boot more enclave instances when the device has more resources. Hence, Guarded Page Table-based secure memory management can achieve scalability and flexibility.

Startup latency. We evaluate the startup latency of enclaves in PENGLAI and Keystone using the different sizes of enclaves. We compare two approaches of PENGLAI: normal startup (enclave create) and fast startup (shadow fork). The baseline is a traditional booting solution that loads an eapp file into the memory, prepares the resources, and verifies the measurement. PENGLAI shadow fork leverages shadow enclave to spawn a new instance, which can boost the procedure. To further compare with Keystone, we configure Keystone to use the minimum size of their eyrie runtime. We run both systems on Qemu and use the number of executed instructions (icount enabled) to represent the performance. The result is shown in Figure 11 (b). Compared with the baseline, shadow fork can achieve 4x–989x speedup (from 16KB to 32MB size). Keystone is orders of magnitude slower than PENGLAI with shadow fork, as it needs to calculate enclave measurement and prepare a new PMP region as well as load runtime in the supervisor mode.

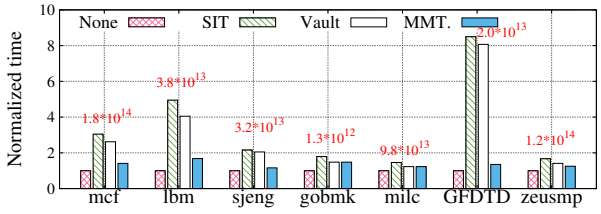
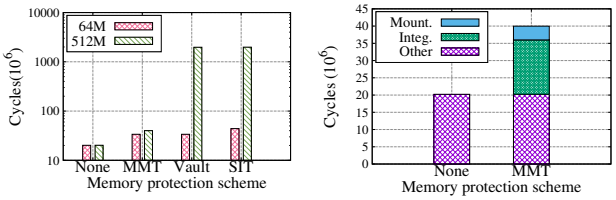


Figure 12: SPECCPU with different integrity protection schemes. *None* represents the ideal performance (w/o integrity and encryption protection). The memory costs of gobmk and milc are less than 128MB, so it will not trigger the swap mechanism in SGX and VAULT. The red numbers are the concrete execution time (cycles) of the "None".

Memory integrity and encryption. We evaluate the performance of memory integrity and encryption using SPECCPU benchmark and compare PENGLAI with VAULT and SGX. We retain the same SoC resource for all implementations (SoC storage in each integrity scheme can only protect 128MB memory). We want to demonstrate that with the same hardware resource, MMT can achieve the best performance as well as minimum memory overhead. The major reason to choose SPECCPU benchmark is that most of the related work like BMT [85], SIT [55] and VAULT [92] also use this benchmark to measure runtime overhead.

We implement all of the four systems on GEM5. As shown in Figure 12, the performance of PENGLAI is much better than the two baselines. Take milc as an example. The runtime overhead reduces from 2.05x (in SGX) and 1.60x (in VAULT) to 0.40x (PENGLAI's MMT). This is because 128MB memory is insufficient in this case. Thus, both SGX and VAULT need to swap and encrypt pages from secure memory to non-secure



(a) Random memory access. **(b) MMT performance breakdown.**

Figure 13: Worst case performance for Mountable Merkle Tree. 64MB and 512MB in figure(a) mean the memory size for random access. *Mount.* represents the cost of the mount/unmount operations, *Integ.* represents the cost of the memory integrity check, *Other* represents original runtime cost.

memory, which can bring 10x overhead in some memory-intensive cases [92]. Instead, MMT can mount corresponding SubTrees to protect more than 128MB secure memory with less than 1% extra overhead (mounting only costs about 300 cycles). We also test the performance of MMT when the memory used is less than 128MB (gobmk, milc). The result shows that MMT will not bring extra overhead when all SubTrees can be mounted in SoC.

To further evaluate the worst-case performance of the MMT, we test the MMT and other memory protection schemes on a random memory access benchmark, as shown in Figure 13(a). We choose two different memory access ranges: 64MB and 512MB. The first one is smaller than the capacity of SoC-protected memory (128MB), while the second is larger. As for 64MB memory size, MMT incurs 67% overhead and SIT incurs 118% overhead. The performance improvement mainly comes from the optimized tree structure compared with MMT and SIT. As for 512MB memory size, MMT incurs 0.97x overhead and SIT incurs 98x overhead. Copying and encrypting the pages inside SoC-protected memory into the normal memory causes an enormous overhead in SIT (takes up 97% of total runtime cycles). This result is also validated by SCONE [28] — *"random memory access beyond available EPC may cause an overhead of three orders of magnitude"* and the real SGX-enable machine (Intel Core i7-7567U @ 3.5GHz, 300x overhead for random access of 512MB memory). MMT significantly reduces the overhead of page copying and encryption by the SubTree mounting mechanism. To further calculate the overhead of mount/unmount operations, we inventory the performance breakdown of the MMT. For random memory access, mount/unmount operations only take up ten percent of the total execution time and twenty percent of the integrity protection cost. Meanwhile, each mounting operation costs about 300 cycles on average. Hence, the mount/unmount overhead is much less than the prior art, even in the worst-case situation.

Costs of cache line locking. We perform a microbenchmark to evaluate the performance costs of cache line locking. The cache configuration of our FPGA implementation is 64 cache

Table 2: Costs of cache line locking. PENGLAI runs on FPGA.

	Latency (Kcycles)
Cache line locking	66.976
Normal	56.888

sets, 4 ways and 64-Byte cache line. The test case will sequentially read and write 16KB contiguous memory. We compare the end-to-end latency of two systems, PENGLAI enabling locking (using a single way) and PENGLAI disabling locking. As shown in Table 2, the locking can cause 17.73% overhead in the case. Although the costs are non-trivial, PENGLAI will only enable the locking for a critical section, which can significantly mitigate the overheads.

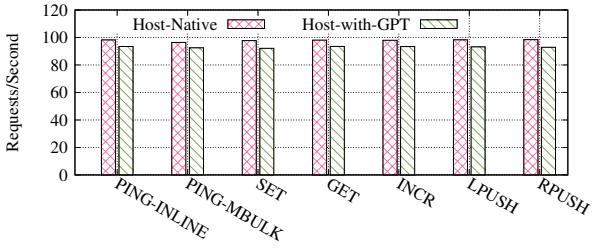


Figure 14: Redis benchmark suite. Evaluate the performance of GPT on the Redis benchmark. *Host-Native* means the native linux kernel. *Host-with-GPT* means the modified linux kernel with GPT extension.

6.3 Benchmark Suites

Redis benchmark suite. We use Redis benchmark suite to evaluate the worst-case performance of Guarded Page Table. As Redis performs as an in-memory database, it needs to call map/unmap operations frequently. Prior art also uses Redis to evaluate the performance degradation due to the mapping overhead (Shadow Page Table may incur 80% overhead on Redis [48]). As shown in Figure 14, Guarded Page Table only introduces 5% overhead in SET requests and 6% overhead in GET requests, which significantly mitigates overhead compared with Shadow Page Table. The main optimization comes from the fact that the monitor only checks page status in its bitmap. On the contrary, Shadow Page Table needs to re-construct the combined page table (traverses the host page table and extended page table), which is much more complicated and time-consuming.

RV8 benchmark suite. We use RV8 benchmark suite to answer two questions: 1) whether the isolation will incur performance overheads to enclaves, and 2) whether the Guarded Page Table hardware extensions will cause performance degradation on CPU-intensive applications. We port the benchmarks to PENGLAI, with 85LoC modifications to use PENGLAI API. In addition, we run the benchmark suite in the host kernel with two settings. One is that the host OS running in unmodified hardware without HPT Area extension, and the other is that the OS running in PENGLAI hardware with Guarded Page Table extension.

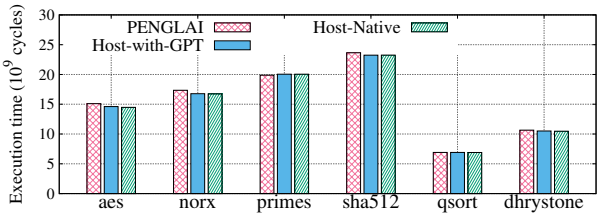
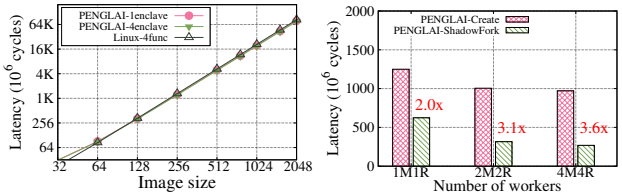


Figure 15: RV8 benchmark suite. PENGLAI can achieve almost the same performance as native Linux. Guarded Page Table only incurs minor overhead for host applications.



(a) Image processing. **(b) MapReduce wordcount.**

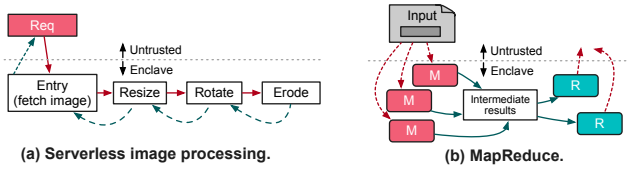
Figure 16: Evaluation of case studies. Figure (a) evaluates image processing on PENGLAI and linux. Figure (b) evaluates MapReduce performance with multiple workers.

As shown in Figure 15, the performance overheads in PENGLAI are <4.3% in all the cases and 1.7% on average, and Guarded Page Table will not affect the performance of CPU-intensive applications (the overhead is negligible compared with the GPT and Native). The overhead in PENGLAI is mainly caused by memory allocation in enclaves, which is slower than the host, as the monitor will dynamically allocate secure memory and perform the security check.

Coremark. We port Coremark with 43LoC modifications to meet PENGLAI API, and take the native Linux as our baseline and run Coremark in two systems on our FPGA board. The result shows that the score for native Linux is 2,018 and the score for PENGLAI is 2,049, which proves the strong isolation provided by PENGLAI will not hurt the performance of CPU-intensive applications.

6.4 Case Study: Serverless Computing

Existing serverless platforms [1, 3–5] use processes, containers, or VMs to isolate each function. In the case study, we try to illustrate the possibility of isolating serverless functions using enclaves (with higher security assurance) and analyze the performance impacts. We choose a representative serverless application, image processing [6, 25].



(a) Serverless image processing. **(b) MapReduce.**

Figure 17: Case studies using PENGLAI.

As shown in Figure 17 (a), the image processing application is composed of four functions, and each function is running inside a different enclave. The application is triggered when an untrusted user issues a request (with an image file). The function "Entry" is fired to get the command and fetch the host's image file through untrusted shared memory. Then, the image file will be passed into the subsequent handling functions, i.e., "Resize", "Rotate" and "Erode". Finally, the resulting image is returned to the untrusted host. Since the images may contain sensitive information, it is necessary to protect them in enclaves. The workload is ported from AWS-examples repository [6] to a C version using Sod library [19].

We evaluate the performance of image processing in PENGLAI with two settings, "1enclave" (four functions in one enclave) and the default "4enclave" (four enclaves). We implement a baseline in native Linux, using four functions running in four processes. We carefully tune the Linux performance to be state-of-the-art, i.e., using a mixed approach of the socket (for timely notification) and shared memory (for zero-copy data transfer). The input image size for the evaluation is from 32x32 to 2048x2048. The result is shown in Figure 16 (a). Compared with Linux, PENGLAI can achieve even better performance (7%–9% improvement when image size is larger than 128x128), as PENGLAI's communication does not include kernel scheduling costs. Moreover, benefiting from efficient communication, using four functions in PENGLAI incurs minor costs. This can motivate more modular applications to be deployed in enclaves.

6.5 Case Study: Secure MapReduce

MapReduce [47] is a popular programming model in the cloud for data processing, and the recent study [67] shows that even a single machine can be competent with large-scale data processing. Since the processed data may contain sensitive information, VC3 [87] and Civet [96] are proposed to enhance the security with SGX enclaves. However, using SGX to run MapReduce workloads has some limitations: first, it needs a long latency to boot a worker enclave (i.e., mapper or reducer); second, it needs to load redundant enclave code into secure memory.

PENGLAI overcomes the issues with shadow fork, as shown in Figure 17 (b). Each mapper or reducer can instantiate itself from a pre-prepared shadow enclave, as they all have the same processing logic. With shadow fork, a MapReduce scheduler running in the host can instantiate multiple workers into different enclave instances with low startup latency and significantly save secure memory when running all workers in enclaves. During the processing, the mapper nodes invoke the map function on the input and produce intermediate key-value pairs. All the intermediate results are saved in memory and distributed to reducer nodes. The reducer nodes invoke the reduce function to produce the final result and return it to the host. We have implemented a prototype of MapReduce in PENGLAI with two settings: PENGLAI-Create and PENGLAI-ShadowFork. PENGLAI-Create creates each worker enclave using normal enclave creation, and PENGLAI-ShadowFork leverages the shadow enclave to fork a new worker instance. As shown in Figure 16 (b), PENGLAI-ShadowFork can achieve 2.0x lower latency over PENGLAI-Create when both systems use one mapper and one reducer. If we create more workers for the same job, PENGLAI-ShadowFork can gain 3.6x speedup compared with the PENGLAI-Create (4 mappers and 4 reduces, on a 4-core machine). The speedup mainly comes from the fast startup with shadow fork. As for normal startup, PENGLAI calculates enclave measurement every time. The tremendous attestation overhead will significantly affect the performance in the multi-workers situation. However, as shadow fork can reduce the overhead of secure memory initialization, enclave measurement is not the bottleneck for the whole job. Hence, PENGLAI-ShadowFork can gain better performance improvement with more worker enclaves.

6.6 Hardware Costs

We use Vivado [11] tool to generate the hardware and get the report of resource utilization in FPGA. The report shows that the overall hardware costs are small (0.56%–0.81% in LUT and 0.00% in RAM) over the original resources (the RISC-V core). It means that the extensions incur small costs, which is essential to add the extensions into real hardware.

7 Discussion

Architectures assumptions. Although the implementation of our prototype is based on RISC-V, the design is independent of specific architectures and can be adopted by other enclave systems. We highlight two major assumptions. First, the ISA should have a privileged level higher than the OS and hypervisor. This is a reasonable assumption for the prevailing ISA; cases include RISC-V's machine mode, ARM's EL3 mode, etc. Second, to support fine-grained memory management, the CPU shall have an MMU module, which is common in modern high-performance cores.

Security discussion. PENGLAI is designed to provide the same (or stronger) security guarantees compared with prior work. Besides the isolation and integrity protection mentioned above, PENGLAI can also defend against following attacks.

- **Controlled-channel attacks.** PENGLAI allows enclaves to validate the presence of some expected mappings, similar to Autarky [81]. The monitor will verify all these mappings when an enclave is invoked for the first time and check the validity when the OS changes the mapping. As the OS cannot directly access the enclave page tables, it cannot perform controlled-channel attacks by monitoring the access/dirty bits in page tables.
- **Cache-based side-channel attacks.** Existing enclaves still suffer from the cache-based side-channel attacks [74, 82, 107, 109, 110] or incur high overhead to solve it [45,

53, 75]. Our on-demand cache line locking mechanism (§4.4) defends the attacks with minor costs. A recent work, CURE [21], adopts a similar approach with on-demand partition cache. CURE binds the cache ways with certain enclave ID, while PENGLAI assigns cache ways to each core with less hardware modification, and is more suitable for scenarios like shared memory.

- **Other attacks.** There are some attacks caused by specific CPU bugs, including Foreshadow, Spectre [65], Meltdown [72], etc. The defense mechanisms are orthogonal to our design. We also believe that PENGLAI's monitor-assisted enclave design is more suitable for handling the emerging attacks than HW-based enclave systems, which are hard to update after release.

8 Related Work

Secure processor assisted enclave architecture. Some prior work uses a secure processor to support the trusted execution environment [12, 33, 39, 42, 44, 53, 70, 71, 85, 90, 104]. The Security-enhanced processor integrated with encryption and integrity engine can support compartments that are immune to both modification and observation. XOM [70, 71], SecureME [44] and SecureBlue++ [33] allow the trusted user processes running on the untrusted OS, as OS cannot obtain the plaintext content of the process. SEV and HyperCoffer [104] allow VMs to run on an untrusted hypervisor. However, such a method using encryption to isolate memory space brings non-negligible overhead, and the key management and traditional integrity scheme may restrict the enclave size and number.

Memory integrity scheme. Several different schemes have been proposed for memory integrity protection [38, 78, 85, 91, 92]. BMT [85] introduces the counter-based message authentication algorithm into the integrity protection scheme and reduces in-memory overhead. Bastion [91] unifies the integrity of in-memory pages and on-device pages. VAULT [92] adopts the various arity for different levels of tree nodes. However, all these prior work do not solve the inherent overhead in memory and SoC, and are not scalable at all. PENGLAI proposes the MMT with a mounting mechanism, which can reduce both on-die and in-memory overhead to achieve scalable memory integrity protection.

Virtualization-based isolation. Virtualization-based isolation [42, 60, 62, 91, 104, 111] has been researched in past decade. They rely on hypervisor to enhance the isolation among VMs using techniques like shadow page table [94, 98], nested/extended page table [31, 97], or HLAT [8]. These techniques have similarities to our Guarded Page Table—we both rely on higher privileged software to validate memory mappings. However, nested virtualization usually introduces non-trivial performance costs, e.g., tests show that the shadow PT can incur 40% overhead in Memcached [54] and 80% overhead in Redis [48], and Nested page table also causes 20% overhead in Memcached [54]. Shadow paging needs to re-construct shadow page table costly, and nested/extended page table incurs extra overhead due to 2-level page table walker during the TLB miss. However, PENGLAI proposes the Guarded Page Table and achieves page-grained isolation without introducing high performance overhead (only 5% even for memory-intensive benchmark: Redis). What's worse, the hypervisor and cloud service providers may not be trusted in cloud scenarios. Recent works utilize TEE techniques to propose secure VM, e.g., AMD SEV [12, 27, 61], Intel TDX [16] and vTZ [58], which can protect the VMs from the untrusted hypervisor. Nevertheless, the protection has defects, e.g., both SEV and TDX cannot defend against physical rollback attacks, and vTZ does not consider physical memory attacks. Also, the secure VM suffers the same performance degradation as the traditional VM due to the memory virtualization overhead. Compared with virtualization-based isolation methods, PENGLAI achieves better security guarantees, higher performance and scalability.

Cross-zone communication. Cross-zone communication or IPC has been extensively researched for microkernel and user-level processes [30, 49, 63, 95, 101]. SCONE [28] and HotCalls [103] optimize the host-enclave communication in SGX using asynchronous approaches, e.g., polling and shared untrusted memory. However, they are not suitable for E-E communication and tend to waste the CPU cycles. XPC [49] has proposed the ownership transfer based communication and reduced the remapping overhead. PENGLAI shares the same idea of XPC, but overcoming new challenges to transfer pages crossing the boundary between the secure and non-secure world. It outperforms existing enclave systems with the zero-copy and secure data transfer mechanism for both E-E and E-H communication.

9 Conclusion

This paper has presented a hardware-software co-design of scalable memory protection based on the PENGLAI enclave system. Our evaluation shows that PENGLAI can significantly optimize enclave number, secure memory capacity with integrity protection, enclave startup latency, as well as resource flexibility for both microbenchmarks and real-world applications.

10 Acknowledgments

We sincerely thank our shepherd Edouard Bugnion, Andrew Baumann and anonymous reviewers for their insightful suggestions. This work is supported in part by National Key Research and Development Program of China (No. 2020AAA0108500), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Yubin Xia is the corresponding author.

References

[1] Apache openwhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced December 2018.

[2] Arm corelink tzc-400 trustzone address space controller technical reference manual. https://developer.arm.com/documentation/ddi0504/c/. Referenced November 2020.

[3] Aws lambda - serverless compute. https://aws.amazon.com/lambda/. Referenced December 2018.

[4] Azure functions serverless architecture. https://azure.microsoft.com/en-us/services/functions/. Referenced December 2018.

[5] Google cloud function. https://cloud.google.com/functions/. Referenced December 2018.

[6] lambda-refarch-imagerecognition. https://github.com/aws-samples/lambda-refarch-imagerecognition. Referenced December 2019.

[7] Lmbench. http://lmbench.sourceforge.net/, 2005. Referenced Nov 2020.

[8] Hlat. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2018. Referenced May 2018.

[9] Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm, 2018. Referenced August 2018.

[10] Sifive. https://www.sifive.com/, 2018. Referenced November 2018.

[11] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html, 2018. Referenced August 2018.

[12] Amd secure encrypted virtualization (sev) - amd. https://developer.amd.com/sev/, 2019.

[13] Keystone | an open framework for architecting tees. https://keystone-enclave.org, 2019.

[14] Amd epyc 7002 series processors. https://www.amd.com/en/processors/epyc-7002-series, 2020. Referenced Aug. 2020.

[15] The intel sgx memory encryption engine. https://software.intel.com/content/www/us/en/develop/blogs/memory-encryption-an-intel-sgx-underpinning-technology.html, 2020. Referenced May 2020.

[16] Intle tdx. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html, 2020. Referenced Nov 2020.

[17] RISC-V Proxy Kernel. https://github.com/riscv/riscv-pk, 2020. Referenced May 2020.

[18] riscv/opensbi: Risc-v open source supervisor binary interface. https://github.com/riscv/opensbi, 2020.

[19] Sod - an embedded, modern computer vision and machine learning library. https://sod.pixlab.io, 2020. Referenced May 2020.

[20] wolfssl embedded ssl/tls library. https://www.wolfssl.com, 2020.

[21] CURE: A security architecture with customizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[22] Intel scalable software guard extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, 2021. Referenced Mar 2021.

[23] Intel software guard extensions. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html, 2021. Referenced Mar 2021.

[24] Penglai enclave. https://github.com/Penglai-Enclave, 2021. Referenced Mar 2021.

[25] 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 *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, pages 923–935, 2018.

[26] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.

[27] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020. Referenced Aug. 2020.

[28] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind,

Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[29] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[30] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. *ACM SIGOPS Operating Systems Review*, 23(5):102–113, 1989.

[31] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[32] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[33] Risk Boivie and Perter Williams. Secureblue++: Cpu support for secure execution. pages 1–9. IBM, IBM Research Division, 2012.

[34] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 645–650, 2018.

[35] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[36] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[37] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[38] David Champagne, Reouven Elbaz, and Ruby B Lee. The reduced address space (ras) for application memory authentication. In *International Conference on Information Security*, pages 47–63. Springer, 2008.

[39] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[40] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[41] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[42] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.

[43] Yi-Lin Cheng, Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. High resource utilization auto-scaling algorithms for heterogeneous container configurations. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 143–150, 2017.

[44] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, pages 108–119, 2011.

[45] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[46] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[48] Boris Teabe Djomgwe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. (no) compromis: Paging virtualization is not a fatality. In *VEE 2021-17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–12, 2021.

[49] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 671–684. ACM, 2019.

[50] 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.

[51] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, 1995. ACM.

[52] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[53] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8, 2012.

[54] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile paging: exceeding the best of nested and shadow paging. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016.

[55] Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[56] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[57] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11, 2013.

[58] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[59] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[60] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283. IEEE, 2011.

[61] David Kaplan. Protecting vm register state with sev-es. *White paper, Feb*, 2017.

[62] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361, 2010.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[64] 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.

[65] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[66] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 393–404. IEEE, 2009.

[67] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[68] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[69] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proceedings of the 48th International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14–19, 2021*. IEEE, 2021.

[70] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.

[71] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[72] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[73] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[74] 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.

[75] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324, 2013.

[76] Anupama Mampage, S. Karunasekera, and R. Buyya. Deadline-aware dynamic resource management in serverless computing environments. 2020.

[77] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[78] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[79] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 225–242, 2019.

[80] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[81] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[82] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[83] Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[84] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[85] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196. IEEE, 2007.

[86] Aakanksha Saha and Sonika Jindal. Emars: Efficient management and allocation of resources in serverless. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 827–830, 2018.

[87] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[88] 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.

[89] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[90] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.

[91] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. *ACM SIGPLAN Notices*, 47(4):437–450, 2012.

[92] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[93] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[94] Eric P Traut, Matthew D Hendel, and Rene Antonio Vega. Enhanced shadow page table algorithms, November 20 2007. US Patent 7,299,337.

[95] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[96] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[97] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[98] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[99] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[100] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[101] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 2016.

[102] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[103] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[104] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, pages 246–257, 2013.

[105] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–874, 2017.

[106] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360. IEEE, 2017.

[107] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World*, page 0. IEEE, 2019.

[108] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[109] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[110] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[111] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[112] 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.

[113] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41. IEEE, 2020.

NrOS: Effective Replication and Sharing in an Operating System

Ankit Bhardwaj[1], Chinmay Kulkarni[1], Reto Achermann[2], Irina Calciu[3], Sanidhya Kashyap[4], Ryan Stutsman[1], Amy Tai[3], and Gerd Zellweger[3]

[1]University of Utah, [2]University of British Columbia, [3]VMware Research, [4]EPFL

Abstract

Writing a correct operating system kernel is notoriously hard. Kernel code requires manual memory management and type-unsafe code and must efficiently handle complex, asynchronous events. In addition, increasing CPU core counts further complicate kernel development. Typically, monolithic kernels share state across cores and rely on one-off synchronization patterns that are specialized for each kernel structure or subsystem. Hence, kernel developers are constantly refining synchronization within OS kernels to improve scalability at the risk of introducing subtle bugs.

We present NrOS, a new OS kernel with a safer approach to synchronization that runs many POSIX programs. NrOS is primarily constructed as a simple, sequential kernel with no concurrency, making it easier to develop and reason about its correctness. This kernel is scaled across NUMA nodes using node replication, a scheme inspired by state machine replication in distributed systems. NrOS replicates kernel state on each NUMA node and uses operation logs to maintain strong consistency between replicas. Cores can safely and concurrently read from their local kernel replica, eliminating remote NUMA accesses.

Our evaluation shows that NrOS scales to 96 cores with performance that nearly always dominates Linux at scale, in some cases by orders of magnitude, while retaining much of the simplicity of a sequential kernel.

1 Introduction

Operating system kernels are notoriously hard to build. Manual memory management, complex synchronization patterns [36], and asynchronous events lead to subtle bugs [2–4], even when code is written by experts. Increasing CPU core counts and non-uniform memory access (NUMA) have only made it harder. Beyond correctness bugs, kernel developers must continuously chase down performance regressions that only appear under specific workloads or as core counts scale. Even so, prevailing wisdom dictates that kernels should use custom-tailored concurrent data structures with fine-grained locking or techniques like read-copy-update (RCU) to achieve good performance. For monolithic kernels, this slows development to the extent that even large companies like Google resort to externalizing new subsystems to userspace [57] where they can contain bugs and draw on a larger pool of developers.

Some have recognized that this complexity isn't always warranted. For example, wrapping a single-threaded, sequential microkernel in a single coarse lock is safe and can provide good performance when cores share a cache [67]. This approach does not target NUMA systems, which have many cores and do not all share a cache. Increased cross-NUMA-node memory latency slows access to structures in shared memory including the lock, causing collapse.

Multikernels like Barrelfish [17] take a different approach; they scale by forgoing shared memory and divide resources among per-core kernels that communicate via message passing. This scales well, but explicit message passing adds too much complexity and overhead for hosts with shared memory. Within a NUMA node, hardware cache coherence makes shared memory more efficient than message passing under low contention.

We overcome this trade-off between scalability and simplicity in NrOS, a new OS that relies primarily on *single-threaded*, sequential implementations of its core data structures. NrOS scales using node replication [28], an approach inspired by state machine replication in distributed systems, which transforms these structures into linearizable concurrent structures. Node replication keeps a separate replica of the kernel structures per NUMA node, so operations that read kernel state can concurrently access their local replica, avoiding cross-NUMA memory accesses. When operations mutate kernel state, node replication collects and batches them from cores within a NUMA node using *flat combining* [44], and it appends them to a shared log; each replica applies the operations serially from the log to synchronize its state.

The NrOS approach to synchronization simplifies reasoning about its correctness, even while scaling to hundreds of cores and reducing contention in several OS subsystems (§4.2,

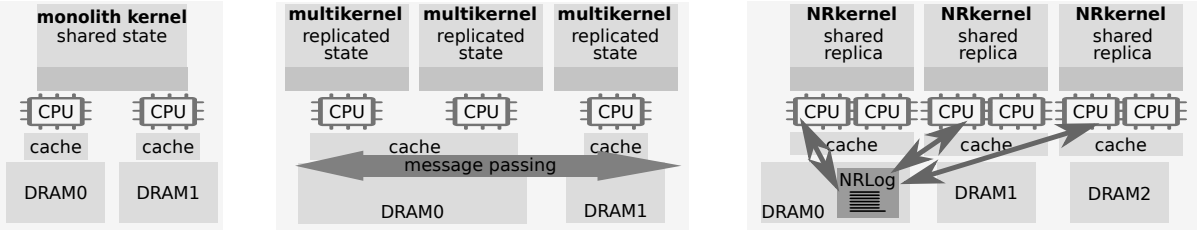


Figure 1: Architectural overview of NRkernel vs. multikernel and monoliths.

§4.4). However, node replication is not a panacea: while implementing an in-memory file system, we have encountered scenarios where frequent state mutations hinder performance. To address this challenge, we propose *concurrent node replication* (§3), which exploits operation commutativity with multiple logs and concurrent replicas to improve the performance and scalability of the file system (§4.3).

NrOS is implemented in Rust, which interplays with node replication. The Rust type system makes mutability explicit, enabling our node replication library to distinguish effortlessly between update operations that must use the log and read-only operations that can proceed concurrently at the local replica. However, we still had to solve several practical challenges, such as safely dealing with out-of-band reads by hardware, efficiently allocating buffers, and garbage collecting the log. To summarize, we make the following contributions.

1. We designed and implemented NrOS, a kernel that simplifies synchronization via node replication and runs many POSIX programs; we describe its subsystems, including processes, scheduling, address spaces, and an in-memory file system.
2. We implemented node replication in Rust, leveraging the Rust type system to distinguish mutable and read-only operations. In addition, we extended the node replication approach to exploit operation commutativity using multiple logs and concurrent replicas.
3. We evaluated NrOS on hosts with up to 4 NUMA nodes running bare metal and in virtual machines, and we compared its performance to that of Linux, sv6, and Barrelfish on file system, address space, and application-level benchmarks (LevelDB, memcached). NrOS largely outperforms conventional OSes on read-heavy workloads and on contending workloads thanks to its use of node replication.

NrOS[1] and its node replication[2] library are open source.

2 Background and Related Work

OS & Software Trends. Linux continues to be the prevalent data center OS; it uses a *monolithic kernel* (Figure 1), which shares all OS state and protects access using locks and other synchronization primitives. Despite being widely used, this model has multiple limitations. As the numbers of cores per server keeps increasing, the performance and scalability of the kernel are impacted. Its synchronization primitives, in particular, do not scale well to large numbers of cores. Moreover, architectures such as non-uniform memory access (NUMA) exacerbate scalability problems. With NUMA, each processor has lower latency and higher bandwidth to its own local memory. This causes significant performance degradation when state is shared across all processors [23].

The Linux community has reacted with point solutions to these problems, optimizing performance through fine-grained synchronization [31, 32, 36] or better locks/wait-free data structures [51, 61, 73]. However, these solutions have amplified the looming problem of complexity with the monolithic design. Correct concurrency in the kernel in the presence of fine-grained locking and lock-free data structures is hard, and it is the source of many hard-to-find bugs [26, 34, 37–39, 41, 47, 55, 79, 80].

For example, Linux has an entire class of bugs due to lockless access to lock-protected shared variables [2–4]. Use-after-free bugs are also common in Linux due to improper locking [5, 6]. These are some of the many bugs that still exist because of the heavy use of lockless or improper accesses in the Linux kernel. Such bugs would be trivially avoided in NrOS, which absolves the developer from reasoning about fine-grained locking for concurrent data structures.

The OS research community has proposed new kernel models to address these limitations [22, 30, 70, 78]. In particular, the *multikernel* model replicates kernel state and uses message passing to communicate between replicas [17], but, a decade after its introduction, multikernels have not seen widespread adoption. In part, this is because cache coherence and shared memory have not gone away (and probably will not in the foreseeable future). For example, the Barrelfish multikernel mainly uses point-to-point message passing channels between the OS nodes, avoiding use of shared memory between cores within the kernel altogether. Ultimately, this means the system needs to use n^2 messages to communicate between cores, and each core must monitor n queues; this has little benefit to offset its cost since many large groups of cores can already communicate more efficiently via shared memory access to a shared last-level cache. This also increases the number of operations that must be coordinated across cores, and some

[1] https://github.com/vmware-labs/node-replicated-kernel
[2] https://github.com/vmware/node-replication

operations require that all kernel nodes reach agreement. For example, capability operations in Barrelfish require a blocking two-phase protocol so complex that it is explicitly encoded as a large state machine, and the full Barrelfish capability system is about 8,000 lines of code not including the two domain-specific languages used for RPCs and capability operations. So, despite their scaling benefits, multikernels fail to exploit the simplicity and performance of shared memory even when it is efficient to do so.

Prior work has investigated approaches to combine the monolithic and the multikernel models [16, 74] or to apply replication for parts of the kernel [20, 25, 33, 72]. Tornado [42] and K42 [54] use clustered objects, which optimize shared state through the use of partitioning and replication. More recently, Mitosis [10] retrofitted the replication idea to page table management in the Linux kernel, and showed benefits for a wide variety of workloads. Implementing Mitosis required a major engineering effort to retrofit replication into a conventional kernel design.

Our main observation is that shared memory can be used judiciously with some replication to get the best of both worlds: a simple, elegant and extensible model like the multikernel that can run applications designed for a monolithic kernel. Based on this observation, we propose the *NRkernel*, a new multikernel model that replicates the kernel, but allows replica sharing among cores, balancing performance and simplicity.

The NRkernel is inspired by NR (§2.1), and it has at its core operation logs that provide linearizability. The logs act as global broadcast message channels that eliminate the complex state machine used by Barrelfish for consensus. What remains are simple, single-threaded implementations of data structures that apply updates from the shared log. Based on the NRkernel model, we designed and implemented NrOS, a representative system to evaluate its characteristics.

Hardware Trends. Two major hardware trends motivate the NRkernel. First, shared memory is part of every system today, and current hardware trends indicate that there will be some form of memory sharing – not necessarily with global coherence – available for many new heterogeneous components and disaggregated architectures. The industry is working on multiple specifications that will enable such sharing (*e.g.,* CXL [71], CAPI [64], CCIX [1], Gen-Z [7]). While sharing memory does not scale indefinitely as we add more cores, it works more efficiently than passing messages among a limited number of cores [27]. In such a model, a shared log and replication work well because the log can be accessed by all independent entities connected over a memory bus.

Second, main memory capacities are growing [40] and are expected to increase further with new 3D-stacked technologies [76] and the arrival of tiered memory systems comprising various types of memory, such as DRAM and SCM. Amazon already has offerings for servers with up to 24 TiB. Like other systems [49, 68], the NRkernel leverages the abundance of memory to improve performance with replication.

2.1 Node Replication (NR)

NR creates a linearizable NUMA-aware concurrent data structure from a sequential data structure [28]. NR replicates the sequential data structure on each NUMA node, and it uses an operation log to maintain consistency between the replicas. Each replica benefits from read concurrency using a readers-writer lock and from write concurrency using a technique called *flat combining*. Flat combining batches operations from multiple threads to be executed by a single thread (*the combiner*) per replica. This thread also appends the batched operations to the log using a single atomic operation for the entire batch; other replicas read the log and update their local copy of the structure with the logged operations.

NR relies on three main techniques to scale well:

(1) The operation log uses a circular buffer to represent the abstract state of the concurrent data structure. Each entry in the log represents a mutating operation, and the log ensures a total order among them. The *log tail* gives the index to the last operation added to the log. Each replica consumes the log lazily and maintains a per-replica index into the log that indicates which operations of the log have been executed on its copy of the structure. The log is implemented as a circular buffer of entries that are reused. NR cannot reuse entries that have not been executed on all replicas. This means at least one thread on each NUMA node must occasionally make progress in executing operations on the data structure, otherwise the log could fill up and block new mutating operations. Section 4 discusses how NrOS addresses this.

(2) Flat combining [44] in NR allows threads running on the same NUMA node to share a replica, resulting in better cache locality both from flat combining and from maintaining the replica local to the node's last-level cache. The combiner also benefits from batching by allocating log space for all pending operations at a replica with a single atomic instruction.

(3) The optimized readers-writer lock in NR is a writer-preference variant of the distributed readers-writer lock [75] that ensures correct synchronization between the combiner and reader threads when accessing the sequential replica. This lock lets readers access a local replica while the combiner is adding a batch of operations to the log, increasing parallelism.

NR executes updates and reads differently:

A concurrent mutating operation (update) needs to acquire the combiner lock on the local NUMA node to add the operation to the log and to execute the operation against the local replica. If the thread T executing this operation fails to acquire it, another thread is the combiner for the replica already, so T spin-waits to receive its operation's result from the existing combiner. If T acquires the lock, it becomes *the combiner*. The combiner first flat combines all operations from all threads that are concurrently waiting for their update operations to be appended to the log with a single compare-and-swap. Then, the combiner acquires the writer lock on the local replica's structure, and it sequentially executes all

unexecuted update operations in the log on the structure in order. For each executed operation, the combiner returns its results to the waiting thread.

A concurrent non-mutating operation (read) can execute on its thread's NUMA-node-local replica without creating a log entry. To ensure that the replica is not stale, it takes a snapshot of the log tail when the operation begins, and it waits until a combiner updates the replica past the observed tail. If there is no combiner for the replica, the reading thread becomes the combiner to update the replica before executing its read operation.

NR simplifies concurrent data structure design by hiding the complexities of synchronization behind the log abstraction. NR works well on current systems because the operation log is optimized for NUMA. We adopt NR's NUMA-optimized log design, but we use it to replicate kernel state.

Linearizable operation logs are ubiquitous in distributed systems. For example, many protocols such as Raft [63], Corfu [35], and Delos [15] use a log to simplify reaching consensus and fault tolerance, as well as to scale out a single-node implementation to multiple machines. Recently, the same abstraction has been used to achieve good scalability on large machines both in file systems [19] and general data structures [24, 44, 52, 58, 69]. Concurrent work has developed NUMA-aware data structures from persistent indexes [77].

2.2 NR Example

Listing 1 shows an example where a Rust standard hashmap is replicated using NR. `NRHashMap` wraps an existing sequential hashmap (line 2-4). Programs specify the read (line 7) and update (line 10) operations for the structure and how each should be executed at each replica (lines 20-31) by implementing the `Dispatch` trait.

Listing 2 shows a program that creates a single `NRHashMap` with two NR replicas that use a shared log to synchronize update operations between them. The code creates a log (line 3) which is used to create two replicas (lines 6-7). Finally, the threads can register themselves with any replica and issue operations against it (lines 14-15). NR supports any number of replicas and threads; programs must specify a configuration that is efficient for their structure and operations. For example, NrOS allocates one replica per NUMA node, and each core in a node registers with its NUMA-local replica in order to benefit from locality.

```
1  // Standard Rust hashmap node replicated to each NUMA node.
2  pub struct NRHashMap {
3    storage: HashMap<usize, usize>,
4  }
5
6  // NRHashMap has a Get(k) op that does not modify state.
7  pub enum HMReadOp { Get(usize) }
8
9  // NRHashMap has a Put(k,v) op that modifies replica state.
10 pub enum HMUpdateOp { Put(usize, usize) }
11
12 // The trait implementation describes how to execute each
13 // operation on the sequential structure at each replica.
14 impl Dispatch for NRHashMap {
15   type ReadOp = HMReadOp;
16   type UpdateOp = HMUpdateOp;
17   type Resp = Option<usize>;
18
19   // Execute non-mutating operations (Get).
20   fn dispatch(&self, op: Self::ReadOp) -> Self::Resp {
21     match op {
22       HMReadOp::Get(k) => self.storage.get(&k).map(|v| *v),
23     }
24   }
25
26   // Execute mutating operations (Put).
27   fn dispatch_mut(&mut self, op: Self::UpdateOp) ->
       Self::Resp {
28     match op {
29       HMUpdateOp::Put(k, v) => self.storage.insert(k, v),
30     }
31   }
32 }
```

Listing 1: Single-threaded hashmap transformed using NR.

```
1  // Allocate an operation log to synchronize replicas.
2  let logsize = 2 * 1024 * 1024;
3  let log = Log::<<NRHashMap as
       Dispatch>::UpdateOp>::new(logsize);
4
5  // Create two replicas of the hashmap (one per NUMA node).
6  let replica1 = Replica::<NRHashMap>::new(log);
7  let replica2 = Replica::<NRHashMap>::new(log);
8
9  // Register threads on one NUMA node with replica1.
10 let tid1 = replica1.register();
11 // Threads on other node register similarly with replica2.
12
13 // Issue Get and Put operations and await results.
14 let r = replica1.execute(HMReadOp::Get(1), tid1);
15 let r = replica1.execute_mut(HMUpdateOp::Put(1, 1), tid1);
```

Listing 2: Creating replicas and using NR.

3 Concurrent Node Replication (CNR)

For some OS subsystems with frequent mutating operations (*e.g.,* the file system) NR's log and sequential replicas can limit scalability. Multiple combiners from different replicas can make progress in parallel, but write scalability can be limited by appends to the single shared log and the per-replica readers-writer lock, which only allows one combiner to execute operations at a time within each replica.

To solve this, we extend NR to exploit operation commutativity present in many data structures [30, 45]. Two operations are *commutative* if executing them in either order leaves the structure in the same abstract state. Otherwise, the operations are *conflicting*. Like NR, CNR replicates a data structure across NUMA nodes and maintains consistency between replicas. However, CNR scales the single shared NR log to multiple logs by assigning commutative operations to different logs. Conflicting operations are assigned to the same log,

which ensures they are ordered with respect to each other. Also, CNR can use concurrent or partitioned data structures for replicas, which allows multiple concurrent combiners on each replica – one per shared log. This eliminates the per-replica readers-writer lock and scales access to the structure.

CNR transforms *an already concurrent data structure* to a NUMA-aware concurrent data structure. The original data structure can be a concurrent (or partitioned) data structure that works well for a small number of threads (4-8 threads) within a single NUMA node. This data structure can be lock-free or lock-based and may exhibit poor performance under contention. CNR transforms such a concurrent data structure to one that works well for a large number of threads (*e.g.,* 100s of threads) across NUMA nodes and is resilient to contention.

Similar to transactional boosting [45], CNR only considers the abstract data type for establishing commutativity, not the concrete data structure implementation. For example, consider a concurrent hashmap with an *insert(k, v)* operation. One might think that *insert(k, v)* and *insert(k+1, v′)* are not commutative because they may conflict on shared memory locations. However, the original data structure is concurrent and already safely orders accesses to shared memory locations; hence, these operations commute for CNR and can be safely executed concurrently.

CNR's interface is nearly identical to NR's interface, but it introduces *operation classes* to express commutativity. Implementers of a structure provide functions that CNR uses to map each operation to an operation class. These functions map conflicting operations to the same class, and each class is mapped to a log. Hence, if two conflicting operations execute on the same NUMA node they are executed by the same combiner, which ensures they are executed in order. In contrast, commutative operations can be executed by different combiners and can use different shared logs, allowing them to be executed concurrently.

Overall, CNR increases parallelism within each NUMA node by using a concurrent replica with multiple combiners, and it increases parallelism across NUMA nodes by using multiple (mostly) independent shared logs. However, ultimately every update operation must be executed at all replicas; hence, it comes at a cost, and it cannot scale update throughput beyond that of a single NUMA node. We refer to the general mechanism of replicating a data structure using operation logs as NR; when we need to explicitly distinguish cases that rely on a concurrent data structure with multiple logs (rather than a sequential one with a single log) we use the term CNR.

3.1 CNR Example

The code to use CNR to scale `Put` throughput for a replicated hashmap is almost identical to the example given in Section 2.2; it only changes in two ways. First, the structure embedded in each replica must be thread-safe, since (commutative) operations are executed on it concurrently, *i.e.,* it must implement Rust's `Sync` trait. This creates a subtle, mostly inconsequential, distinction in CNR's `Dispatch` trait because a mutable reference is not required to execute an operation on the structure; hence, Listing 1 line 27 would read `&self` rather than `&mut self`.

```
impl LogMapper for HMUpdateOp {
  fn hash(&self, nlogs: usize, logs: Vec<usize>) {
    logs.clear();
    match self {
      HMUpdateOp::Put(key, _v) => logs.push(*key % nlogs),
    }
  }
}
```

Listing 3: LogMapper implementation for update operations.

Second, CNR uses multiple logs to scale update operations; programs must indicate which operations commute so CNR can distribute commuting operations among logs. To do this, programs implement the `LogMapper` trait for their update operations (Listing 3). The program must implement this trait for read operations as well. `Get` and `Put` operations on a hashmap commute unless they affect the same key, so this example maps all operations with a common key hash to the same class and log. CNR also allows passing multiple logs to the replicas; otherwise, its use is similar to Listing 2. Some operations may conflict with operations in multiple classes, which we discuss in the next section, so a `LogMapper` may map a single operation to more than one class/log.

3.2 Multi-log Scan Operations

In addition to read and update operation types, CNR adds a scan operation type, instances of which belong to more than one operation class. These are operations that conflict with many other operations. Often these are operations that involve the shape of the structure or that need a stable view of many elements of it. Examples include returning the count of elements in a structure, hashmap bucket array resizing, range queries, or, in our case, path resolution and directory traversal for file system `open`, `close`, and `rename` operations. If these operations were assigned to a single class, all other operations would need to be in the same class, eliminating any benefit from commutativity.

Scan operations conflict with multiple operation classes, so they must execute over a consistent state of the replica with respect to all of the classes and logs involved in the scan obtained after its invocation. To obtain this *consistent state*, the thread performing the scan creates an atomic snapshot at the tails of the logs involved in the operation. Then, the replica used by the scan needs to be updated *exactly* up to the snapshot without exceeding it (unlike NR read operations, which can update past the read thread's observed log tail).

Hence, there are two challenges that CNR needs to solve for a correct scan operation: (1) obtaining the atomic snapshot of the log tails while other threads might be updating the logs;

and (2) ensuring that the replica is updated *exactly* up to the observed atomic snapshot.

The problem of obtaining an atomic snapshot is well-studied in concurrent computing [11, 14, 56]. Unlike prior solutions, which are wait-free, we designed a simple, blocking solution that works well in practice and addresses both of the above challenges simultaneously. The scan thread inserts the operation into all of the logs associated with the scan's operation classes. To ensure that two threads concurrently inserting scan operations do not cause a deadlock by inserting the operations in a different order on different logs, each thread must acquire and hold a *scan-lock* while inserting a scan operation in the logs that participate in the operation's atomic snapshot. Update threads can continue to insert their operations into the logs after the unfinished scan and without holding the scan-lock. These operations will be correctly linearized after the scan. Update threads from the same replica as the scan block when they encounter a colocated unfinished scan. With updates blocked on the replica, the scan thread can proceed to execute the operation on the replica once the replica is updated up to the scan's log entries (either by the scan thread or by combiners). After the scan has been executed at that replica, blocked threads at the replica continue with their updates.

Similar to NR read and update operations, scan operations can be of type scan-update (if the scan modifies the data structure) or scan-read (if it does not). With a scan-read operation, the operation only needs to be performed once at the replica where it was invoked; the other replicas ignore log entries for scan-read operations. Like update operations, scan-update operations must affect all replicas, but they must also be executed at each replica only when the replica is at a consistent state for that scan operation. The first combiner that encounters the scan-update operation on a replica acquires all necessary combiner locks, updates the replica to the consistent state, and executes the scan, just as is done on the replica where the scan was initiated.

Scan operations incur higher overhead than other operations, and their cost depends on how many operation classes they conflict with. In our experience, scan operations are rare, so CNR is carefully designed so that scans absorb the overhead while leaving the code paths for simpler and more frequent (read and update) operations efficient.

4 NrOS Design

We designed and implemented NrOS, a representative system for the NRkernel. The overall NrOS kernel as a whole is designed around per-NUMA node *kernel replicas* that encapsulate the state of most of its standard kernel subsystems. Kernel operations access the local replica, and state inside replicas is modified and synchronized with one another using NR, so cross-node communication is minimized (Figure 2). The collective set of kernel replicas act as a multikernel.

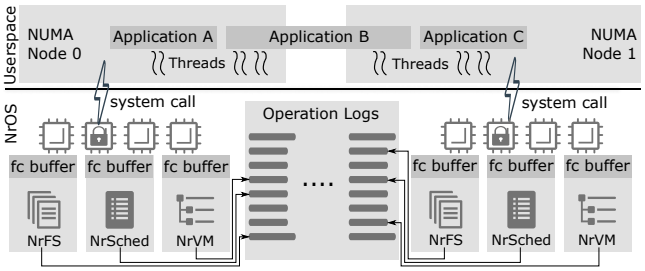


Figure 2: NrOS overview. A per-NUMA-node kernel replica services syscalls on cores on that node. Operations read local replica state; state mutating operations are replicated via NR data structures and are executed at all replicas. Each replica flat combines operations from all cores within a NUMA node, efficiently appending them to one (for NR) or multiple (for CNR) logs. Operations block until completion, ensuring linearizability.

Here, we describe the design of three of NrOS's major subsystems; all of them are replicated via NR:

NR-vMem (§4.2): A virtual memory system that replicates per-process page mapping metadata and hardware page tables for a given process.

NR-FS (§4.3): An in-memory file system that, by design, replicates all metadata and file contents.

NR-Scheduler (§4.4): A process manager that loads and spawns ELF binaries with replicated (read-only) ELF sections and makes global scheduling decisions.

In userspace, NrOS runs native NrOS programs or POSIX programs that are linked against NetBSD libOS components (§5). The libOS uses system calls for memory, process management, and file system operations. This approach can run many POSIX applications like memcached (§6.3.3), Redis, and LevelDB (§6.2.2), though it lacks `fork` and some other functionality which it would need for full POSIX support. If a process wants to write to a file or map a page, it traps into the kernel and enqueues an operation against its local replica file system or process structure and blocks until the request completes before returning from the syscall. Operations that must modify state acquire a local writer lock (if the data structure is not using CNR) before applying the operation directly. If the lock is contended, the operation is enqueued in the local flat combining buffer, then it waits until the current combiner within the node applies the operation. The logs make sure that all modifications to the state of all of the above components are replicated to the other kernel replicas through NR. Operations that only read replica state first ensure the local replica is up-to-date by applying any new operations from the NR log (if needed), and then reading from the local replica.

The kernel is also responsible for physical memory management, interrupt routing and PCI pass-through (network I/O is done directly by processes). These subsystems are not replicated using NR. Devices and interrupts are functional, but their details are outside of the scope of this paper. The full functionality provided by the kernel can be best com-

pared with a lightweight hypervisor which runs processes as containerized/isolated applications using the libOS.

NRkernel Principles. Overall NrOS represents a new point in the multikernel design space we call the *NRkernel* model, which is encapsulated by three key principles.

(1) Combining replicated and shared state. Multikernels like Barrelfish rely on per-core replicas which are prohibitively expensive; NRkernels strike a balance by maintaining a kernel replica per NUMA node; within a NUMA node cores share access to their NUMA-local replica. This maximizes the benefit of shared last-level caches while eliminating slow cross-NUMA node memory accesses. With per-NUMA-node replicas, memory consumption grows with the number of NUMA nodes rather than the number of cores.

(2) Replica consistency via an operation log. Unlike multikernels' expensive use of message passing between all pairs of cores, in NRkernels kernel replicas efficiently synchronize with shared operation logs; logging scales within a NUMA node using flat combining to batch operations across cores. The logs encode all state-changing operations for each subsystem, and ensure replica consistency while hiding the details of architecture-specific performance optimizations.

(3) Compiler-enforced memory and concurrency safety. Rust's compile-time memory-safety and thread-safety guarantees are easy to extend to a kernel's NR implementation. Its segregation of mutating and non-mutating references ensures correct, efficient code is generated where each kernel operation safely accesses the local replica when possible or is logged to synchronize replicas. Rust's `Send` and `Sync` annotations for types are a helpful mechanism to prevent putting data structures on the log that have meaning only on a local core (*e.g.,* a userspace pointer) and prevents them from ever being accessed by another core due to flat combining.

Encapsulating concurrency concerns in a single library with compiler-checked guarantees ensures most operations scale without concerns about subtle concurrency bugs. Having NR isolated in a single logical library also makes it easier to reason about concurrency correctness. For instance, in future work we plan to formally verify the NR mechanism, which would guarantee correct translation for any data structure that leverages NR for concurrency. Contrast this with a traditional kernel such as Linux, where bugs can be introduced not only in the lock library implementation (such as RCU) but especially in the way the rest of the kernel uses the library; only in 2019 did kernel developers consolidate the Linux RCU library to prevent users from mismatching locking calls [8, 59].

In the remainder of this section, we describe NrOS's subsystems, which demonstrate these principles and resolve the challenges of putting them into practice.

4.1 Physical Memory Management

Physical memory allocation and dynamic memory allocation for kernel data structures are the two basic subsystems that do not use NR. Replicated subsystems often require physical frames, but that allocation operation itself should not be replicated. For example, when installing a mapping in a page table, each page table entry should refer to the same physical page frame on all replicas (though, each replica should have its own page tables). If allocator state were replicated, each allocation operation would be repeated on each replica, breaking this. As a result, some syscalls in NrOS must be handled in two steps. For example, when installing a page, the page is allocated up front, outside of NR, and a pointer to it is passed as an argument to the NR operation. This also helps with performance; zeroing a page is slow, and it can be done before the replicated NR operation is enqueued. Operations from a log are applied serially at each replica, so this optimization eliminates head-of-line-blocking on zeroing.

At boot time, the affinity for memory regions is identified, and memory is divided into per-NUMA node caches (`NCache`). The `NCache` statically partitions memory further into two classes of 4 KiB and 2 MiB frames. Every core has a local cache `TCache` of 4 KiB and 2 MiB frames for fast, no-contention allocation when it contains the requested frame size. If it is empty, it refills from its local `NCache`. Similar to slab allocators [21], NrOS `TCache` and `NCache` implement a cache frontend and backend that controls the flow between `TCaches` and `NCaches`.

Unlike Barrelfish or seL4 [53] where all dynamic memory management is externalized to userspace, NrOS makes use of dynamic memory allocation in the kernel. For arbitrary-sized allocations, NrOS implements a simple, scalable allocator with per-core, segregated free lists of 2 MiB or 4 KiB frames. Each frame is divided into smaller, equal-sized objects. A bit field tracks per-object allocations within a frame.

Since NrOS is implemented in Rust, memory management is greatly simplified by relying on the compiler to track the lifetime of allocated objects. This eliminates a large class of bugs (use-after-free, uninitialized memory, *etc.*), but the kernel still has to explicitly handle running out of memory. NrOS uses fallible allocations to handle out-of-memory errors gracefully by returning an error to applications.

However, handling out-of-memory errors in presence of replicated data structures becomes challenging: Allocations that happen to store replicated state must be deterministic (*e.g.,* they should either succeed on all replicas or none). Otherwise, the replicas would end up in an inconsistent state if after executing an operation, some replicas had successful and some had unsuccessful allocations. Making sure that all replicas always have equal amounts of memory available is infeasible because every replica replicates at different times, and allocations can happen on other cores for outside of NR. We solve this problem in NrOS by requiring that all memory allocations for state within node replication or CNR must go through a deterministic allocator. In the deterministic allocator, the first replica that reaches an allocation request allocates memory on behalf of all other replicas too. The deterministic

allocator remembers the results temporarily, until they are picked up by the other replicas which are running behind. If an allocation for any of the replica fails, the leading replica will enqueue the error for all replicas to ensure that all replicas always see the same result. Allocators in NrOS are chainable, and it is sufficient for the deterministic allocator to be anywhere in the chain so it doesn't necessarily have to be invoked for every fine-grained allocation request. Our implementation leverages the custom allocator feature in Rust, which lets us override the default heap allocator with specialized allocators for individual data structures.

4.2 Virtual Memory Management

NrOS relies on the MMU for isolation. Like most conventional virtual memory implementations, NrOS uses a per-process mapping database (as a B-Tree) to store frame mappings which is used to construct the process's hardware page tables. NrOS currently does not support demand paging. Due to increased memory capacities, we did not deem demand paging an important feature for demonstrating our prototype. Both the B-Tree and the hardware page tables are simple, sequential data structures that are wrapped behind the node replication interface for concurrency and replication. Therefore, the mapping database and page tables are replicated on every NUMA node, forming the NR-vMem subsystem. NR-vMem exposes the following mutating operations for a process to modify its address space: `MapFrame` (to insert a mapping); `Unmap` (to remove mappings); and `Adjust` (to change permissions of a mapping). NR-vMem also supports a non-mutating `Resolve` operation (that advances the local replica and queries the address space state).

There are several aspects of NR-vMem's design that are influenced by its integration with node replication.

For example, NR-vMem has to consider out-of-band read accesses by cores' page table walkers. Normally a read operation would go through the node replication interface, ensuring replay of all outstanding operations from the log first. However, a hardware page table walker does not have this capability. A race arises if a process maps a page on core X of replica A and core Y of replica B accesses that mapping in userspace before replica B has applied the update. Luckily, this can be handled since it generates a page fault. In order to resolve this race, the page fault handler advances the replica by issuing a `Resolve` operation on the address space to find the corresponding mapping of the virtual address generating the fault. If a mapping is found, the process can be resumed since the `Resolve` operation will apply outstanding operations. If no mapping is found, the access was an invalid read or write by the process.

`Unmap` or `Adjust` (*e.g.,* removing or modifying page table entries) requires the OS to flush TLB entries on cores where the process is active to ensure TLB coherence. This is typically done in software by the OS (and commonly referred to as performing a TLB "shootdown"). The initiator core starts by enqueuing the operation for the local replica. After node replication returns it knows that the unmap (or adjust) operation has been performed at least against the local page table replica and that it is enqueued as a future operation on the log for other replicas. Next, it sends inter-processor interrupts (IPIs) to trigger TLB flushes on all cores running the corresponding process. As part of the IPI handler the cores first acknowledge the IPI to the initiator. Next, they advance their local replica to execute outstanding log operations (which forces the unmap/adjust if it was not already applied). Then, they poll a per-core message queue to get information about the regions that need to be flushed. Finally, they perform the TLB invalidation. Meanwhile the initiator invalidates its own TLB entries, and then it waits for all acknowledgments from the other cores before it returns to userspace. This shootdown protocol incorporates some of the optimizations described in Amit *et al.* [12]; it uses the cluster mode of the x86 interrupt controller to broadcast IPIs up to 16 CPUs simultaneously, and acknowledgments are sent to the initiator as soon as possible when the IPI is received (this is safe since flushing is done in a non-interruptible way).

4.3 File System

File systems are essential for serving configuration files and data to processes. NR-FS adds an in-memory file system to NrOS that supports some standard POSIX file operations (`open`, `pread`, `pwrite`, `close`, *etc.*). NR-FS tracks files and directories by mapping each path to an inode number and then mapping each inode number to an in-memory inode. Each inode holds either directory or file metadata and a list of file pages. The entire data structure is wrapped by node replication for concurrent access and replication.

There are three challenges for implementing a file system with node replication. First, historically POSIX read operations mutate kernel state (*e.g.,* file descriptor offsets). State mutating operations in node replication must be performed at each replica serially, which would eliminate all concurrency for file system operations. Fortunately, file descriptor offsets are implemented in the userspace libOS, and all NrOS file system calls are implemented with positional reads and writes (pread/pwrite), which do not update offsets. This lets NR-FS apply read operations as concurrent, non-mutating operations.

Second, each file system operation can copy large amounts of data with a single read or write operation. The size of the log is limited, so we do not copy the contents into it. Instead we allocate the kernel-side buffers for these operations and places references to the buffers in the log. These buffers are deallocated once all replicas have applied the operation.

Third, processes supply the buffer for writes, which can be problematic for replication. If a process changes a buffer during the execution of a write operation, it could result in inconsistencies in file contents since replicas could copy data

into the file from the buffer at different times. In NR-FS the write buffer is copied to kernel memory beforehand. This also solves another problem with flat combining: cores are assigned to processes (§4.4) and any core within a replica could apply an operation, but a particular core may not be in the process's address space. Copying the data to kernel memory before enqueuing the operation ensures that the buffer used in the operation is not modified during copies and is readable by all cores without address space changes.

4.3.1 Scaling NR-FS Writes

NR-FS optimizes reads so that many operations avoid the log, but other operations (`write`, `link`, `unlink`, *etc.*) must always be logged. This is efficient when these operations naturally contend with one another since they must serialize anyway and can benefit from flat combining. However, sometimes applications work independently and concurrently on different regions of the file system. For those workloads, node replication would be a limiting bottleneck as it unnecessarily serializes those operations.

To solve this, we developed CNR (§3). CNR uses the same approach to replication as node replication, but it divides commutative mutating operations among multiple logs with a combiner per log to scale performance. Others have observed the benefits of exploiting commutativity in syscalls and file systems [19, 30, 60], and CNR lets NR-FS make similar optimizations. CNR naturally scales operations over multiple combiners per NUMA node under low contention workloads that mutate state, and it seamlessly transitions to use a single combiner when operations contend.

Augmenting NR-FS to use CNR mainly requires implementing the `LogMapper` trait that indicates which log(s) an operation should serialize in (Listing 3). NR-FS hash partitions files by inode number, so operations associated with different files are appended to different logs and applied in parallel.

Some operations like `rename` may affect inodes in multiple partitions. Our current version of NR-FS handles this by serializing these operations with operations on all logs as a scan-update (§3.2). Using scans ensures that if the effect of any cross-partition operation (like `rename`) could have been observed by an application, then all operations that were appended subsequently to any log linearize after it (external consistency). We plan to experiment in the future with more sophisticated approaches that avoid serializing all operations with every such cross-partition operation.

4.4 Process Management and Scheduling

Process management for userspace programs in NrOS is inspired by Barrelfish's "dispatchers" and the "Hart" core abstraction in Lithe [65] with scheduler activations [13] as a notification mechanism.

In NrOS, the kernel-level scheduler (NR-Scheduler) is a coarse-grained scheduler that allocates CPUs to processes. Processes make system calls to request for more cores and to give them up. The kernel notifies processes of core allocations and deallocations via upcalls. To run on a core, a process allocates executor objects (*i.e.,* the equivalent of a "kernel" thread) that are used to dispatch a given process on a CPU. An executor mainly consists of two userspace stacks (one for the upcall handler and one for the initial stack) and a region to save CPU registers and other metadata. Executors are allocated lazily but a process keeps a per-NUMA-node cache to reuse them over time.

In the process, a userspace scheduler reacts to upcalls indicating the addition or removal of a core, and it makes fine-grained scheduling decisions by dispatching threads accordingly. This design means that the kernel is only responsible for coarse-grained scheduling decisions, and it implements a global policy of core allocation to processes.

NR-Scheduler uses a sequential hash table wrapped with node replication to map each process id to a process structure and to map process executors to cores. It has operations to create or destroy a process, to allocate and deallocate executors for a process, and to obtain an executor for a given core.

Process creation must create a new address space, parse the program's ELF headers, allocate memory for program sections, and relocate symbols in memory. A naive implementation might apply those operations on all replicas using node replication, but this would be incorrect. It is safe to independently create a separate read-only program section (like `.text`) for the process by performing an operation at each of the replicas. However, this would not work for writable sections (like `.data`), since having independent allocations per replica would break the semantics of shared memory in the process. Furthermore, we need to agree on a common virtual address for the start of the ELF binary, so position independent code is loaded at the same offset in every replica.

As a result of this, process creation happens in two stages, where operations that cannot be replicated are done in advance. The ELF program file must be parsed up front to find the writable sections, to allocate memory for them, and to relocate entries in them. After that, these pre-loaded physical frames and their address space offsets are passed to the replicated NR-Scheduler create-process operation. Within each replica, the ELF file is parsed again to load and relocate the read-only program sections and to map the pre-allocated physical frames for the writable sections.

Removing a process deletes and deallocates the process at every replica, but it also must halt execution on every core currently allocated to the process. Similar to TLB shootdowns, this is done with inter-processor interrupts and per-core message queues to notify individual cores belonging to a replica.

4.5 Log Garbage Collection

As described in Section 2.1, operation logs are circular buffers, which fixes the memory footprint of node replication. However, entries can only be overwritten once they have been applied at all replicas; progress in applying entries at a replica can become slow if operations are rare at that replica (*e.g.,* if cores at one replica spend all of their time in userspace).

NrOS solves this in two ways. First, if a core at one replica cannot append new operations because another replica is lagging in applying operations, then it triggers an IPI to a core associated with the lagging replica. When the core receives the IPI, it immediately applies pending log operations to its local replica, unblocking the stalled append operation at the other replica. On initialization, NrOS provides a callback to the node replication library that it can use to trigger an IPI; the library passes in the id of the slow replica and the id of the log that is low on space. Second, frequent IPIs are expensive, so NrOS tries to avoid them by proactively replicating when cores are idle. So long as some core at each replica sometimes has no scheduled executor, IPIs are mostly avoided.

Finally, some operations hold references to data outside of the log that may need to be deallocated after an operation has been applied at all replicas (*e.g.,* buffers that hold data from file system writes). If deallocation of these resources is deferred until a log entry is overwritten, then large pools of allocated buffers can build up, hurting locality and putting pressure on caches and TLBs. To more eagerly release such resources, these references embed a reference count initialized to the number of replicas, which is decremented each time the operation is applied at a replica; when the count reaches zero, the resource is released.

5 Implementation

We implemented NrOS from scratch in Rust; it currently targets the x86-64 platform. It also has basic support for Unix as a target platform, which allows kernel subsystems to be run within a Linux process and helps support unit testing. The core kernel consists of 11k lines of code with 16k additional lines for kernel libraries (bootloader, drivers, memory management, and platform specific code factored out from the core kernel into libraries). In the entire kernel codebase, 3.6% of lines are within Rust `unsafe` blocks (special blocks that forego the compiler's strong memory- and thread-safety guarantees). Most of this unsafe code casts and manipulates raw memory (*e.g.,* in memory allocators or device drivers), a certain amount of which is unavoidable in kernel code.

Node Replication. We implemented node replication in Rust as a portable library totaling 3.4k lines of code (5% in `unsafe` blocks). We made some notable changes to the state-of-the-art node replication design [28] and built CNR on top of it. Specifically, our implementation relies on Rust's generic types, making it easy to lift arbitrary, sequentially-safe Rust structures into node-replicated, concurrent structures. This is done by implementing the `Dispatch` interface in Listing 1.

Name	Memory	Nodes/Cores/Threads
2×14 Skylake	192 GiB	2x14x2 Xeon Gold 5120
4×24 Cascade	1470 GiB	4x24x2 Xeon Gold 6252

Table 1: Architectural details of our evaluation platforms.

Userspace Library. NrOS provides a userspace runtime support library (vibrio) for applications. It contains wrapper functions for kernel system calls, a dynamic memory manager, and a cooperative scheduler that supports green threads and standard synchronization primitives (condition variables, mutexes, readers-writer locks, semaphores, *etc.*).

This library also implements hypercall interfaces for linking against rumpkernels (a NetBSD-based library OS) [50]. This allows NrOS to run many POSIX programs. rumpkernel provides libc and libpthread which, in turn, use vibrio for scheduling and memory management through the hypercall interface. The hypercall interface closely matches the reference implementation of the rumprun-unikernel project [9]; however, some significant changes were necessary to make the implementation multi-core aware. The multi-core aware implementation was inspired by LibrettOS [62].

The NrOS kernel itself does not do any I/O, but it abstracts interrupt management (using I/O APIC, xAPIC and x2APIC drivers) and offers MMIO PCI hardware passthrough to applications. Applications can rely on the rump/NetBSD network or storage stack and its device drivers for networking and disk access (supporting various NIC models and AHCI based disks). The I/O architecture is similar to recent proposals for building high performance userspace I/O stacks [18, 66].

6 Evaluation

This section answers the following questions experimentally:

- How does NrOS's design compare against monolithic and multikernel operating systems?
- What is the latency, memory and replication mechanism trade-off in NrOS' design compared to others?
- Does NrOS's design matter for applications?

We perform our experiments on the two machines given in Table 1. For the Linux results, we used Ubuntu version 19.10 with kernel version 5.3.0. If not otherwise indicated, we did not observe significantly different results between the two machines and omit the graphs for space reasons. We pinned benchmark threads to physical cores and disabled hyperthreads. Turbo boost was enabled for the experiments. If not otherwise indicated we show bare-metal results.

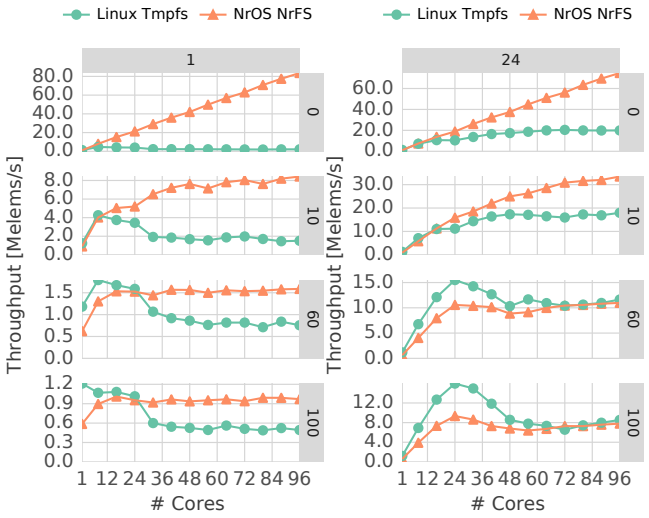


Figure 3: NR-FS read/write scalability for write ratios 0%, 10%, 60% and 100% with 1 or 24 files on 4×24 Cascade.

6.1 Baseline Node Replication Performance

We have extensively tested our Rust-based NR library in userspace on Linux using a variety of structures to compare its performance with the existing state-of-the-art NR implementation [28] and with other optimized concurrent structures. We omit these results as they are orthogonal to NrOS's contributions, but we summarize a few key results when comparing with RCU which is the most relevant comparison for NrOS.

We tested scaling a hash table on 4×24 Cascade, NR (just wrapping the unmodified, sequential `HashMap` from the Rust standard library as shown in Listing 1) outperforms other concurrent hash maps written in Rust and the *urcu* library's read-modify-write-based hash table. With 0% updates, urcu and node replication scale linearly, but urcu lags behind; NR achieves perfect NUMA locality by replicating the hash table. The NR hash table also stores elements in-place, whereas urcu stores a pointer to each element, leading to an additional de-reference. This is inherent in the urcu design since elements are deallocated by the last reader. In short, for read-only workloads, NR performs about twice as well as urcu, which is the next fastest approach we tested. Even with any fraction of read operations it performs strictly better at scale. However, we find that urcu can outperform NR when reads and writes are split between threads rather than when reads and writes are mixed on every thread. This is because RCU allows readers to proceed without any synchronization overhead whereas node replication must acquire the local reader lock.

6.2 NR-FS

We evaluate NR-FS performance by studying the impact of issuing read and write operations to files using a low-level microbenchmark and a LevelDB application benchmark.

6.2.1 Microbenchmark: NR-FS vs `tmpfs`

In this benchmark, we are interested in the file operation throughput while varying cores, files and the read/write ratio. We pre-allocate files of 256 MiB upfront. Each core then accesses exactly one file, and each file is shared by an equal number of cores when the number of cores exceeds the number of files. The cores either read or write a 1 KiB block in the file uniformly at random. This general access pattern is typical for many databases or storage systems [43, 46]. We compare against the Linux's (in-memory) `tmpfs` to minimize persistence overhead [23].

Figure 3 shows the achieved throughput for write ratios 0%, 10%, 60%, and 100%, while increasing the number of cores (x-axis). The left graphs measured the throughout if a single file is read from/written to concurrently. With $WR = 0\%$, NR-FS achieves ∼40× better read performance at max. utilization. This increase is due to replication of the file system and making reads an immutable operation; largely the benefit comes from higher available memory bandwidth (4×24 Cascade has 88 GiB/s local vs. 16 GiB/s remote read bandwidth). However, replication increases the memory consumption significantly; for 24 files, each 256 MiB, `tmpfs` uses 6.1 GiB (6 GiB data and 0.1 GiB metadata) as compared to 24.1 GiB (24 GiB data and 0.1 GiB metadata) for NR-FS. For higher write ratios, `tmpfs` starts higher as NR-FS performs an additional copy from user to kernel memory to ensure replica consistency (Section 4.3) and its write is likely not as optimized as the Linux codebase. However, the `tmpfs` throughput drops sharply at the first NUMA node boundary due to contention and increased cache coherence traffic. For $WR = 100\%$, NR-FS performs ∼2× better than `tmpfs` at max. utilization.

Discussion: With the Intel architectures used in our setting, single file access generally outperforms Linux as soon as the file size exceeds the combined L3 size of all NUMA nodes (128 MiB on 4×24 Cascade). A remote L3 access on the same board is usually faster than a remote DRAM access; therefore, replication plays a smaller role in this case. As long as the file fits in L3 or the combined L3 capacity, NR-FS has on-par or slightly worse performance than `tmpfs`. NR-FS gains its advantage by trading memory for better performance.

The right side of the figure shows the less contended case where the cores are assigned to 24 files in a round-robin fashion (at 96 cores, each file is shared by four cores). For $WR = 0\%$, NR-FS performs around 4× better than `tmpfs` due to node local accesses from the local replica. For higher write ratios (60%, 100%), `tmpfs` performs better than NR-FS on the first NUMA node. On top of the additional copy, the major reason for the overhead here is that intermediate buffers for writes in NR-FS remain in the log until all replicas have applied the operation. This results in a larger working set and cache footprint for writes than `tmpfs`, which can reuse the same buffers after every operation. We empirically verified that this is the case by making the block size smaller; with

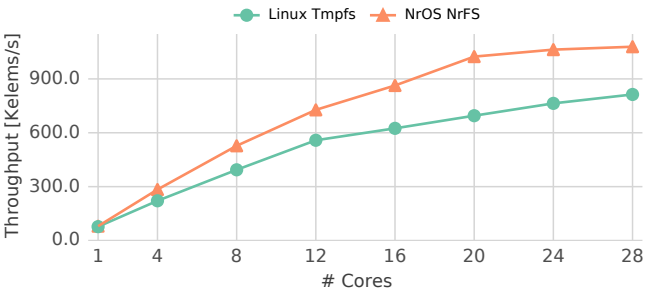


Figure 4: LevelDB readrandom throughput on 2×14 Skylake.

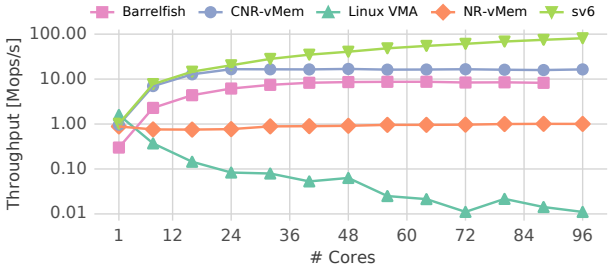


Figure 5: NrOS page insertion throughput on 4×24 Cascade in comparison with other OSes.

this change the performance discrepancy between `tmpfs` and NR-FS disappears.

After the first NUMA node, `tmpfs` throughput degrades due to contention, and cross-node memory accesses. NR-FS manages to keep the same performance as a single NUMA node. At *cores* = 96 both systems have similar throughput, but NR-FS actively replicates all writes on all 4 nodes. We omit the results using 2 to 23 files because the trend is the same: NR-FS performs much better for read-heavy workloads and the same or better for write-heavy operations. On 4×24 Cascade with 24 logs and $WR = 100\%$, NR-FS scalability stops for this benchmark with more than 24 files because of the additional CPU cycles required for active replication of all writes, the observed throughput remains constant instead.

Impact of multiple logs: Figure 3 shows the advantages of using CNR over node replication for less contended, write-intensive workloads. As discussed in (§2.1), the write performance for node replication data structures is often limited by a single combiner per replica. While it is certainly possible to build compound structures using multiple node replication instances (*e.g.,* one per file), this is typically too fine-grained, as often we have much less compute cores than files. We resolve this issue with our CNR (§3) scheme. A CNR based NR-FS performs 8× better (for $wr = 100$) than a node replication based NR-FS while preserving read performance.

6.2.2 LevelDB Application Benchmark

To evaluate the impact of the NR-FS design on real applications we use LevelDB, a widely used key-value store. LevelDB relies on the file system to store and retrieve its data, which makes it an ideal candidate for evaluating NR-FS. We use a key size of 16 Bytes and a value size of 64 KiB. We load 50K key-value pairs for a total database size of 3 GiB, which LevelDB stores in 100 files.

NR-FS outperforms `tmpfs` when running LevelDB. Figure 4 shows LevelDB throughput when running its included `readrandom` benchmark while varying core count. After *cores* = 12, contention on a mutex within LevelDB begins to affect the scalability of both systems. At *cores* = 28, LevelDB on NrOS has 1.33x higher throughput than on Linux.

6.3 NR-vMem

We evaluate the performance of NR-vMem with microbenchmarks that stress the address-space data structures under contention, and exercise the respective TLB shootdown protocols on different operating systems. Finally, we measure the impact of page table replication with memcached.

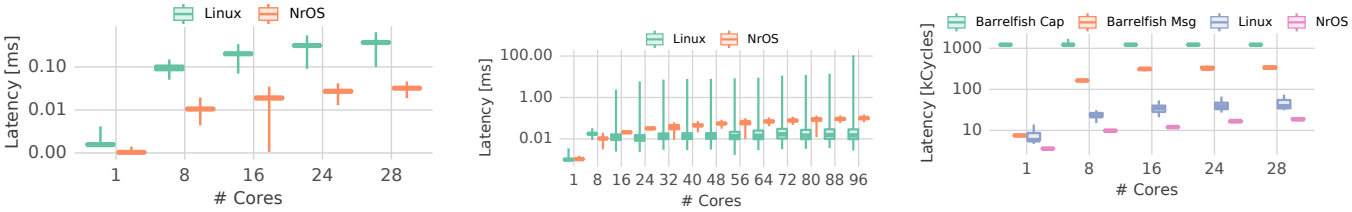
6.3.1 Map Performance

For this benchmark we compare NrOS against Linux, sv6, and Barrelfish. We allocate a backing memory object (*e.g.,* a physical memory region on NrOS, a shared memory object on Linux, a physical frame referenced by a capability on Barrelfish, and a memory-backed file on sv6) and repeatedly map the same memory object into the virtual address space of the process. The benchmark focuses on synchronization overheads; it creates the mapping and updates bookkeeping information without allocating new backing memory.

We evaluate a partitioned scenario where each thread creates new mappings in its own address space region (the only comparison supported by all OSes). We ensure that page tables are created with the mapping request by supplying the appropriate flags. sv6 does not support `MAP_POPULATE`, so we force a page fault to construct the mapping. We show throughput of the benchmark in Figure 5.

NR-vMem wraps the entire address space behind a single instance of node replication, therefore it does not scale even for disjoint regions. As this benchmark consists of 100% mutating operations, it has constant throughput – similar to the other benchmarks it remains stable under heavy contention.

Linux is very similar to NR-vMem in its design (apart from missing replication). It uses a red-black tree to keep track of mappings globally. For each iteration of the benchmark, a new mapping has to be inserted into the tree and afterwards the page fault handler is called by `mmap` to force the population of the page table. The entire tree is protected by a global lock and therefore performance decreases sharply under contention. The single-threaded performance of Linux VMA is slightly better than NR-vMem which has still room for improvement:



(a) Map latency for 2×14 Skylake.

(b) Map latency for 4×24 Cascade.

(c) Unmap latency on 2×14 Skylake.

Figure 6: Virtual memory operation (map and unmap) observed latency distributions.

For example, our current implementation zeroes page tables while holding the combiner lock.

Barrelfish (git revision `06a9f5`) tracks resources with a distributed, partitioned capability system. Updating a page table and allocating new memory corresponds to a capability operation, which may require agreement among the multikernel nodes. The Barrelfish memory management code shares a capability to the top level page table and statically partitions the virtual address space of a process (*e.g.,* a core has only capabilities to its own partition which only it can modify). So, inserts to private regions do not require any agreement. Furthermore, this design uses a single page table per process in the system; therefore, there is no overhead to synchronize replicated state.

For good scalability, we eliminated the use of $O(n)$ linked-list operations for the address-space meta-data tracking of the OS. Once we fixed those issues, Barrelfish throughput scaled with a sub-linear factor. Concurrent updates to the same partition from multiple cores are not supported. This could be implemented with delegation, by using the provided messaging infrastructure in the library OS and capability operations.

sv6 (git revision `9271b3f`) uses a radix tree data structure [29] for its mapping database. It is able to apply fine-grained locking for non-overlapping regions of the address space by atomically setting bits at each level in the radix tree. Compared to all evaluated systems, sv6 performs best for disjoint regions with near-linear scalability. A potential downside is the memory overhead, since sv6 replicates page tables on every core. It mitigates this issue with lazy construction of page tables and discarding them under memory pressure.

CNR-vMem. While NR-vMem does not scale as well as sv6 or Barrelfish for mappings in disjoint regions, it keeps a relatively complex interaction of multiple data structures as entirely single-threaded code. If better scalability is desired, we can scale NR-vMem updates by using CNR. Similar to Barrelfish, CNR-vMem partitions the address space into 512 separate regions and maps updates to partitions with different logs. CNR-vMem matches sv6' performance on the first NUMA node. Afterwards, scaling stops because of per-NUMA replication. Compared to Barrelfish, we find that CNR-vMem is more flexible: concurrent updates from multiple cores to the same partition are supported without extra implementation effort thanks to flat-combining.

Latency. To understand how the batching in node replication impacts latency, we further instrument Linux and NR-vMem by measuring the completion time for 100k requests per core. Figure 6a and 6b show the latency distributions (with the min representing p1, and max p99) for 4×24 Cascade and 2×14 Skylake. We observe slightly worse median latencies for NrOS on 4×24 Cascade but overall better tail characteristics and throughput. On the other hand, we find that NrOS has better latencies than Linux on 2×14 Skylake. Latency is directly correlated with the number of cores participating in flat combining (*i.e.,* 2×14 Skylake has only 14 cores per replica vs. 24 on 4×24 Cascade). Per-NUMA replicas offer a good compromise for memory consumption vs. throughput, but we can tune latency further by having more than one replica per node on NUMA nodes with more cores.

6.3.2 Unmap and TLB Shootdown Performance

We evaluate the scalability of unmapping a 4 KiB page by measuring the time it takes to execute the unmap system call, which includes a TLB shootdown on all cores running the benchmark program.

We compare our design to Linux and Barrelfish. Linux uses a single page table which is shared among the cores used in this benchmark. The general shootdown protocol is similar to the one in NrOS, except that NrOS has to update multiple page tables in case a process spawns multiple NUMA nodes. Barrelfish partitions control over its page table per core and uses the capability system (Barrelfish Cap) or userspace message passing (Barrelfish Msg) to ensure consistency among the replicas and coherency with the TLB. Barrelfish uses point-to-point message channels instead of IPIs.

Figure 6c shows the latency results. NrOS outperforms all other systems. Barrelfish Cap has a constant latency when using a distributed capability operation because all cores participate in the 2PC protocol to revoke access to the unmapped memory regardless of whether this memory was actually mapped on that core. Moreover, implementing the TLB shootdown using point-to-point messages (Barrelfish Msg) has a higher constant overhead compared to using x2APIC

OS	Time	Throughput	System Mem.	PT Mem.	PT Walks
NrOS 4-replicas	251 s	63 Mop/s	424 GiB	3.3 GiB	1.20 kcyc/op
NrOS 1-replica	276 s	57 Mop/s	421 GiB	840 MiB	1.54 kcyc/op
Linux	327 s	48 Mop/s	419 GiB	821 MiB	1.63 kcyc/op

Table 2: memcached on NrOS (1 and 4 replicas) and Linux running on 4×24 Cascade, comparing runtime, throughput, total system memory consumption, process page table memory and cycles spent by the page table walkers.

with broadcasting in NrOS and Linux due to sequential sending and receiving of point-to-point messages. Using more optimized message topologies could potentially help [48].

Linux should achieve better results than NrOS, especially when we spawn across NUMA since it only has to update one page table. However, the proposed changes to Linux from the literature [12] which inspired our TLB shootdown protocol have not yet been integrated to upstream Linux. We expect Linux to be comparable once early acknowledgments and concurrent shootdown optimizations become available.

6.3.3 Page Table Replication with Memcached

As a final benchmark, we measure the impact of replicated page tables on memcached. When taking into account the implicit reads of the MMU, page tables often end up being read much more than modified. memcached serves as a representative application for workloads with generally high TLB miss ratios (*i.e.,* applications with large, in-memory working sets and random access patterns).

We measure the throughput of memcached with `GET` requests (8 byte keys, 128 byte values, 1B elements) on 4×24 Cascade. Our benchmark directly spawns 64 client threads inside of the application. For this experiment, we run Linux and NrOS inside KVM because we want to have access to the performance counters, which is currently not supported on NrOS. To limit the effects of nested-paging, we configure KVM to use 2 MiB-pages, and use 4 KiB pages in both the Linux and NrOS guests.

Table 2 compares memcached running on NrOS in different configurations and Linux. Overall, the achieved throughput for NrOS (with per-NUMA replication) is 1.3× higher than Linux. To quantify the impact of page table replication on the throughput, we can configure NrOS to use a single replica for the process (NrOS 1-replica). We find that the page table replication accounts for a third of the overall improvement compared to Linux. The systems have different physical memory allocation policies, locking implementation, scheduling, and page tables *etc.*, so it is difficult to attribute the other two thirds to specific causes.

By instrumenting performance counters, we find that remote page table walks – a key bottleneck for this workload – decreased by 23% with replication. NrOS does use 4× more memory for the replicated page tables structures. In total, this still amounts to less than 1% of the total memory.

7 Conclusion and Future work

We designed and implemented NrOS, an OS that uses single-threaded data structures that are automatically adapted for concurrency via operation logging, replication, and flat combining. Our results show that the NRkernel model can achieve performance that is competitive with or better than well-established OSes in many cases.

NrOS' unique design makes it an interesting platform to explore several future directions:

Relaxing consistency. We apply node replication on relatively coarse-grained structures, which makes reasoning about concurrency easy. CNR improves performance by exploiting commutativity among mutating operations. However, we could achieve better performance by relaxing strong consistency between replicas for some operations.

Verifying correctness. NrOS might also serve as a useful basis for a verified multi-core operating system by using verification in two steps: verify the node replication transformation from a sequential data structure to a concurrent one, then verify the sequential data structures. Verifying node replication is harder, but it only needs to be done once. Verifying new sequential data structures is substantially easier.

Extending NrOS for disaggregated compute. NrOS' log-based approach with replication is most useful when systems have high remote access latencies. Thus, NrOS could be extended to work over interconnects that offer shared memory in compute clusters via Infiniband or other high-speed networks by designing a new log optimized for the interconnect.

Acknowledgments

We thank our OSDI 2020 and 2021 reviewers and our shepherd Irene Zhang for their thoughtful feedback. Ankit Bhardwaj and Chinmay Kulkarni contributed to this work as PhD students at University of Utah and during internships at VMware Research. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Chinmay Kulkarni is supported by a Google PhD Fellowship.

References

[1] CCIX. https://www.ccixconsortium.com.

[2] Concurrency bugs should fear the big bad data-race detector. https://lwn.net/Articles/816850/.

[3] Fix a data race in ext4_i(inode)->i_disksize. https://lore.kernel.org/patchwork/patch/1190562/.

[4] Fix a data race in mempool_free(). https://lore.kernel.org/patchwork/patch/1192684/.

[5] Fix locking in bdev_del_partition. https://patchwork.kernel.org/project/linux-block/patch/20200901095941.2626957-1-hch@lst.de/.

[6] Fix two RCU related problems. https://lore.kernel.org/patchwork/patch/990695/.

[7] Gen-Z Consortium. https://genzconsortium.org/.

[8] The RCU API, 2019 Edition. https://lwn.net/Articles/777036/.

[9] The Rumprun Unikernel. https://github.com/rumpkernel/rumprun.

[10] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 283–300, 2020.

[11] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[12] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *European Conference on Computer Systems (EuroSys)*, 2020.

[13] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 95–109, 1991.

[14] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 340–349, 1990.

[15] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual consensus in Delos. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 617–632, 2020.

[16] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of Ottawa Linux Symposium (OLS)*, 2014.

[17] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[18] 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 *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–65, 2014.

[19] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–86, 2017.

[20] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 19–31, 1989.

[21] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, page 6, 1994.

[22] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 43–57, 2008.

[23] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 1–16, 2010.

[24] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical report, 2014.

[25] 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, November 1997.

[26] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[27] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 83–97, 2013.

[28] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, 2017.

[29] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *European Conference on Computer Systems (EuroSys)*, page 211–224, 2013.

[30] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4), January 2015.

[31] Jonathan Corbet. The big kernel lock strikes again, 2008. https://lwn.net/Articles/281938/.

[32] Jonathan Corbet. Big reader locks, 2010. https://lwn.net/Articles/378911/.

[33] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, 2013.

[34] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *International Conference on Automated Software Engineering (ASE)*, pages 166–177, 2015.

[35] Medhavi Dhawan, Gurprit Johal, Jim Stabile, Vjekoslav Brajkovic, James Chang, Kapil Goyal, Kevin James, Zeeshan Lokhandwala, Anny Martinez Manzanilla, Roger Michoud, Maithem Munshed, Srinivas Neginhal, Konstantin Spirov, Michael Wei, Scott Fritchie, Chris Rossbach, Ittai Abraham, and Dahlia Malkhi. Consistent clustered applications with Corfu. *Operating Systems Review*, 51(1):78–82, 2017.

[36] Hugh Dickins. [PATCH] mm lock ordering summary, 2004. http://lkml.iu.edu/hypermail/linux/kernel/0406.3/0564.html.

[37] Marco Elver. Add Kernel Concurrency Sanitizer (KCSAN). https://lwn.net/Articles/802402/, 2019.

[38] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[39] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 151–162, 2010.

[40] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[41] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, 1999.

[43] Sanjay Ghemawat and Jeff Dean. LevelDB, 2011.

[44] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[45] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 207–216, 2008.

[46] Richard D Hipp. SQLite, 2020.

[47] Dae R. Jeong, Kyungtae Kim, Basavesh Ammanaghatta Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[48] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–48, 2016.

[49] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *USENIX Annual Technical Conference (ATC)*, pages 263–276, 2015.

[50] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, 2012.

[51] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 586–599, 2019.

[52] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling Read-Log-Update with multi-versioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 779–792, 2019.

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[54] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *European Conference on Computer Systems (EuroSys)*, pages 133–145, 2006.

[55] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-Based Analysis of Locking in the Linux Kernel. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 11:1–11:15, 2019.

[56] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[57] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, page 399–413, 2019.

[58] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-Log-Update: A lightweight synchronization mechanism for concurrent programming. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2015.

[59] Paul E McKenney. A critical RCU safety property is... ease of use! In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 132–143, 2019.

[60] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding many-core scalability of file systems. In *USENIX Annual Technical Conference (ATC)*, 2016.

[61] Ingo Molnar and Davidlohr Bueso. Generic Mutex Subsystem, 2017. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[62] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A dynamically adaptable multiserver-library OS. In *International Conference on Virtual Execution Environments (VEE)*, page 114–128, 2020.

[63] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–320, 2014.

[64] OpenCAPI consortium. http://opencapi.org.

[65] Heidi Pan, Benjamin Hindman, and Krste Asanovic. Composing parallel software efficiently with Lithe. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[66] 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. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2014.

[67] 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)*, 2015.

[68] Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. Analytics with smart arrays: Adaptive and efficient language-independent data. In *European Conference on Computer Systems (EuroSys)*, 2018.

[69] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *European Conference on Computer Systems (EuroSys)*, page 305–315, 2006.

[70] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87, 2018.

[71] Navin Shenoy. A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data.

[72] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–289, 1996.

[73] Al Viro. parallel lookups, 2016. https://lwn.net/Articles/684089/.

[74] Michael von Tessin. The Clustered Multikernel: An approach to formal verification of multiprocessor os kernels. In *Workshop on Systems for Future Multicore Architectures (SFMA)*, 2012.

[75] Dmitry Vyukov. Distributed reader-writer mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2011.

[76] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. Evaluation of Intel 3D-Xpoint NVDIMM technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, page 277–287, 2019.

[77] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A black-box approach to NUMA-aware persistent memory indexes. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[78] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2):76–85, April 2009.

[79] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 163–176, 2010.

[80] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

A Artifact Appendix

Abstract

The evaluated artifact is provided as a git repository and contains the source code of NrOS, build instructions and scripts to run the OS and benchmarks used in this paper.

Scope

The artifact contains code and steps to reproduce results obtained in Figure 3, Figure 4, Figure 5 and Figure 6.

Contents

The artifact consists of NrOS, including libraries, userspace programs and benchmarks. The documentation to build and run NrOS, along with the necessary commands to run the benchmarks are written down in the `doc` folder of the repository. The document which lists the steps to execute the artifact evaluation is located at `doc/src/benchmarking/ArtifactEvaluation.md`.

Hosting

The artifact source code for NrOS is published on Github under https://github.com/vmware-labs/node-replicated-kernel.
The code used in the artifact evaluation is tagged as `osdi21-ae-v2`.

Requirements

Building NrOS requires an x86-64 system set-up with Ubuntu 20.04 LTS.
NrOS itself requires an Intel CPU (Skylake microarchitecture or later) to run. The following CPUs are known to work: Xeon Gold 5120, 6252 or 6142. For virtualized execution on these platforms, a Linux host system with QEMU (version >= 5.0.0) and KVM is required. For bare-metal execution, DELL PowerEdge R640 and R840 servers systems are known to work.

Addra: Metadata-private voice communication over fully untrusted infrastructure

Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta

University of California, Santa Barbara

Abstract

Metadata from voice calls, such as the knowledge of who is communicating with whom, contains rich information about people's lives. Indeed, it is a prime target for powerful adversaries such as nation states. Existing systems that hide voice call metadata either require trusted intermediaries in the network or scale to only tens of users. This paper describes the design, implementation, and evaluation of Addra, the first system for voice communication that hides metadata over fully untrusted infrastructure and scales to tens of thousands of users. At a high level, Addra follows a template in which callers and callees deposit and retrieve messages from private mailboxes hosted at an untrusted server. However, Addra improves message latency in this architecture, which is a key performance metric for voice calls. First, it enables a caller to push a message to a callee in two hops, using a new way of assigning mailboxes to users that resembles how a post office assigns PO boxes to its customers. Second, it innovates on the underlying cryptographic machinery and constructs a new private information retrieval scheme, FastPIR, that reduces the time to process oblivious access requests for mailboxes. An evaluation of Addra on a cluster of 80 machines on AWS demonstrates that it can serve 32K users with a 99-th percentile message latency of 726 ms—a 7× improvement over a prior system for text messaging in the same threat model.

1 Introduction

Voice call metadata—the parties involved in the call, the duration of the call, and the time of the call—can be incredibly revealing. The former General Counsel of NSA, Stewart Baker, has said, "metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content" [21, 22, 66]. Several academic studies [27, 54, 55] have confirmed the power of metadata. As an example, Mayer et al. [54] used telephone metadata to infer that a study participant "received a long phone call from the cardiology group at a regional medical center, talked briefly with a medical laboratory, ... and made brief calls to a self-reporting hotline for a cardiac arrhythmia monitoring device." The authors confirmed that the participant had a cardiac arrhythmia. A study of whistle-blowers also revealed that metadata can identify a journalist's sources [39].

Given the information contained in metadata, a significant question is: how can one make a voice call without revealing to anyone the metadata associated with the call? Fortunately, several systems have tackled this problem [13, 34, 49, 51, 70, 72] (§7). Although these systems hide metadata and keep message latency low, they either restrict scalability to only tens of users [34, 70], or are vulnerable to attacks by requiring trusted intermediaries in the communication infrastructure [13, 49, 51, 72]. An example of a trust assumption is that the system guarantees security only if the adversary can compromise at most a fraction (20%) of the servers that route user calls [49]. Trusting intermediaries can be risky as powerful adversaries like nation states are the ones that try to collect metadata. Such adversaries have been known to wield their vast political, technical, and financial power to gain access to metadata [12, 53, 59, 67].

A system that can withstand strong adversaries while serving more than tens of users is Pung [7, 10]. Pung makes *no* assumptions about the communication infrastructure—the adversary may compromise a part or all of the infrastructure. However, Pung targets applications such as email and chat with long-lived messages that are retrieved asynchronously. Indeed, a Pung client makes $\lceil \log_2(n+1) \rceil$ round trips to a remote server to obliviously search and retrieve a message (n is the number of users), thereby incurring several seconds of message latency (§6.1). In contrast, voice calls have a strict time budget. If a user sends a packet every few hundred milliseconds, then each hop in the communication infrastructure must not spend longer than this time period to process and forward the packet, to avoid an unbounded packet build up.

We present Addra, the first system that provably hides metadata for voice calls, makes no assumptions about the underlying infrastructure, and scales to tens of thousands of users. In terms of privacy guarantees, Addra provides relationship unobservability—an adversary cannot detect whether a relationship (voice call) exists between any two users of the system [63] (§2.1). These privacy guarantees are achieved with practical latency performance of under 750 ms, and for low-bandwidth voice synthesis at a rate of 1.6 Kbit/s as in the Mozilla LPCNet voice codec [57, 74, 75].

Addra, like Pung, relies on a set of mailboxes hosted at an untrusted server. Callers deposit messages and callees retrieve messages from these mailboxes using a private information retrieval (PIR) cryptographic protocol [19, 20, 43] (§3.2). This protocol ensures that the untrusted server does not learn which mailbox a callee is accessing, thereby unlinking the callee from the caller. However, Addra must address two challenges in this architecture to support low-latency voice calls (§2.3). First, it must reduce the number of round trips a caller or callee makes to the server to transfer or retrieve a voice packet. Second, Addra must reduce the time the server takes to process caller and callee requests, particularly, the PIR requests.

Addra addresses the first challenge through a new, and remarkably simple, use of mailboxes (§3). When someone rents a conventional post office box, or PO box, at a post office, they get a mailbox with a unique and fixed address into which the mailman deposits incoming mail. Addra inverts this architecture. In Addra, a caller (rather than a recipient or callee) gets a dedicated mailbox with a fixed address or "phone number". The caller deposits its outgoing messages into this mailbox—independent of who the caller is calling. (Thus, an adversary cannot tell whom the caller is calling.) Meanwhile, a callee retrieves a message from the mailbox tied to the caller's phone number using a PIR protocol. Crucially, to transmit a message, a caller makes one push request to the server, and the server makes one push request to the callee—a hop count of two. In contrast, prior work requires multiple round trips between the server and the callees.

Addra addresses the second challenge mentioned above, of reducing server-side processing time for PIR, by two means. First, it parallelizes PIR processing across multiple server machines and multiple CPU cores on a machine. The fact that PIR is parallelizable is known and studied [29, 37]. Second, and more saliently, Addra constructs a new PIR scheme, FastPIR, that fundamentally reduces the server-side PIR processing time relative to prior state-of-the-art schemes [4, 7] (§4). Even though FastPIR was motivated by Addra, it can be used for other applications of PIR [14, 30, 36, 56].

FastPIR builds on the homomorphic encryption scheme of Brakerski/Fan-Vercauteren (BFV) [15, 33] (§4.1) and leverages two of its features. First, it uses the single instruction, multiple data (SIMD) capability of BFV ciphertexts to compute on compressed PIR requests. Prior state-of-the-art schemes [4, 7] also exploit SIMD capabilities but not in a way that keeps PIR requests compressed in memory. Meanwhile, such compression improves memory utilization, reduces CPU time, and eliminates the time to uncompress requests (§4.2). However, working over compressed requests naively increases PIR response size. So, second, FastPIR uses homomorphic rotation operations in BFV to pack multiple pieces of a PIR response, thereby reducing response size. Further, FastPIR reduces both the CPU time per rotation and the number of calls to this operation (§4.3, §4.4).

For completeness, Addra includes a dialing protocol that allows a callee to detect that a caller is calling and learn the caller's phone number (mailbox address). For this purpose, Addra uses the dialing protocol from Pung (§5).

We have implemented (§5) and evaluated (§6) a prototype of Addra. Our prototype runs on Amazon EC2 where the server runs in the US East region, and the clients (callers and callees) run geographically apart in the US West region. When the server uses 80 machines, Addra supports 32K clients communicating with each other with a 99-th percentile message latency of 726 ms. In contrast, Pung (the only other system that works at scale over completely untrusted infrastructure) transmits messages for the same number of users with a message latency of 5.2 seconds. Besides, Addra requires a network download bandwidth of 1.46 Mbps and an upload bandwidth of 30 Kbps for every client.

Although Addra achieves low message latency for a few tens of thousands of users, it does not currently scale to hundreds of thousands or a few million users due to the overhead of PIR which grows quadratically with the number of users. Furthermore, although its instantaneous bandwidth requirements are modest, the total network transfers are high as a client must remain online even if it is not participating in a call to hide call initiation patterns. Thus, Addra assumes clients with unlimited data plans. Nevertheless, Addra demonstrates, for the first time, that even over completely untrusted infrastructure, metadata for voice calls can be hidden at scale for tens of thousands of users.

2 Goals, threat model, and challenges

Addra's goal, at a high level, is to enable its users to make peer-to-peer voice calls while hiding metadata from a powerful adversary that may compromise the entire communication infrastructure.

2.1 Goals

Performance and scalability. Voice calls require the communication infrastructure to transmit messages with low latency. Addra targets a sub-second message latency due to the feasibility of voice calls under such a setting [49]. Thus, if Alice sends a voice packet to Bob, then Bob should receive it within one second. Additionally, the infrastructure must not queue up voice packets indefinitely. For instance, if Alice generates a voice packet every 500 ms, then every hop in the infrastructure must spend no more than 500 ms to process the packet before sending it forward towards Bob. Addra must also provide sufficient throughput so that the transmitted voice is understandable. For this purpose, Addra targets the LPCNet voice codec [74, 75], which specializes in low-bandwidth voice synthesis at a rate of 1.6 Kbit/s. Finally, we want Addra to scale to a large number of users (for example, tens of thousands on a cluster of hundred machines).

Content privacy. Addra must ensure that only the caller and callee of a voice call can comprehend the content of the voice packets they send to each other.

Metadata privacy. Addra, similar to Pung [10], targets the guarantee of *relationship unobservability* as defined by Pfitzmann and Hansen [63]. Relationship unobservability states that it is undetectable whether a relationship (voice call) exists between a sender (caller) and a recipient (callee), unless the sender or the recipient are compromised. If either the caller or the callee is compromised, then offering privacy guarantees has little value, as the compromised party can trivially reveal the existence of communication (or lack thereof).

2.2 Threat model and assumptions

As motivated in the introduction (§1), Addra assumes an adversary who can compromise the entire communication infrastructure, including routers, switches, and middleboxes. The adversary can observe network traffic, perform traffic analysis, and manipulate traffic: reorder, replay, change, and inject network packets.

Callers and callees trust their own devices. More generally, the adversary can compromise a subset of end user devices. In this case, Addra must provide content and metadata privacy to the users of non-compromised devices.

The adversary may not break standard cryptographic primitives such as public-key and symmetric-key encryption.

The adversary may mount a denial-of-service attack: bring down the entire communication infrastructure or selectively drop traffic. In such cases, Addra cannot guarantee voice communication but must continue to guarantee privacy.

2.3 Challenges

Meeting the performance and privacy goals stated above under the threat model just described is challenging. Indeed, prior work either relaxes the threat model or does not meet the performance goals. For instance, Yodel [49] is a metadata-private voice communication system that scales to several million users but assumes that a server in the communication infrastructure is compromised with only a 20% chance. On the other hand, Pung [7, 10] works in the stronger threat model. However, it cannot push frequent messages from a caller to a callee. As mentioned earlier (§2.1), if a caller samples voice every 500 ms, then each hop of the communication infrastructure must process a voice packet within 500 ms before the arrival of the next packet to avoid packet build up. This time budget entails that a caller or a callee cannot make multiple round trips to a server in the communication infrastructure to send or receive a single packet. But Pung requires message recipients to make multiple such trips to its server. Addra addresses these challenges and meets the performance requirements for tens of thousands of users without making any trust assumptions, as described next.

3 Architecture and overview of design

3.1 Architecture

Figure 1 shows Addra's architecture. Addra consists of a server and user (participant) devices. The server runs over untrusted infrastructure. It is logically centralized but physically distributed over multiple machines. The server's role is to facilitate communication among the user devices in a privacy-preserving manner.

The server exposes *mailboxes*. Specifically, it exposes n mailboxes, where n is the number of user devices using the system. Each mailbox can store one message and it has an ID, which is a number between (and inclusive of) 0 and $n-1$. As we will describe later (§4), it is helpful to view the n mailboxes as a matrix with n rows and m columns, where each row is an individual mailbox, and the m pieces of a message are m elements of a matrix row.

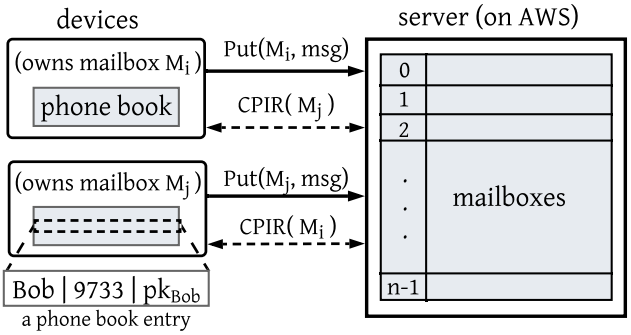


Figure 1—High-level architecture of Addra. The server runs over untrusted infrastructure and exposes mailboxes that user devices read from or write into. The mailbox identifiers (integers 0 to $n-1$) play the role of "phone numbers". A device stores phone numbers of the device owner's contacts in a local phone book. CPIR refers to the private information retrieval cryptographic primitive (§3.2).

The user devices run logic to enable users to initiate, pick up, and participate in calls. Each device gets assigned a mailbox ID, which acts as its phone number. Each device also contains a *phone book*, which stores information on device owner's contacts. Each phone book entry is a tuple of a phone number of the contact, a cryptographic public key belonging to the contact, and other standard information such as the contact's name, work place, and photograph. We assume that a device owner either knows this information or can obtain it privately through out-of-band means such as in-person meetings or personal websites.

3.2 Protocol

Addra relies on a cryptographic protocol called *private information retrieval* or *PIR* [19, 43]. We begin with a short background on PIR; Section 4 describes a new PIR scheme.

A primer on PIR. A PIR protocol [19, 43] runs between a user device and the server in Addra, where the device is interested in retrieving the message in the *idx*-th mailbox at the server without revealing the value of *idx*.

A PIR protocol has three procedures: QUERY, ANSWER, and DECODE. QUERY is run by a device. It takes as input the index *idx* between 0 and $n-1$ and returns a query, q. Typically, q is an encryption of a suitable encoding of *idx*. ANSWER is run by the server; it takes as input the query q and the set of n mailboxes, and returns an encoding of the message in the *idx*-th mailbox (without learning the value of *idx*). Finally, DECODE is run by the device; it takes the output of ANSWER and returns the *idx*-th mailbox message.

Addra's protocol. User devices in Addra participate in a round-based protocol consisting of a one-time registration step followed by synchronous rounds, each consisting of a dialing phase followed by a communication phase consisting

```
1: function RECV(key key, resp resp)
2:     // resp is the output of ANSWER PIR procedure
3:     c ← DECODE(resp)          // DECODE is a PIR procedure
4:     msg ← AES.DEC(key, c)
5:     play msg to user
6: function SEND(mailbox M, token t, message msg, key key)
7:     c ← AES.ENC(key, msg)
8:     send (M, t, c) to server
9: function MAIN( )
10:    // Register device and obtain a mailbox ID and unique token
11:    (M_self, tkn, n) ← REGISTERDEVICE()
12:    while True do
13:        // Run dialing phase. k_enc is for encrypting content
14:        (M_peer, k_enc) ← DIAL/PICKUP()
15:        q ← QUERY(M_peer, n)      // QUERY is a PIR procedure
16:        send q to server
17:        // Asynchronously listen for server responses
18:        register callback RECV(k_enc, ...) for server responses
19:        // Run communication phase consisting of t subrounds
20:        for r = 0 to t − 1 do
21:            wait for message generation interval
22:            call SEND(M_self, tkn, msg_r, k_enc)
```

Figure 2—Pseudocode for a user device in Addra. n is the number of mailboxes at the server. QUERY, ANSWER, DECODE are procedures of a PIR scheme (§3.2, §4).

of multiple subrounds of communication. In more detail, initially a device performs a one-time *registration step* to register itself with the server and obtain its phone number (mailbox ID). This is followed by a sequence of rounds. Each round starts with a *dialing phase*, where the device initiates a call to another device or picks an incoming call. The dialing phase is followed by the *communication phase*, consisting of multiple subrounds, where each device sends exactly one message to the server and receives one message from the server. Notably, a device always writes a message to its assigned mailbox, while it receives a message from its peer's mailbox.

We now describe Addra's protocol in more detail. Figure 2 shows the pseudocode for a user device. A device starts executing the MAIN function (line 9 in Figure 2).

One-time registration step. When a user device joins Addra, it registers itself with the server and obtains three pieces of information: a mailbox ID, a unique authentication token *tkn*, and the number n of mailboxes (line 11 in Figure 2). As mentioned above, the mailbox ID acts as the phone number assigned to the device. Meanwhile, the authentication token is a 128-bit uniformly generated string shared between the server and the device that enables the server to verify that a device is writing messages to its assigned mailbox (and not to a mailbox assigned to another device). One may use digital signatures instead of authentication tokens, but Addra prefers the *symmetric* tokens due to their better efficiency. The number of mailboxes n may increase if new devices join the system; when this happens, the server broadcasts an updated value of n.

Addra's server is untrusted and may assign mailbox IDs or authentication tokens incorrectly; for instance, it may reassign a previously assigned mailbox ID. Besides, it may distribute different values of n to different devices. The privacy guarantees of Addra's protocol do not depend on the server assigning correct values for these items. However, a malicious server can deny service to system participants, which is not prevented by our threat model (§2.2). A service provider who runs the server will likely be incentivized to provide a continuous service to keep its customer base.

Dialing phase. Once registered, a user device, who we refer to using its phone number, M_{self}, executes the round-based protocol. At the beginning of each round, M_{self} initiates a call or picks up an incoming call (line 14 in Figure 2). If the device initiates a call, it selects the phone number of the peer device it is calling, M_{peer}, and an encryption key, k_{enc}, to hide the content of the messages it will send. On the other hand, if the device picks up an incoming call then it learns the phone number of the caller and its content encryption key. For now, we leave out the details of how a device picks up a call till later (§5). After initiating or picking up a call, M_{self} generates a PIR query $q \leftarrow \text{QUERY}(M_{peer}, n)$ for the peer's mailbox, and sends q to the server (lines 15 and 16 in Figure 2). The PIR query indicates, without revealing the value of M_{peer}, that M_{self} is interested in receiving messages deposited into M_{peer}'s mailbox. The device M_{self} then registers an asynchronous callback to process PIR responses from the server (line 18 in Figure 2). Meanwhile, the server stores the PIR queries from all devices and uses them across all subrounds of the round's communication phase.

Communication phase. In each subround of the communication phase, (1) a device deposits an encrypted message into its assigned mailbox at the server, (2) the server processes PIR queries from all devices and pushes the results to devices who registered these queries, and (3) each device decodes its PIR response from the server. In more detail, at the beginning of a subround, a device encrypts the message it wants to send to its peer with the key k_{enc} to create a ciphertext c. It sends the tuple (M_{self}, tkn, c) to the server (line 22 in Figure 2), where *tkn* is the device's assigned authentication token obtained during the registration step. The server uses the token to validate that the messages being written to mailboxes indeed come from devices that own the mailboxes. After performing these checks, the server runs the ANSWER PIR procedure for all PIR queries. That is, for a query q sent by a device during the dialing phase, the server runs $resp \leftarrow \text{ANSWER}(\text{mailboxes}, q)$ and pushes the PIR response *resp* to the device. Finally, on receiving a response, a device invokes the callback it registered during the dialing phase (line 1 in Figure 2). This callback decodes the PIR response using the DECODE PIR procedure, decrypts the underlying message sent by the device's peer M_{peer}, and delivers the message to the user.

Dummy participation and messages. The protocol described so far does not address the case when a device owner does not participate in a call. During such idle periods, like prior systems for strong metadata privacy (e.g., [10, 49]), a device adds *cover traffic* (also called chaff). In particular, if a device does not initiate or pick up a call in a round's dialing phase, it calls itself: inputs M_{self} into QUERY (line 15 in Figure 2). Besides, if a device does not have a message to send during a subround, it writes an encryption of a random message into its mailbox. Sending cover traffic is necessary, as otherwise an adversary can learn connections between users by monitoring if they join and leave at similar times.

Security analysis. Addra's protocol satisfies relationship unobservability, meaning that an adversary cannot detect the existence of relationships between system users (§2.1). We provide a rigorous proof in an extended version of this paper [5]. Briefly, Addra's protocol meets the property because the protocol a user device executes is independent of whom the user is communicating with or the behavior of the (malicious) server. First, a user device encrypts messages using a content encryption key known only to its peer. Further, it always writes outgoing messages at fixed intervals to its own mailbox—independent of whether the device is engaged in a call, or the identity of its peer, or the behavior of the server who may or may not deliver incoming messages to the device, or who may replay messages. Second, the security property of PIR ensures that an adversary cannot tell the IDs of the mailboxes from which devices are retrieving messages. Again, the server may process PIR queries incorrectly, or broadcast an incorrect value n for the number of mailboxes, but a user device always registers a PIR query for one of the n mailboxes, no matter the value of n. Thus, the adversary cannot detect whether a user Alice is communicating with Bob or Charlie or someone else, or even communicating at all (i.e., retrieving messages from its own mailbox).

Performance characteristics. Addra's protocol exhibits two key characteristics that set it on the path to meeting its performance goals (§2.1). First, the protocol pushes messages from senders to recipients in two hops—independent of the number of users in the system. Specifically, in each subround, a sender pushes a message to the server, who then processes the PIR query provided beforehand by the recipient, and pushes the PIR response to the recipient. This two-hop communication pattern is crucial for voice calls which require low latency. Second, the protocol amortizes the cost of generating and transferring a PIR query across subrounds of a round (our prototype runs a round every five minutes, and a subround every 480 ms; §6). Thus, the server does not have to deal with PIR query management (and certain preprocessing of query) during the time-sensitive subrounds. Nevertheless, the server must complete computing ANSWER for all PIR queries in a time smaller than the voice packet generation interval (that is, the duration of a subround) to avoid packet build up. Besides, the network transfers from the server to the devices are dictated by the size of the output of ANSWER. Thus, a low cost of the ANSWER PIR procedure is key for Addra's performance.

4 FastPIR: A new CPIR scheme

As described above (§3.2), a critical component of Addra's protocol is the ANSWER PIR procedure. It not only dictates Addra's message latency but also the resource consumption (both CPU and network) imposed by Addra.

PIR schemes are of two types: computational PIR (CPIR) [43] and information-theoretic PIR (IT-PIR) [19, 20]. CPIR schemes assume a single (untrusted) server and rely only on cryptographic assumptions; in contrast, IT-PIR schemes are more efficient but require two or more non-colluding servers. In Addra, we use a CPIR scheme as its trust assumptions are in line with Addra's goal of not trusting the communication infrastructure (§2.2).

One can plug in an existing CPIR scheme, either XPIR [4] or SealPIR [7], which are the state-of-the-art CPIR schemes, into Addra's protocol (§3.2). However, these schemes exhibit a tension between the CPU time to run ANSWER (and thus the wall-clock time for ANSWER) and the output size of ANSWER (and thus the network overhead).

Suppose a CPIR client wants to privately retrieve the idx-th message from a library L of n messages (mailboxes) held at a server. In prior work, a typical way to construct a CPIR query is to treat the library as a matrix with n rows and generate a ciphertext for every row of L.[1] The ciphertext for the idx-th row encrypts the value 1, and the ciphertexts for the other rows encrypt 0. However, this strategy creates large queries with a number of ciphertexts that is proportional to the value of n (e.g., XPIR's query size is ≈33 MiB for n=2^{15}, and ≈1 GiB for n=2^{20}; §6.5). When the server processes larger queries, it consumes more memory and CPU cycles to read them into CPU caches, which slows down query processing. (SealPIR compresses the query while transferring it on the network, but expands it to the larger query at the server).

A popular technique due to Stern [71] to address the query-size issue is called *recursion*. This technique is parameterized by a depth parameter d. A value of $d = 2$ or higher shrinks the query—it contains $d \cdot \sqrt[d]{n}$ ciphertexts instead of n—by rearranging the library as a d-dimensional hypercube. However, this rearrangement increases the CPIR ANSWER output size exponentially with d. Thus, if we plug in existing CPIR schemes (XPIR or SealPIR), then Addra would compromise on either server CPU time or network bandwidth.

Our CPIR scheme, FastPIR, works without recursion and thus keeps the smaller CPIR answer size. However, it optimizes the computation time for ANSWER. In fact, FastPIR

[1] A technique called aggregation [4, 11] further combines multiple rows (messages) into wider rows, resulting in a matrix with n/a rows, where the value a depends on the size of each message.

takes less time than both XPIR and SealPIR (with or without recursion) to run ANSWER, particularly when the number of messages n in the library is greater than a threshold (≈20K; §6.5), thereby improving the scalability and message latency of Addra. FastPIR may be a good fit for other applications of CPIR where costs are dominated by those of the CPIR ANSWER procedure.

FastPIR, like SealPIR [7], builds on the lattice-based homomorphic encryption scheme of Brakerski/Fan-Vercauteren (BFV) [15, 33]. BFV offers superior efficiency than a traditional number-theoretic homomorphic encryption scheme such as Paillier [61], resists attacks by quantum computers, is implemented in mature and actively maintained codebases [2, 69], and is in the preliminary stages of being standardized (e.g., with ISO/IEC) [6]. We start with a necessary background on BFV (§4.1), and then delve into the details of FastPIR (§4.2–§4.4).

4.1 Background: The BFV cryptosystem

We focus here on describing the more efficient vectorized variant of BFV in which a single homomorphic operation operates over multiple plaintext inputs (single instruction, multiple data or SIMD; also called batching in the literature).

In this BFV variant, a plaintext is a vector of dimension N, where the parameter N equals a power of two and is at least 2^{10} for the security of the BFV scheme [6]. Each component of the plaintext is an integer in $\mathbb{Z}_p = \{0, \ldots, p-1\}$, the set of integers modulo p. Sometimes, we will view a BFV plaintext as a matrix with two rows and $N/2$ columns rather than a vector with dimension N.

A BFV ciphertext is also a vector but of dimension $2 \cdot N$. Each of its component is an element of $\mathbb{Z}_q$, where $q \gg p$.

The BFV encryption procedure, BFV.ENC, adds noise when it converts a plaintext vector into a ciphertext vector. This noise grows as homomorphic operations are performed on the ciphertext. If the noise grows beyond a threshold, then the ciphertext decryption procedure BFV.DEC does not produce the correct plaintext. Hence, $q \gg p$ for enough noise budget.

The size of the plaintext vector, N, the size of the domain of each component of the plaintext, p, and the size of the domain of each component of the ciphertext, q, are all tunable parameters. Typically, one picks a combination of p, q, N depending on the application, the required noise budget, and the desired security level; we discuss concrete values for these parameters for Addra in §5.

BFV supports the following homomorphic operations that are used in FastPIR:

- **BFV.ADD**(c_0, c_1) takes as input encryptions c_0 and c_1 of plaintext vectors v_0 and v_1, and outputs an encryption of $v_0 + v_1$ (component-wise vector addition).
- **BFV.SCMULT**(v_0, c_1) takes as input a plaintext vector v_0 and an encryption c_1 of a plaintext vector v_1, and produces an encryption of the product $v_0 \odot v_1$, where the operator $\odot$ denotes component-wise multiplication.
- **BFV.ROWROTATE**(c_0, i) takes as input an encryption c_0 of a plaintext v_0 and an integer $0 < i < N/2 - 1$, and produces an encryption of v_0 rotated right by i positions cyclically row-wise. For instance, if plaintext dimension is $N = 8$ and v_0 is $((a,b,c,d),(e,f,g,h))$ in its matrix representation, then a right rotation by $i = 1$ produces an encryption of $((d,a,b,c),(h,e,f,g))$.
- **BFV.COLROTATE**(c_0) takes as input an encryption c_0 of a plaintext v_0 and returns an encryption of a plaintext produced by swapping the two rows of v_0. For the example above, the result is an encryption of $((e,f,g,h),(a,b,c,d))$.

The BFV homomorphic operations require public keys generated by a key generation procedure. In particular, the rotation procedures require a set of rotation keys. While BFV.COLROTATE requires one key, the size of the set of keys for BFV.ROWROTATE can vary. On the one extreme, this set can be configured to contain one key that rotates the plaintext vector by one position. Thus, to perform a rotation by $i > 1$ positions, BFV.ROWROTATE calls itself i times, incurring i times the cost of one BFV.ROWROTATE operation. On the other extreme, the set can contain $N/2 - 1$ keys for all possible values of i between 0 and $N/2$. This extreme reduces CPU time for BFV.ROWROTATE as it does not call itself recursively, but this configuration increases the key size. For the BFV parameters we choose (§5), each rotation key is 128 KiB, and the set of all possible rotation keys is 256 MiB. Thus, in practice, one generates $\log_2(N/2)$ keys for all powers-of-two between 0 and $N/2 - 1$, and each invocation of BFV.ROWROTATE calls itself recursively up to $\log_2(N/2)$ times.

4.2 The FastPIR scheme

Recall the CPIR scenario (§4): a server holds a library L of n messages where each message has m components, while a client holds an integer $0 \leq idx \leq n-1$ and wants to retrieve the idx-th library message without revealing idx to the server.

To build intuition for FastPIR, suppose that L is an $N \times 1$ matrix consisting of N unit length messages, where N is the plaintext vector dimension in BFV. Then, the client constructs the CPIR query q for the idx-th message by encrypting a BFV plaintext whose idx-th entry is one and the rest are zeros (this is called *one-hot encoding* of idx). For instance, if $N = 4$ and $idx = 1$, the client encrypts the BFV plaintext $(0,1,0,0)$. The server multiplies this encryption q with L by computing BFV.SCMULT(L, q) to obtain an encryption of the idx-th entry of L. For the example above, if L is (a_0, a_1, a_2, a_3), BFV.SCMULT produces an encryption of $(0, a_1, 0, 0)$ as the multiplication is component-wise. The client receives the output and decrypts it to get a_1.

The advantage of this strategy is that a query consumes only a component of a ciphertext for each of the n rows of L (instead of a ciphertext per row). However, a challenge is that this strategy generates one output ciphertext for each of the m columns of L. FastPIR addresses this challenge by combining ciphertexts for m columns into a single cipher-

```
1:  function QUERY(index idx, n)
2:      // Create a one-hot encoding of idx
3:      for i = 0 to n − 1 do
4:          f_i ← (i == idx) ? 1 : 0
5:      // Split and encrypt the one-hot vector
6:      for i = 0 to (n/N) − 1 do // N is BFV plaintext dimension
7:          q_i = BFV.ENC(pk, (f_{i·N}, ..., f_{(i+1)·N−1}))
8:      return q = (q_0, ..., q_{(n/N)−1})
9:  function ANSWER(library L, query q = (q_0, ..., q_{(n/N)−1}))
10:     // Represent L as a matrix of elements in Z_p: L ∈ Z_p^{n×m}
11:     // q is an output of QUERY
12:     for j = 0 to m − 1 do
13:         sum_j = BFV.ENC(pk, 0)
14:         for i = 0 to (n/N) − 1 do
15:             p_{i,j} ← SUBMAT(L, i·N, (i+1)·N − 1, j, j)
16:             t_{i,j} = BFV.SCMULT(p_{i,j}, q_i)
17:             sum_j = BFV.ADD(sum_j, t_{i,j})
18:     // Combine outputs from all columns
19:     Initialize s_top, s_bot to encryptions of zero vectors
20:     for j = 0 to m − 1 do
21:         if j < N/2 then
22:             sum_j ← BFV.ROWROTATE(sum_j, j)
23:             s_top ← BFV.ADD(s_top, sum_j)
24:         else
25:             sum_j ← BFV.ROWROTATE(sum_j, j − N/2)
26:             s_bot ← BFV.ADD(s_bot, sum_j)
27:     return BFV.ADD(s_top, BFV.COLROTATE(s_bot))
28: function DECODE(answer ans, index idx)
29:     // ans is an output of ANSWER
30:     ans_pt ← BFV.DEC(sk, ans)
31:     if idx mod N > N/2 then
32:         ans_pt ← PTCOLROTATE(ans_pt)
33:     return ans_pt ← PTROWROT(ans_pt, N/2 − (idx mod N/2))
```

Figure 3—QUERY, ANSWER, and DECODE procedures for a basic version of FastPIR. (pk, sk) are a (public, private) key pair for the BFV scheme (§4.1). SUBMAT extracts a sub-matrix of a matrix. PTROWROT and PTCOLROTATE are like BFV.ROWROTATE and BFV.COLROTATE, respectively except they operate on BFV plaintexts rather than BFV ciphertexts.

text using the BFV rotation operations (BFV.ROWROTATE and BFV.COLROTATE), thereby reducing CPIR answer sizes.

Before describing the details of rotation, we remark that the use of vectorized operations (SIMD capabilities of BFV) is common. In fact, both XPIR and SealPIR use vectorized operations. The difference is that these prior CPIR schemes apply vectorization across columns of the matrix while FastPIR applies it across rows of the matrix, which is a more efficient use of vectorization in the PIR context (§6.5).

Details. Figure 3 shows the FastPIR scheme. It assumes that n is a multiple of N, i.e., $n = k \cdot N$ for some $k \geq 1$, and $m \leq N$. If these constraints do not hold, then the server pads L with empty rows and splits L into sets of N columns.

The QUERY procedure and the top half of ANSWER (until line 17) follow the intuition described above. That is, QUERY creates a one-hot encoding of *idx* (line 4 in Figure 3), splits the encoding into multiple BFV plaintexts, and encrypts each plaintext separately (line 7 in Figure 3). The top half of ANSWER multiplies the $k = n/N$ plaintext column vectors of each column of L with the corresponding ciphertexts in the query (line 16 in Figure 3), and adds the k output ciphertexts to get one ciphertext per column of L (line 17 in Figure 3). For instance, if $n = 8$, $N = 4$, $idx = 1$, and a column of L is $(a_0, a_1, \ldots, a_7)$, then ANSWER computes encryptions of $(0, a1, 0, 0)$ and $(0, 0, 0, 0)$ in line 16 of Figure 3, and adds them to get an encryption of $(0, a1, 0, 0)$ in line 17 of Figure 3.

The bottom half of ANSWER packs together outputs from each column into a single ciphertext (lines 19–27 in Figure 3). Suppose the number of columns is $m = 4$ and the outputs corresponding to them are encryptions of $(0, a_1, 0, 0)$, $(0, b_1, 0, 0)$, $(0, c_1, 0, 0)$, and $(0, d_1, 0, 0)$, or equivalently encryptions of $((0, a_1), (0, 0))$, $((0, b_1), (0, 0))$, $((0, c_1), (0, 0))$, and $((0, d_1), (0, 0))$, when the underlying plaintexts are viewed in their matrix form. Then, ANSWER uses the BFV.ROWROTATE and BFV.ADD operations to produce encryptions of $((b_1, a_1), (0, 0))$ and $((d_1, c_1), 0, 0))$ (lines 20–26 in Figure 3), before column rotating the second ciphertext, and adding the result to the first ciphertext to obtain an encryption of $((b_1, a_1), (d_1, c_1))$ (line 27 in Figure 3). Using rotations to pack outputs from multiple columns into a single ciphertext is crucial as otherwise a CPIR answer size can contain multiple ciphertexts (instead of one).

DECODE is straightforward; it decrypts the output of ANSWER and then rotates the plaintext depending on the value of the requested index. For the example above, DECODE first obtains the plaintext matrix $((b_1, a_1), (d_1, c_1))$, and then performs a rotation on this matrix by $idx = 1$ to obtain $((a_1, b_1), (c_1, d_1))$.

4.3 Reducing the CPU cost of rotations

Recall that one goal of FastPIR is to optimize the CPU time of ANSWER procedure (§3.2). A source of inefficiency in what is described above is the cost of BFV.ROWROTATE (lines 22 and 25 in Figure 3), as the CPU time taken by it depends on the value of i—the positions by which the underlying plaintext is rotated. When i is a power of two, then BFV.ROWROTATE is fast, whereas when i is a not a power of two, BFV.ROWROTATE calls itself up to $\log_2(i+1)$ times (§4.1). For example, a call to BFV.ROWROTATE with an input $i = 7$ translates into three rotations by amounts one, two, and four—powers of two that add to seven.

FastPIR eliminates the calls to expensive rotations whose input rotation amount is not a power of two. As intuition, suppose that the ANSWER procedure (Figure 3) needs to make two calls to BFV.ROWROTATE—one for rotating a vector by two positions and the other for rotating a vector for another matrix column by three positions. Then, the straw man design presented in the previous subsection treats each rotation separately. Particularly, it breaks down the rotation by three

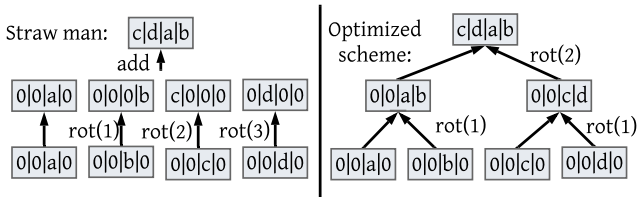


Figure 4—Illustration of optimized rotations in FastPIR. The straw man (left) performs a mix of slow rotations (with rotations amounts that are not powers of two) and fast rotations (with rotation amounts that are powers of two) to combine multiple vectors. FastPIR's optimized scheme combines vectors using fast rotations only.

positions into a rotation by one position followed by a rotation by two positions. Instead, FastPIR first rotates the second vector by one position and adds the result to the first vector. Then, it rotates the combined vector once by two positions, thereby rotating only by powers-of-two amounts.

Figure 4 illustrates the idea for our running example with $m = 4$ matrix columns, where the FastPIR processing for each column produces a ciphertext. FastPIR arranges the vectors to be combined as leaf nodes of a tree; it then builds up to the root of the tree. When producing a parent at a given height h of the tree, FastPIR rotates the right child by 2^{h-1} positions and adds the rotated vector to the left child. The effect is that FastPIR combines m ciphertexts in lines 22 and 25 in Figure 3 using m fast rotations.

4.4 Reducing the number of rotations

This optimization reduces the number of calls to BFV.RowRotate by a factor of two, and eliminates the call to BFV.ColRotate, thereby further reducing the CPU cost of Answer. The trade-off is a 2× increase in CPIR query size.

The key idea is to exploit the matrix representation of a BFV plaintext (§4.1) and retrieve two elements of a matrix row (instead of one) at a time.

As motivation, suppose that the matrix L is of dimension $N/2 \times 2$, and the client wants the idx-th row. Then, the client sends an encryption of a vector whose idx-th and $idx + N/2$-th entries are one (and the rest are zeros). For instance, if $N = 4$ and $idx = 1$, then the client sends an encryption of $(0,1,0,1)$, or equivalently, $((0,1),(0,1))$. The server multiplies this query with L to get an encryption of a vector whose idx-th and $idx + N/2$-th entries are the desired elements from the two columns of L. As an example with $idx = 1$, say L is $((a_0,a_1),(b_0,b_1))$, then the multiplication operation produces an encryption of $((0,a_1),(0,b_1))$.

Figure 5 shows the procedures of FastPIR with this optimization. The procedures assume that n is a multiple of $N/2$, i.e., $n = k \cdot (N/2)$ for some $k \geq 1$, and m is even and $\leq N$. As before (§4.2), if these constraints do not hold, then the server appropriately pads and splits L.

The Query procedure encrypts a set of vectors that in total contain two non-zero entries (line 6 in Figure 5). The Answer procedure multiplies k parts of every pair of columns of L with the k ciphertexts in the query, and adds the results to

```
1:  function Query(index idx, n)
2:      for i = 0 to n − 1 do
3:          f_i ← (i == idx) ? 1 : 0                    // one-hot encoding
4:      for i = 0 to n/(N/2) − 1 do
5:          v ← (f_{i·N/2}, ..., f_{(i+1)·N/2−1})
6:          q_i = BFV.Enc(pk, v||v)      // || denotes concatenation
7:      return q = (q_0, ..., q_{n/(N/2)−1})

8:  function Answer(library L, query q = (q_0, ..., q_{n/(N/2)−1}))
9:      // Represent L as a matrix of elements in Z_p: L ∈ Z_p^{n×m}
10:     // q is an output of Query
11:     for j = 0 to (m/2) − 1 do
12:         sum_j = BFV.Enc(pk, 0)
13:         for i = 0 to n/(N/2) − 1 do
14:             p_{i,j} ← SubMat(L, i·N/2, (i+1)·N/2−1, 2j, 2j+1)
15:             t_{i,j} = BFV.ScMult(p_{i,j}, q_i)
16:             sum_j = BFV.Add(sum_j, t_{i,j})
17:     // Combine outputs from all pairs of columns
18:     return RotateAndCombine(sum_0, ..., sum_{m/2−1})

19: function Decode(answer ans, index idx)
20:     // ans is an output of Answer
21:     ans_pt ← BFV.Dec(sk, ans)
22:     return PtRowRot(ans_pt, N/2 − (idx mod N/2))
```

Figure 5—Query, Answer, and Decode procedures for FastPIR. (pk, sk) is a (public, private) key pair for the BFV scheme (§4.1). SubMat extracts a sub-matrix of a matrix. RotateAndCombine refers to the optimized procedure to combine ciphertexts (§4.3). PtRowRot is like BFV.RowRotate except that it operates on BFV plaintexts rather than BFV ciphertexts.

get one ciphertext for every pair of columns. Then, Answer packs these outputs using the optimized scheme to combine ciphertexts described previously (§4.3). The Decode procedure decrypts the output of Answer and performs a rotation on the plaintext output.

Security analysis. The security of a CPIR scheme requires the output of Query to not reveal any information about the requested index [20, 43]. FastPIR meets this property because its Query procedure (i) produces semantically-secure BFV ciphertexts, and (ii) outputs $n/(N/2)$ ciphertexts independent of the value of the desired index idx.

5 Implementation details

FastPIR. Our prototype of FastPIR is ≈1000 lines of C++ and is available at https://github.com/ishtiyaque/FastPIR. We used the Microsoft SEAL library v3.5 [69] for the underlying cryptographic operations of the BFV scheme. Recall that FastPIR configures BFV so that it supports vectorized operations (§4.1). For vectorization, the plaintext modulus p has to be a prime number congruent to 1 $\pmod{2N}$, where N is the vector dimension of a BFV plaintext and equals 2^{10} or a higher power of two (§4.1). Moreover, one needs to choose $p \ll q$ to ensure correct decryption. For Addra, we choose $N = 2^{12}$, p a 19-bit prime 270337, and q a 109-bit composite that is the product of

a 54-bit prime (18014398509309953) and a 55-bit prime (36028797018652673). These parameters provide a 128-bit security level as guided by the homomorphic encryption standard [6]. (One may choose different parameters for FastPIR based on application requirements.)

Master-worker architecture for Addra. We implemented Addra server using a master-worker architecture with many worker machines to distribute the PIR workload. Specifically, during the dialing phase of a round in Addra's protocol (§3.2), the master receives CPIR queries from all devices and shards them across the workers, where a worker gets a subset of the queries. Then, during the communication phase, the master initiates each subround at a fixed schedule. During each subround, it waits to receive messages from the clients, compiles them into a message library, and broadcasts the entire message library to the workers. In case a laggard client fails to get its message to the master during the time period the master waits for incoming messages, the master buffers the laggard's message for the next subround. If more than one message arrives at the master from a client for the same subround, the master retains the latest message. Meanwhile, to process CPIR queries, each worker computes the output of ANSWER on its assigned subset of the queries and pushes the outputs to the client devices who registered the queries.

Dialing protocol. Addra uses Pung's protocol to initiate calls [11, Chapter 4.5.3] (which in turn is based on Alpenhorn [50]). Briefly, a caller sends "hello" messages encrypted with the callee's public key to the server, who then broadcasts the set of "hello" messages from all callers to all user devices. A callee decrypts the ciphertexts using its private key and learns the content encryption key and the caller's phone number (which are inside the hello message). This protocol is not efficient as the server broadcasts the ciphertexts to the participants (although the server could use a CDN or multicast protocols), and a callee decrypts ciphertexts from all users. Thus, Addra runs this protocol infrequently (every five minutes; §6.3). A more efficient dialing protocol in Addra's threat model is still an open problem.

Options for which call to pick. A device may receive multiple incoming calls, or may make an outgoing call at the same time a call comes in. In such scenarios, Addra exposes all options to the device owner and lets them pick the call they want to participate in. However, depending on which option a user chooses, they could leak some information to the users who are on the other end in the non-chosen options. For instance, if Alice receives a call from both Bob and Charlie, and decides to pick Bob's call, then Charlie may infer that Alice is busy. This leakage is not specific to Addra but applies to any metadata-private system [8, 9]. As efficient solutions to this problem become available, one could enhance the options-based approach currently implemented in Addra.

Other libraries and lines of code. Our prototype of Addra (`https://github.com/ishtiyaque/Addra`) is ≈2,000 lines of C++ on top of existing libraries, including FastPIR. Our implementation of the dialing protocol uses the libscapi [1] library for public-key encryption using the Cramer-Shoup scheme [26] with a key size of 3072 bits which provides 128 bits of security. It also uses AES-CBC implementation from OpenSSL with a 128-bit key for end-to-end content encryption with 128 bits of security. It implements the message library broadcasting mechanism from master to workers using rpclib [3]. Finally, we use the open source implementation of LPCNet [57] for speech encoding/decoding.

6 Evaluation

Our evaluation answers the following questions:

1. What is Addra's message latency, and how does it vary with the number of users and server machines?
2. How much resource overhead (CPU, network upload and download) does Addra impose on its server and users?
3. How does Addra compare to Pung [7, 10, 11], which is the state-of-the-art prior system for metadata-private communication over completely untrusted infrastructure?
4. How does FastPIR compare to the state-of-the-art CPIR schemes, XPIR [4] and SealPIR [7]?

A highlight of our evaluation results is as follows:

- Addra's 99-th percentile message latency is 726 ms for 32,768 users and 80 server machines. For the same configuration, Pung's message latency is 5.2 seconds.
- Addra's server consumes 22.3 minutes of CPU time for a subround with 32,768 users, where a subround corresponds to 480 ms of voice call. Translated to provisioning burden, each user requires the server to provision 0.085 CPU for its call. In contrast, Pung consumes 77.1 minutes of CPU time (3.45× higher) per subround.
- An Addra user downloads and uploads 55.1 and 1.08 MiB of data for each round when 32,768 users use Addra, where a round corresponds to five minutes of voice call. Thus, translated into bandwidth, Addra requires a download and upload bandwidth of 1.46 Mbps and 30 Kbps, respectively. In contrast, a Pung client downloads and uploads 250 MiB (4.6× higher) and 313 MiB (289× higher) for five minutes of voice call data.
- FastPIR has a smaller server-side CPU time *and* a smaller response size relative to XPIR and SealPIR, particularly when the number of messages in the PIR library is greater than a threshold (≈ 20K).

Setup and method. We compare Addra to two variants of Pung: Pung-XPIR (P-XPIR) and Pung-SealPIR (P-SPIR). The former is the original Pung system from OSDI 2016 [10] that instantiates CPIR with the XPIR scheme [4]. The second variant replaces the XPIR scheme with the SealPIR CPIR

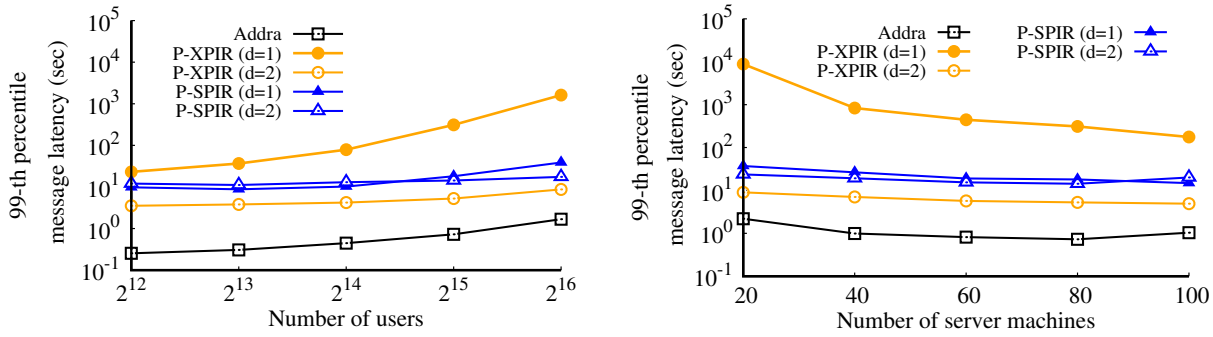


Figure 6—(Left) Message latency with a varying number of users for eighty server worker machines. (Right) Message latency with a varying number of server worker machines for 32,768 users. Messages are 96 bytes in size. The y-axis is log-scaled. d denotes CPIR recursion depth, where $d = 1$ denotes no recursion and $d = 2$ enables recursion. Addra does not use recursion (§4).

scheme [7]. We include both variants as there is no clear winner between them across all performance metrics. Further, we evaluate these variants without ($d = 1$) and with CPIR recursion ($d = 2$). We do not experiment with a recursion depth $d > 2$ as the server CPU time and the network transfers from the server to the clients, which are the two key overhead metrics, grow significantly with depth greater than two [7].

We configure Addra and Pung to provide a security level of 128-bits. Also, we configure Pung to use its BST retrieval scheme in which a message recipient obliviously searches through a tree while retrieving one message from the Pung server. This scheme is the most scalable retrieval scheme for Pung especially as the number of system users increase; we discuss other retrieval schemes Pung supports in the related work section (§7). For all of the systems, we deploy the server on a cluster of machines in AWS EC2 US East region (Ohio). Addra requires a master machine and a set of worker machines (§5). For the master, we use a machine of type `c5.24xlarge` (96 vCPU, 192 GiB of RAM and 25 Gbps of network bandwidth) which provides a high network bandwidth to enable the master to broadcast the message library (the mailboxes) to the workers. For the workers, we use the compute-optimized machines of type `c5.12xlarge` (48 vCPU, 96 GiB of RAM, and 12 Gbps of network bandwidth). Pung does not have a master and therefore we use machines of type `c5.12xlarge` as its workers. To compensate for the extra master machine assigned to Addra (relative to Pung), we assign two additional worker machines of type `c5.12xlarge` to Pung.

Addra is required to process queries from all clients in every subround to meet its security goals. Since we cannot run tens of thousands of clients in our infrastructure, we employ a combination of real and simulated clients. We deploy 256 geographically distant real clients in a machine of type `c5.24xlarge` in AWS US West (N. California). The mean network RTT, as measured by Ping, between the server and these clients is 51 ms. During each round and subround, real clients send their queries and messages to the server, and the server inserts the queries and messages of the remaining simulated clients.

We configure Addra to run a round every five minutes and a subround every 480 ms. This configuration results in a fixed message size of 96 bytes at each subround as the LPCNet voice codec encodes a 40 ms audio frame into 8 bytes (§2.1) [74, 75]. We vary the number of users (from 4,096 to 65,536) and the number of worker machines (from 20 to 100). We repeat experiments for 10 trials. To account for tail latency, we process the queries from real clients only after processing the queries from all simulated clients. Then, we measure the 99-th percentile latency observed by the real clients over the 10 trials, the CPU time consumed by the server and the real clients, and the amount of data uploaded and downloaded by the real clients.

6.1 Message latency

Variation with the number of users. Figure 6 (left) shows the 99-th percentile message latency with a varying number of users when the server has 80 worker machines.

Addra's message latency is 254 ms for 4,096 users and increases to 1678 ms for 65,536 users. This increase is due to three reasons. First, as the number of users increases, so does the number of mailboxes and the time to broadcast their content from the master to the workers (§5). Second, the number of CPIR queries the server processes every subround equals the number of users (§3.2). Third, the time to process a CPIR query increases with the number of mailboxes, so each worker takes longer to generate CPIR responses. For 32,768 users, the latency is 726 ms, of which 398 ms is for CPIR query processing at the workers, 186 ms is for broadcast of mailbox content from the master to the workers, and the rest is for network transfers between the client and the server. However, for 65,536 users, the latency increases to 1,678 ms, of which 1,186 ms is for CPIR query processing alone. This processing time is higher than the 480 ms subround time budget and thus voice packets start queuing up at the server for these many users.

Addra's message latency is lower than Pung's, specifically, that of Pung-XPIR by a factor of 7.2× for 32,768 users, due to two reasons. First, a sender in Addra pushes a message to the server, who performs CPIR processing and pushes the

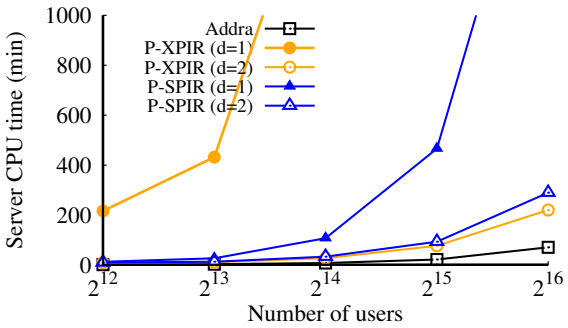


Figure 7—Server-side CPU time per subround with a varying number of users. A subround corresponds to 480 ms of voice call; in a subround, each user sends and receives one 96 byte message.

response to the recipient—in total, the message traverses two hops (§3.2). In contrast, while the sender in Pung pushes a message to the server in one hop, a recipient has to make $\lceil \log_2(n+1) \rceil$ sequential round-trips to the server to fetch a message, where n is the number of users. Second, Addra uses FastPIR, which has lower server-side CPIR answer generation time than XPIR or SealPIR used in Pung; we will expand on this difference shortly (§6.2, §6.5).

Variation with the number of worker machines. Figure 6 (right) shows the 99-th percentile message latency as a function of the number of worker machines when the number of users is fixed to 32,768. Latency decreases for all systems with an increase in the number of worker machines due to increased parallelization for CPIR answer generation, but only up to an inflection point. Beyond this inflection point, adding workers does not improve latency as the time to replicate mailboxes from the master to the workers goes up, while the CPU on the workers starts to become idle. Thus, an immediate scalability bottleneck in Addra is the time to broadcast mailboxes from the master to the workers. Distributing the master or reducing the number of workers by extracting more efficiency from each may further push out the inflection point.

6.2 Server-side CPU consumption

Figure 7 shows that server-side CPU time increases with the number of users. This is expected as both the number of CPIR queries and the time to generate an answer for each query increases with the number of users (§3.2). Addra's CPU consumption is lower than Pung's. For instance, for 32,768 users, Addra takes 22.3 minutes while Pung (with XPIR and CPIR recursion depth $d = 2$) takes 77.1 minutes (3.45× higher). If we convert these times to CPU provisioning requirements, then for each subround lasting 480 ms or 0.48 seconds, Addra's server consumes 22.3 minutes, or 1,338 seconds, of CPU, which is provided by provisioning $1,338/0.48 = 2788$ CPUs, or 0.085 CPU per user. Similarly, each Pung user requires 0.29 CPU per user. A key reason for this difference is that FastPIR in Addra consumes lower amount of server-side CPU relative to XPIR or SealPIR in Pung (§6.5). We note that even though a recursion depth of $d = 2$ reduces CPU consumption relative to no recursion ($d = 1$), increasing depth further ($d = 3$) does not reduce CPU consumption [7]. Furthermore, a higher depth increases network overhead (§6.3). Thus, as mentioned earlier (§6), we restrict our experiments to a depth of $d = 2$.

6.3 Client-side resource overheads

Network transfers. Figure 8 shows the amount of data a client downloads and uploads for one round of communication (a round corresponds to five minutes of voice call).

An Addra user downloads ≈55.1 MiB in a five-minute round when 32,768 users use Addra. That is, each user requires 1.46 Mbps of network download bandwidth. Of the 55.1 MiB, ≈39 MiB is due to the communication phase of the round while the rest is due to the dialing phase (§3.2). Further, the former is independent of the number of system users, while the latter depends linearly on the number of users.

Relative to a non-private baseline which does not hide metadata, Addra's network overhead is significantly higher due to the use of CPIR, which encrypts messages into BFV ciphertexts. For example, if the non-private baseline uses LPCNet which encodes 480 ms of speech in 96 bytes of data, then a user's network download bandwidth will be 1.56 Kbps. In contrast, Addra encrypts the 96 bytes into a 64 KB ciphertext, which is a 682× increase.

However, relative to Pung, an Addra user downloads less data, by 4.5–45.7×, depending on the Pung variant. The improvement is due to two reasons. First, Pung requires a message recipient to make multiple CPIR queries with the server to search through the message library that is organized as a tree. Second, CPIR answer size increases with a higher CPIR recursion depth ($d = 2$ versus $d = 1$). Addra's FastPIR, on the other hand, operates at $d = 1$ to keep CPIR answer sizes, and thus the downloads, smaller (§4).

An Addra user uploads one CPIR query per round during its dialing phase. The Addra server then reuses the query across subrounds (§3.2). Even though the query size for Addra is larger compared to that of Pung (§6.5), unlike Pung, this cost is amortized over multiple subrounds of the communication phase. As a result, Addra's upload network transfers are small: ≈1.1 MiB per round, or 30 Kbps.

We remark that even though Addra's instantaneous network overhead (1.46 Mbps download and 30 Kbps upload) appears manageable, it adds up over time due to the involvement of a client in dummy calls (§3.2). Thus, Addra requires certain conditions such as unlimited network downloads for its clients to be deployable. We anticipate that in the future, as the need for privacy increases, so will advances in network technology that will provide options for unlimited data.

CPU time. An Addra client consumes ≈27.5 seconds of CPU time per a five-minute round when the number of users is 32,768. 94% of this time is from the dialing protocol (§5). For the same configuration, a Pung client consumes 1.7–63× higher CPU, primarily due to multiple CPIR queries with the server for transmitting each message.

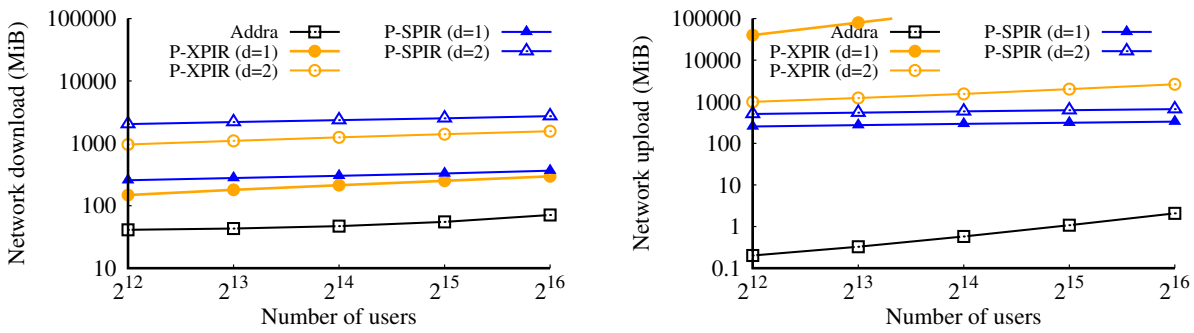


Figure 8—Data downloaded and uploaded by a user per round with varying number of users. A round corresponds to five minutes of voice call.

6.4 Discussion on voice quality

The quality of voice calls and user experience depends on several factors including message transmission latency, jitter (the inconsistencies among packet arrival intervals [41]), and the effectiveness of the voice encoder that converts human speech into a digital signal. This section briefly discusses Addra's performance on these metrics.

We reported Addra's message transmission latency in §6.1. Specifically, latency varies with the number of users, and is lower for a lower number of users. For example, for 8K users, the latency is 306 ms, which is below the ITU-G.114 recommended value of 400ms [40]. As the number of users increases, Addra's latency crosses the recommended value, but stays below one second for a significant number of users (32,768); this value of one second is critical as it is possible to make voice calls at this latency [49].

To measure jitter, we ran Addra for one round (i.e., 5 minutes of voice call) with 80 worker machines and a varying number of users. Ideally, a user should receive a voice packet every 480 ms, which is the duration of one subround. We measured the interval between consecutive packet arrival timestamps and calculated the absolute deviation of this value from 480 ms as jitter. Addra's mean jitter is 4.1 ms for 4,096 users and increases to 36.8 ms for 32,768 users. This increase with the number of users is correlated with higher CPU and network load at the server.

Finally, the effectiveness of voice encoding is a property of the encoder. Addra's current prototype uses the LPCNet [57] encoder developed by Mozilla. Conducting a user experience study on LPCNet's quality is outside the scope of this paper, but we refer the reader to the original paper on LPCNet that discusses a subjective assessment of LPCNet's quality based on an experiment with one hundred human listeners [75].

6.5 Comparison of CPIR schemes

A core component of Addra and Pung is the CPIR cryptographic primitive. Pung uses either XPIR or SealPIR, which are also the state-of-the-art schemes. Addra uses FastPIR (§4). This section compares the cost of these CPIR schemes in isolation. Besides, since CPIR applies to several other contexts [14, 30, 36, 56], this section sheds light on which scheme could be better for which application.

We microbenchmarked the XPIR, SealPIR, and FastPIR libraries on a single CPU of an AWS instance of type `c5.12xlarge` (48 vCPU, 3.6 GHz, 96 GiB RAM). We configured all three libraries for a 128-bit security level. However, XPIR does not set parameters from the homomorphic encryption standard [6], and its parameters are smaller relative to those for SealPIR and FastPIR.

We varied the number of messages in the library ($n \in \{2^{13}, \ldots, 2^{20}\}$) and the size of each message ($m \in \{96\text{B}, 256\text{B}, 1024\text{B}\}$). The lowest value of n captures a small library with a few thousand messages, while the other extreme of $n = 2^{20}$ demonstrates how FastPIR scales with the number of messages relative to the other CPIR libraries. Similarly, the different message sizes demonstrate performance for scenarios with small messages (for example, Addra) and also larger messages.

We measure and report both CPU and network overhead for query generation (QUERY), answer generation (ANSWER), and answer decode (DECODE) CPIR procedures, for 10 trials. Given that the CPIR cost in Addra is dominated by the cost to run the ANSWER procedure, we describe the results while focusing on ANSWER. At a high level, FastPIR keeps both the CPU cost for ANSWER and the size of ANSWER output small, while XPIR and SealPIR sacrifice one of the two.

CPU time for ANSWER. Figure 9 shows the CPU time for the ANSWER procedure for different values of n and m. Beyond a threshold n, and for all values of m, FastPIR consumes the least amount of CPU time for ANSWER independent of whether the baselines use recursion or not ($d = 1$ is no recursion, and $d = 2$ enables it). For instance, when n=2^{20} and m=256B, ANSWER in FastPIR takes 2.5× less time than XPIR ($d = 2$) and 2.7× less time than SealPIR ($d = 2$).

The figure also shows the impact of FastPIR's optimizations (§4.3, §4.4) in reducing its CPU overhead. For smaller values of n, the impact of these optimizations is significant. For instance, for n=2^{15} and m=256B, FastPIR without the two optimizations (F-1 in the figure) is 2.73× more expensive than the full-fledged FastPIR, while FastPIR without its last optimization in §4.4 (F-2 in the figure) is 1.45× more expensive than FastPIR will all optimizations enabled. But,

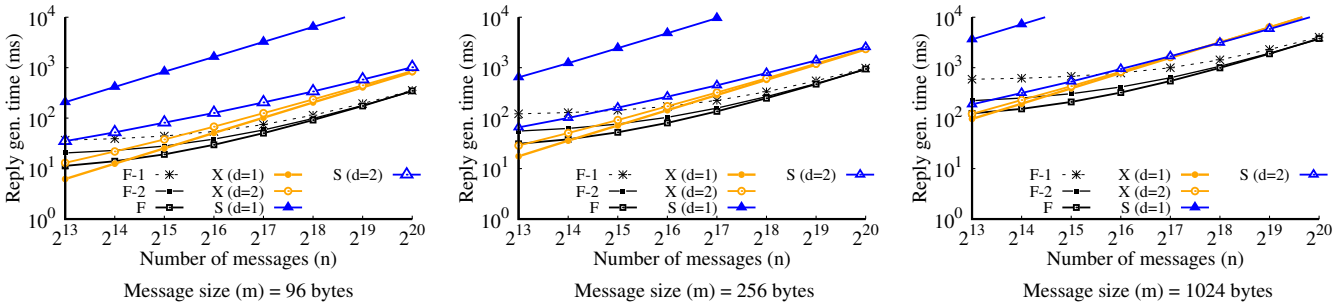


Figure 9—CPU time to run the ANSWER CPIR procedure for XPIR (X), SealPIR (S), and three variants of FastPIR (F) with a varying number of messages n in the server library and a varying size m of each message. F-1 and F-2 are intermediate baselines for FastPIR: F-1 leaves out both optimizations for the rotation operations (§4.3 and §4.4), while F-2 leaves out only the optimization in §4.4. Both axes are log-scaled. d denotes recursion depth (§4). FastPIR does not use recursion (sets $d = 1$, which means no recursion). For both XPIR and SealPIR, an optimization to aggregate multiple small messages into a larger one (called aggregation in the literature) is enabled.

	$n = 32{,}768$					$n = 1{,}048{,}576$				
	X ($d=1$)	X ($d=2$)	S ($d=1$)	S ($d=2$)	F ($d=1$)	X ($d=1$)	X ($d=2$)	S ($d=1$)	S ($d=2$)	F ($d=1$)
query size (KiB)										
$m = 96$ bytes	33,856	2,112	32	64	1,024	1,082,432	11,776	928	64	32,768
$m = 256$ bytes	95,328	3,520	96	64	1,024	3,050,432	19,776	2,752	64	32,768
$m = 1024$ bytes	524,288	8,192	512	64	1,024	16,777,216	46,368	16,384	64	32,768
answer size (KiB)										
$m \in \{96B, 256B, 1024B\}$	32	256	32	320	64	32	288	32	320	64
client CPU costs (ms)										
QUERY ($m = 96B$)	118.6	7.4	0.7	1.4	21.3	3801.8	41.5	19.2	1.4	679.0
QUERY ($m = 256B$)	335.2	12.4	2.0	1.4	21.4	10711.3	69.8	56.9	1.4	678.6
QUERY ($m = 1024B$)	1841.6	28.8	10.6	1.4	21.4	58990.8	164.2	338.8	1.4	678.7
DECODE ($m \in \{96B, 256B, 1024B\}$)	0.1	0.41	0.19	1.88	0.36	0.1	0.37	0.2	1.86	0.41

Figure 10—Network costs and client-side CPU costs for XPIR (X), SealPIR (S), and FastPIR (F) with a varying number of messages (n) and the size of each message (m) in the server library. d denotes recursion depth (§4). FastPIR does not use recursion (sets $d = 1$, which means no recursion). For both XPIR and SealPIR, an optimization to aggregate multiple small messages into a larger one (called aggregation in the literature) is enabled.

as n increases the lower CPU time benefit of the optimizations diminishes. This trend is expected as for larger n the cost for the ANSWER procedure is dominated by the time to run the BFV.SCMULT and BFV.ADD operations rather than the rotation operations, which is what the optimizations focus on (§4.2–§4.4).

Output size of ANSWER. Figure 10 shows the size of the CPIR response generated by the ANSWER procedure for the three CPIR schemes. When the schemes do not use recursion ($d = 1$), their answer output sizes are smaller relative to when they use recursion, although FastPIR's response size is double the size of XPIR and SealPIR. However, $d = 1$ is not a viable solution for either XPIR or SealPIR. For XPIR, the query size is large for $d = 1$, which increases network bandwidth and CPU time for processing of CPIR queries (Figure 7). For SealPIR, the compressed query is smaller on the wire, but the expanded query has comparable size to that of XPIR. Furthermore, the cost to expand adds significant CPU time for SealPIR $d = 1$.

When the schemes use recursion ($d = 2$), both XPIR and SealPIR do not have the query-size drawback, but increase answer output size, by 8 to 10 times, relative to the $d = 1$ setting. Overall, FastPIR produces smaller responses (answer outputs) without large queries (XPIR with $d = 1$) or significant addition to computation time (SealPIR with $d = 1$).

Query-related overheads. Query generation time and query sizes are significantly larger in FastPIR than SealPIR (especially when the latter uses recursion). For instance, query size for 2^{15} items in SealPIR with $d = 2$ is 17 times smaller than the query size in FastPIR (with $d = 1$). However, FastPIR's query sizes are either smaller or comparable to those for XPIR, depending on recursion depth and message size.

Summary. If ANSWER is invoked frequently for an application with a library that has over several tens of thousands of messages, then FastPIR is a better fit. However, if the application cannot be designed such that its costs are dominated by those of ANSWER, then SealPIR or XPIR may be a better fit.

7 Related work

Onion-routing. Systems such as Tor [72], which are based on onion-routing [35, 65], can support anonymous VoIP calls

with low message latency. However, they do not provide strong guarantees. Indeed, a network adversary, such as an ISP, can learn call metadata via traffic analysis [16, 38, 44, 58, 62].

Mix-nets. Chaum introduced a mix-net: a network of nodes in which each node (called a mix) batches incoming messages and releases them in a permuted order [18]. A mix-net based system fundamentally requires at least one mix to be trusted [45–49, 51, 52, 64, 73, 76]. Yodel [49] is a state-of-the-art system based on mix-nets that specifically targets voice calls. Yodel scales to a few million users while providing a sub-second message latency. However, Yodel assumes that a fraction of the mixes it uses (80%) are not compromised. As one relaxes this assumption, say to make the fraction of trusted mixes to be 70% or lower, Yodel increases the latency between a caller and a callee.

DC-nets. Unlike a mix-net, a dining cryptographers network (DC-net) provides unconditional security using a technique that requires broadcasting of messages between network participants [17]. Due to the broadcasting requirement, earlier systems based on DC-nets scaled to only tens of participants [24, 34, 70]. Later systems [25, 77] improved scalability but at the cost of relaxing the threat model. For instance, Dissent in numbers [77] scales to 5000 clients with 600 ms latency for 600-client groups, but runs a DC-net among a (smaller) group of servers while assuming that one of them is trusted. PriFi [13] is the latest DC-net based system. It improves latency for a LAN setting of a small organization with a few hundred users (latency is 100 ms for 100 users). PriFi does not scale to thousands or tens of thousands of users. It also assumes that one of its servers in the group of servers is trusted.

Private mailboxes. Systems based on private mailboxes either obliviously write to [23, 32] or read from [7, 10, 14, 42, 68] mailboxes hosted over untrusted servers. The state-of-the-art system based on this strategy that works over completely untrusted infrastructure is Pung [7, 10] (rest of the systems assume non-colluding servers).

We empirically compared Addra to Pung, particularly to its scalable tree-based message retrieval scheme called BST (§6). Pung offers two other retrieval schemes: one called explicit retrieval and the other based on Bloom filters. The explicit scheme requires two round trips between a message recipient and the server, and incurs comparable server-side CPU overhead as the BST scheme. However, it is not viable in terms of network overhead as the server has to frequently broadcast a mapping comparable in size to the entire message library. For instance, for 32K users, the server pushes 625 MiB of mapping data every five minutes to every user, thus adding a bandwidth requirement of 16.6 Mbps per user. The Bloom filter scheme significantly lowers the network overhead relative to the explicit scheme. However, its overhead is still linear in the number of objects (so it is not a viable solution as the system scales up to hundreds of thousands of users). Besides, it works probabilistically: a message recipient is not guaranteed to download the message sent by the sender, thus degrading the quality of service by a non-zero amount.

Although Addra supports synchronous voice calls at scale, and Pung does not (§6.1), Addra does not replace Pung, which is designed for asynchronous applications such as email and chat. Indeed, Addra cannot retrieve long-lived messages from the server, which is a requirement for such applications.

Private information retrieval (PIR). Chor et al. [19, 20] introduced the problem of PIR over multiple non-colluding servers, while Kushilevitz and Ostrovsky [43] introduced CPIR over a single untrusted server. Since these decades old seminal works, there have been numerous improvements to concrete constructions of PIR. For instance, some schemes reduce PIR overheads [4, 7, 28, 29, 71], while others improve answer recovery against a byzantine server [31, 60]. In this paper, we introduced FastPIR, a new CPIR scheme that reduces the server-side computation overhead relative to the state-of-the-art CPIR schemes [4, 7] (§6.5).

8 Summary and future work

Metadata from voice calls contains rich information about people's lives, and is a prime target for powerful adversaries such as nation states. Prior work that hides metadata either requires trusted intermediaries or does not scale to more than tens of users for low-latency voice calls. This paper described Addra, the first system that hides metadata for voice calls over completely untrusted infrastructure for tens of thousands of users. Addra's current prototype supports 32,768 users on a cluster of 80 machines with a message latency of 726 ms and a voice synthesis rate of 1.6 Kbps. Addra provides its performance and privacy properties through a new, simple, and efficient protocol to access private mailboxes hosted on an untrusted server (§3), and a new private information retrieval (PIR) scheme, FastPIR (§4).

Our future work involves further scaling Addra from tens of thousands of users to hundreds of thousands or a few million users. To accelerate CPIR computation, a promising direction could be to explore efficient implementations of the master-worker architecture of Addra's server, as well as increased efficiency for the workers using GPUs and FPGAs. For the latter, one would have to address challenges related to running PIR on a heterogeneous system. Finally, a full-fledged Addra system would require extending its support from peer-to-peer voice calls to group calls.

Acknowledgments

We thank Sujaya Maiyya, Udit Paul, Nazmus Saquib, our shepherd Sebastian Angel, and the anonymous reviewers of OSDI 2021 for their feedback and insightful comments that helped improve this paper. This work is funded in part by NSF grants CNS-1703560 and CNS-1815733.

References

[1] Libscapi - the secure computation API. https://github.com/cryptobiu/libscapi.

[2] PALISADE homomorphic encryption software library. https://palisade-crypto.org/.

[3] rpclib - modern msgpack-rpc for C++. http://rpclib.net/.

[4] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. *Privacy Enhancing Technologies Symposium (PETS)*, 2016(2):155–174, 2016.

[5] I. Ahmad, Y. Yang, D. Agrawal, A. E. Abbadi, and T. Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure (extended version). Cryptology ePrint Archive, Report 2021/044, 2021. https://eprint.iacr.org/2021/044.

[6] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, November 2018.

[7] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[8] S. Angel, S. Kannan, and Z. Ratliff. Private resource allocators and their applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[9] S. Angel, D. Lazar, and I. Tzialla. What's a little leakage between friends? In *Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[10] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[11] S. G. Angel. *Unobservable communication over untrusted infrastructure*. PhD thesis, The University of Texas at Austin, 2018.

[12] J. Angwin, C. Savage, J. Larson, H. Moltke, L. Poitras, and J. Risen. AT&T helped US spy on Internet on a vast scale. *The New York Times*, 2015.

[13] L. Barman, I. Dacosta, M. Zamani, E. Zhai, A. Pyrgelis, B. Ford, J. Feigenbaum, and J.-P. Hubaux. PriFi: Low-latency anonymity for organizational networks. *Privacy Enhancing Technologies Symposium (PETS)*, 2020(4):24–47, 2020.

[14] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. *Privacy Enhancing Technologies Symposium (PETS)*, 2015(2):4–24, 2015.

[15] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology—CRYPTO*, 2012.

[16] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[17] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[18] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[19] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, 1995.

[20] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[21] D. Cole. We kill people based on metadata. *The New York Review*, May 2014.

[22] D. Cole. Is privacy obsolete? *The Nation*, Mar. 2015.

[23] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[24] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[25] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in Verdict. In *USENIX Security Symposium*, 2013.

[26] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[27] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3(1):1–5, 2013.

[28] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2014.

[29] C. Devet. Evaluating private information retrieval on the cloud. Technical report, University of Waterloo, 2013.

[30] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In *Privacy Enhancing Technologies Symposium (PETS)*, pages 63–82, 2014.

[31] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium*, pages 269–283, 2012.

[32] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security Symposium*, 2021.

[33] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[34] S. Goel, M. Robson, M. Polte, and E. Sirer. Herbivore:

A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003.

[35] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[36] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[37] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[38] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):1–28, 2010.

[39] S. Humphreys and M. De Zwart. Data retention, journalist freedoms and whistleblowers. *Media International Australia*, 165(1):103–116, 2017.

[40] ITU-T. G.114 : One-way transmission time. https://www.itu.int/rec/T-REC-G.114-200305-I/en, 2003.

[41] M. Kassim, R. A. Rahman, M. A. A. Aziz, A. Idris, and M. I. Yusof. Performance analysis of VoIP over 3G and 4G LTE network. In *2017 International Conference on Electrical, Electronics and System Engineering (ICEESE)*, pages 37–41. IEEE, 2017.

[42] L. Kissner, A. Oprea, M. K. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *International Conference on Applied Cryptography and Network Security (ACNS)*, 2004.

[43] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, 1997.

[44] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *USENIX Security Symposium*, 2015.

[45] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[46] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. *Privacy Enhancing Technologies Symposium (PETS)*, 2016(2):115–134, 2016.

[47] A. Kwon, D. Lu, and S. Devadas. XRD: Scalable messaging system with cryptographic privacy. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[48] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[49] D. Lazar, Y. Gilad, and N. Zeldovich. Yodel: Strong metadata security for voice calls. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[50] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[51] S. Le Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *ACM SIGCOMM Conference*, 2015.

[52] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.

[53] R. Lenzner. ATT, Verizon, Sprint are paid cash by NSA for your private communications. *Forbes*, 2013.

[54] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences*, 113(20):5536–5541, 2016.

[55] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: Inferring user profiles in online social networks. In *International conference on Web search and data mining*, 2010.

[56] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *USENIX Security Symposium*, 2011.

[57] Mozilla. LPCNet: Efficient neural speech synthesis. https://github.com/mozilla/LPCNet.

[58] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy (S&P)*, 2005.

[59] Office of the Director of National Intelligence. Statistical transparency report regarding the use of national security authorities. https://www.dni.gov/files/CLPT/documents/2020_ASTR_for_CY2019_FINAL.pdf, 2020.

[60] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *International Conference on Financial Cryptography and Data Security (FC)*, 2011.

[61] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999.

[62] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[63] A. Pfitzmann and M. Hansen. A terminol-

ogy for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. http://www.maroki.de/pub/dphistory/2010_Anon_Terminology_v0.34.pdf, 2010.

[64] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The Loopix anonymity system. In *USENIX Security Symposium*, 2017.

[65] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected areas in Communications*, 16(4):482–494, 1998.

[66] A. Rusbridger. The Snowden leaks and the public. *The New York Review*, Nov. 2013.

[67] D. Rushe. Yahoo $250,000 daily fine over NSA data refusal was set to double "every week". *The Guardian*, 2014.

[68] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: A secure method of pseudonymous mail retrieval. In *Workshop on Privacy in the Electronic Society (WPES)*, 2005.

[69] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Apr. 2020. Microsoft Research, Redmond, WA.

[70] E. G. Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: File sharing with strong anonymity. In *Proceedings of the ACM SIGOPS European workshop*, 2004.

[71] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 1998.

[72] P. Syverson, R. Dingledine, and N. Mathewson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[73] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–440, 2017.

[74] J.-M. Valin and J. Skoglund. LPCNet: Improving neural speech synthesis through linear prediction. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019.

[75] J.-M. Valin and J. Skoglund. A real-time wideband neural vocoder at 1.6 kb/s using LPCNet. *arXiv preprint arXiv:1903.12087*, 2019.

[76] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[77] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

Bringing Decentralized Search to Decentralized Services

Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan[†], Yubin Xia, Sebastian Angel[⋆], Haibo Chen

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Shanghai AI Laboratory

Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

[†]*Northeastern University* [⋆]*University of Pennsylvania*

Abstract

This paper addresses a key missing piece in the current ecosystem of decentralized services and blockchain apps: the lack of decentralized, verifiable, and private search. Existing decentralized systems like Steemit, OpenBazaar, and the growing number of blockchain apps provide alternatives to existing services. And yet, they continue to rely on centralized search engines and indexers to help users access the content they seek and navigate the apps. Such centralized engines are in a perfect position to censor content and violate users' privacy, undermining some of the key tenets behind decentralization.

To remedy this, we introduce DESEARCH, the first decentralized search engine that guarantees the integrity and privacy of search results for decentralized services and blockchain apps. DESEARCH uses trusted hardware to build a network of workers that execute a pipeline of small search engine tasks (crawl, index, aggregate, rank, query). DESEARCH then introduces a *witness* mechanism to make sure the completed tasks can be reused across different pipelines, and to make the final search results verifiable by end users. We implement DESEARCH for two existing decentralized services that handle over 80 million records and 240 GBs of data, and show that DESEARCH can scale horizontally with the number of workers and can process 128 million search queries per day.

1 Introduction

Most of today's online services—including search, social networks, and e-commerce—are centralized for reasons such as economies of scale, compatible monetization strategies, network effects, legal requirements, and technical limitations. Yet, since the birth of the Internet, there have been periods of intense interest in decentralization, including the peer-to-peer systems bonanza of the early and mid 2000s [57, 90, 107] and the current blockchain boom [39, 94, 113]. A rich set of decentralized services have appeared and are able to offer most of the functionalities that common centralized online services provide, as listed in Figure 1. Proponents of decentralization argue that centralized services often employ anti-consumer practices owing to their monopolistic positions [18, 27], and the mismatch between users' expectations and operators' incentives [44]. Further, centralized services are particularly susceptible to censorship [4, 41] (either self-imposed or coerced through technical or legal means) and collect vast amounts of user information [13, 14].

Service	Centralized	Decentralized
Currency	U.S. Dollars	Bitcoin [94]
Online Marketplace	eBay	OpenBazaar [28]
Social Media	Twitter	Steemit [40]
Video Sharing	Youtube	DTube [8]
Social Network	Facebook	Mastodon [16]
Public Storage	DropBox	IPFS [59]
Messaging	Slack	Matrix [25]
Video Conference	Zoom	Zipcall [50]
Website Hosting	WiX [47]	ZeroNet [49]
Financial Betting	Etoro [12]	Augur [1]
Supercomputing	Titan [45]	Golem [17]
Document Collaboration	Google Docs	Graphite [21]

FIGURE 1—Centralized services and decentralized alternatives.

While the idea of building fully decentralized services is alluring, developers must currently make a significant compromise: they must defer search functionality to a centralized service. For example, OpenBazaar [28] makes a strong case for a decentralized marketplace, but users must use a centralized search engine such as Bazaar Dog [46] or Duo Search [9] to discover what items are for sale in the first place. A similar compromise is made by other popular services [8, 16, 28, 40, 49]. This state of affairs is problematic because search is not an optional feature but rather a core component of these systems. Without decentralized search, the purported goals of anti-censorship is hard to attain: the search engine could trivially track users' queries, and opaquely censor particular content [3, 4, 7, 32, 41]. For example, Steemit [40] is a decentralized social media service where posts are stored on the public Steem blockchain [39], but Steemit developers have been known to prevent users' posts from appearing on the front end site [41].

Prior proposals. Several search engines [29, 81] propose reaching consensus amongst replicas to ensure the correctness of search indexes. However, these engines rely on a central website hosted at the third party to answer queries. As a result, an end-user who visits this website has no way to validate the integrity of the displayed results, or to determine whether there are missing entries. As an alternative, peer-to-peer-based search engines [24, 48] allow shared indexes between peers and queries can be issued to any peer (essentially implementing a distributed hash table). However, these engines do not support verifiable indexes, and allow peers to monitor clients'

requests, leading to severe privacy concerns. Finally, many blockchains encourage users to run their own indexer node or to use a third-party indexer [30, 35] to access the content in the blockchain. However, most users lack the resources and the expertise needed to deploy their own indexers, and third-party indexers must be trusted to not censor content or violate users' privacy.

Goals and contribution. Building a decentralized search engine that avoids the aforementioned shortcomings is far from trivial. First, the search engine should be able to authenticate the data source to make sure the dataset contains all data items and is free from forgeries. Second, the user's intention, including the query keywords and search results, should be kept private from any party. Third, the search engine should be able to provide a proof of execution to clients that explains how the search results were generated, and why the results are correct. Last, the search engine should have reasonable costs and scale to support many users.

To meet these goals, we introduce DESEARCH, the first decentralized search engine that allows users to verify the correctness of their search results, and preserves the privacy of user queries. DESEARCH outsources fragments of search tasks such as crawling, indexing, aggregation, ranking, and query processing to trusted execution environments (TEEs) running on untrusted *executors* that compose a decentralized network, and introduces new data structures and mechanisms that make executors' operations reusable and externally verifiable.

First, since each executor only has a local view of the computation, DESEARCH uses *witnesses*, which are a type of object that reflects the dataflow and establishes the correctness of results. A witness ensures that executors cannot lie about which sources they crawled, how they aggregated data, computed the index, or responded to a query. Verifying witnesses is not cheap, so DESEARCH amortizes the verification cost by reusing previously checked witnesses across queries, using designated executors—verifiers—to check witnesses on behalf of clients.

Second, DESEARCH uses a public storage service called *Kanban*. Kanban allows executors in the network to exchange intermediate information, agree on a snapshot of data in the system, manage membership, tolerate faults, and verify results. To detect rollback on Kanban data, DESEARCH summarizes an epoch-based snapshot and stores it on an append-only distributed log.

Finally, DESEARCH protects the privacy of queries with two techniques. To prevent leaks from access patterns, DESEARCH adapts an existing oblivious RAM library [102]. To resist volume side channels, DESEARCH returns the same amount of data for all search queries. It does so by equalizing the lengths of result entries. This approach does not reduce the performance or quality of the service because search engines need not display all of the content but rather a small snippet. As an analogy in the Web context, search engines like Google do not display the entirety of a Web site's content in the search result; instead, they typically display the URL of each site and a small text snippet.

We built a prototype of DESEARCH in $2,600$ lines of code, and have adapted it to work with Steemit (a decentralized social media service), and OpenBazaar (a decentralized e-commerce service). Our evaluation of DESEARCH on 1312 virtual machines across the wide-area network with variable network latency and executor failures shows that DESEARCH scales well as executors join the network, and can handle over 128 million requests per day. Checking the correctness of the displayed results is also affordable: users can verify results in under 1.2 seconds by consulting dedicated verifiers.

To summarize, the contributions of this paper are:

- The design of DESEARCH, the first decentralized search engine that allows any executors with a TEE to join and provide search functionality for decentralized services.
- A witness mechanism that organizes verifiable proofs from short-lived executors to form a global dataflow graph. Through these witnesses, DESEARCH offers fast verification for search queries.
- A prototype of DESEARCH built for Steemit [40] and OpenBazaar [28], and an evaluation of DESEARCH's performance and scalability.

While DESEARCH enables, for the first time, scalable, verifiable, and private search for existing decentralized services, it is not a viable replacement for traditional Web search engines (e.g., Google). Besides the obvious issue of scale, DESEARCH's target applications expose a single source of data (their underlying distributed log or proof of storage mechanism), which gives DESEARCH an anchor for its witness data structure. In contrast, the Web has no such single log, which would prevent DESEARCH from proving that all Web pages had been crawled and indexed. Nevertheless, we believe DESEARCH fills a crucial void.

The rest of the paper is organized as follows. Section 2 describes the motivation, the problem, and the threat model of DESEARCH, and discusses potential solutions. Section 3 provides an overview of DESEARCH, highlights DESEARCH's components, and explains how they work cooperatively. Section 4 presents how DESEARCH achieves a decentralized yet verifiable search; Section 5 introduces Kanban, a verifiable storage; Section 6 describes how DESEARCH provides oblivious search. Section 7 gives the implementation details. Section 8 evaluates DESEARCH in a local heterogeneous cluster and a geo-distributed environment. Finally, Section 9 discusses other aspects and Section 10 compares related work.

2 Motivation and problem statement

This section describes our motivation, target setting and threat model, and potential solutions that fall short.

2.1 Motivation

We give a brief overview of two representative decentralized services and the problems related to search that might arise in those systems.

Social Media. Steemit [40] is a social media platform that stores user-generated contents on the public Steem blockchain [39]. Although the raw data from the blockchain is tamper-proof, Steemit's front-end servers can censor or manipulate the search results before delivering them to users [41]. One can think of these servers as centralized curators for decentralized storage. These servers also know who searches for what, which may reveal users' interests and preferences.

Marketplaces. OpenBazaar [28] is an e-commerce marketplace built on top of a peer-to-peer network and storage. To help users search for items, OpenBazaar provides an API [5] for third-party search engines to crawl and index items. But existing search engines [2, 9, 34, 46] are opaque; they can bias results towards item listings that benefit them financially. For example, a listing owner could pay the search engine to promote its listing or hide other listings. Additionally, these engines can learn users' purchasing habits (and other information) from keywords and search histories.

In short, existing decentralized services currently lack trusted search that offers integrity and privacy. A decentralized search engine should have strict requirements on the visibility of the search queries and the correctness of the search results. Below, we formalize these goals.

2.2 Decentralized search

Consider a decentralized system where volunteers called *executors* together operate a search engine. *Users* want to search over some *source data* (e.g., data stored in a blockchain) using some *search algorithm*. Both the source data and the search algorithm are public and accessible to all users. For each search, a user sends *keywords* to the executors running the search algorithm and expects a *search result*—a ranked list of summaries for entries in the source data alongside pointers to the corresponding full entries.

After receiving the search results, users want to verify that the results are actually correct—that they are derived from executing the search algorithm on the latest source data and the provided keywords. Users also want to keep their searches private. In detail, the challenge is to design a decentralized search engine that meets these goals:

- **Integrity:** the search results should correspond to the correct execution of the published search algorithm on the most recent source data. We divide this into two properties: (1) *Execution integrity*, meaning that the search algorithm is faithfully executed on *some* source data and the output search results are ranked in the correct order. (2) *Data integrity*, meaning that the source data used is legitimate, up-to-date, and there are no missing or forged data records. In combination, these two properties prevent (undetectable) biased results and censorship.
- **Privacy:** the search engine (or any third party) should not leak the user's search query or the corresponding search results. There are two aspects to this. (1) *Content*: the user's query (search keywords) and the corresponding search results are never available in the clear to the search engine. (2) *Metadata*: the number and size of the messages exchanged by the user and the search engine is independent of the search results and the provided keywords.
- **Scalability:** executors that join the search engine should contribute meaningfully towards its capacity: more executors should result in higher search throughput.

Search Engine	Integrity		Privacy		Scalability
	Execution	Data	Content	Metadata	
YaCy [48]	○	○	○	○	●
Presearch [29]	●	●	○	○	●
The Graph [20]	●	●	○	○	●
IPSE [24]	●	○	○	○	●
BITE [89]	●	●	●	●	○
Rearguard [108]	●	○	●	◐	○
Oblix [91]	●	○	●	●	○
vChain [114]	●	●	○	○	○
GEM^2-Tree [117]	●	●	○	○	●
X-Search [92]	○	○	◐	○	○
CYCLOSA [98]	○	○	○	○	●
DESEARCH	●	●	●	●	●

FIGURE 2—Comparison between prior work and DESEARCH. X-Search [92] obfuscates query keywords but does not hide them. Rearguard [108] hides the index size but not the result size.

Figure 2 shows existing decentralized or private search engines. None of them meets all of our desired goals.

2.3 Potential approaches

How to provide integrity and/or privacy for an execution (for example, search) has been studied broadly. We list some approaches below (see more in Section 10).

Replication such as PBFT [67] and Ethereum [113] is one approach to build systems that can guarantee execution integrity within a given number of (Byzantine) faults. However, it requires performing the computation multiple times and traditional replication protocols do not provide privacy.

Another line of work [99, 105, 111] uses cryptography—such as fully homomorphic encryption (FHE) [74], secure multi-party computation (MPC) [116], and verifiable computation (VC) [62, 97, 105]—to provide execution integrity and/or privacy. Though promising, it remains an open problem to build a system that can support a complex enough search model and the large-scale datasets used by today's decentralized services.

Trusted execution environments (TEE) provide another approach to build systems that protect a sensitive execution from being tampered with or eavesdropped. However, existing TEEs either (1) have limited private memory (128 MB

per node); (2) lack memory encryption [55] which makes them vulnerable to physical attacks that are realistic in a decentralized setting; or (3) are susceptible to memory tampering [54, 84]. Although newly proposed TEEs [43] have expanded enclave memory to 1 TB, they relax the security guarantees (e.g., they are vulnerable to physical replay attacks) owing to the loss of integrity tree protection. Prior distributed frameworks like VC3 [104] and Ryoan [77] address many of the above shortcomings but are not designed for a decentralized environment. Their predefined computation graph is not a good fit for a dynamic environment where the source data and the set of executors is constantly in flux.

Finally, systems like Dory [71] that implement private search for a file system allow users to get pointers to objects that match certain keywords, but they do not implement features expected from a search engine such as finding non-exact matches, ranking results, or providing summaries.

DESEARCH uses a combination of new and existing techniques—TEEs, an append-only log, an epoch-based storage service, authenticated data structures (hash trees), and oblivious RAM—to address the challenges in Section 2.2.

2.4 Threat model

DESEARCH assumes a decentralized network where untrusted executors are operated by unknown parties, but are equipped with TEEs. We assume reliable TEEs with no microarchitectural or analog side channels [64, 110] or vulnerabilities to voltage changes or physical tampering [93]. We also assume the TEE manufacturers do not inject backdoors into TEEs, or share their private keys. Finally, we assume that the TEE remote attestation mechanism works as intended. While these assumptions are undeniably strong given existing TEE's track record, we expect this technology to mature over the years and for many of the current weaknesses to be addressed.

In DESEARCH, executors join the system and volunteer their TEEs (or they are paid or incentivized to do so, which is orthogonal). Executors can be malicious: they can deny services, modify the inputs that go into the TEE or the outputs that come out of it. They can also corrupt data outside the TEE's protected memory, or replay inputs and outputs. Moreover, executors observe memory accesses, and I/O patterns [65, 115] (either which memory is accessed or how much data/how many times).

DESEARCH requires that the data over which the decentralized service will offer search be stored in a publicly auditable source. This requirement boils down to ensuring that the data source has its own mechanism to check that the data added is not removed or tampered with (e.g., storing the data in a blockchain or IPFS [59]).

3 System overview

In this section we give the design principles of DESEARCH, which is a decentralized search engine for decentralized services and blockchain apps. We start by highlighting some of the challenges present when building a decentralized search engine that meets our requirements (§2.2). Some of these challenges stem from our decentralized environment, while others come from the limitations of today's TEEs and the dynamic nature of search.

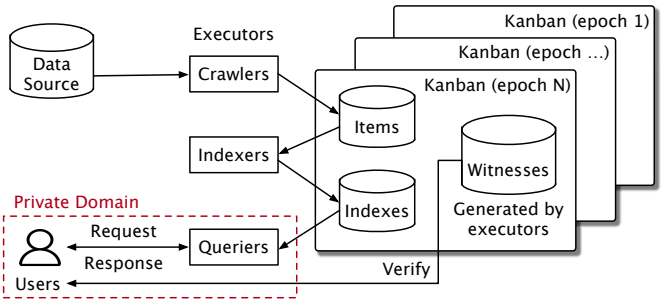


FIGURE 3—DESEARCH's architecture. DESEARCH obtains raw data from public decentralized services (e.g., Steem blockchain) as the data sources, and stores the intermediate data (i.e., items and indexes) on Kanban, a public append-only storage that creates snapshots periodically. DESEARCH executors generate witnesses along with the search pipeline. Privacy (§2.2) is offered for users in the query phase (within the dashed rectangle).

First, decentralization requires that executors be allowed to freely join and depart the system (§2.2), which means that executors can go offline unexpectedly. Thus, a standard search engine design with long-running tasks and stateful components is unfavorable, as one executor's leaving can heavily impact the service.

Second, today's TEE instances (DESEARCH uses Intel SGX) have limited memory (128MB or 256MB); working sets in excess of this limit require expensive paging mechanisms. Indeed, our experiments reveal that the latency incurred by paging far exceeds what is acceptable for a user-facing service like search. For example, when we run a search service for Steemit, which requires a 31 GB index, in a single optimized SGX instance, it takes 16 (resp., 65) seconds to respond to a single-keyword (resp., two-keyword) query. Recent work [95, 96] has explored ways to enlarge the trusted memory through software-based swapping (either via a managed runtime or compiler instrumentation), but those solutions leak the application's memory access pattern. As a result, we find that to achieve acceptable latency, it is necessary for the search engine's functionality to be split into small tasks that are processed by many SGX instances in parallel.

Third, search services are dynamic, and it is hard to track and verify the whole search process from crawling to query servicing. In particular, a search engine is unable to plan a computation graph (like in big-data [73, 104, 111] or machine-learning [79, 87] systems) as the arrival of new source data or user search queries is unpredictable, and the set of available executors is unknown a priori.

Overview. DESEARCH addresses these challenges by decomposing a search into a pipeline of short-lived tasks where

each executor is responsible for a single task at a time and operates only on a small portion of data. Executors are stateless. They fetch inputs and write outputs from and to a storage service named *Kanban*. Kanban is a cloud-based key-value store that provides high-availability and data integrity. We describe Kanban in detail in Section 5.

To track the completion of the dynamically executing tasks within the search pipeline, DESEARCH uses *witnesses* (§4), which are cryptographic summaries that capture the correct transfer of data among executors. A witness is also a proof of an executor's behavior, which allows users to verify their search results *ex post facto*.

Figure 3 shows the architecture of DESEARCH.

Search pipeline and Kanban. In DESEARCH, executors are categorized into four roles: crawlers, indexers, queriers, and masters (for simplicity, masters are omitted in Figure 3). The first three roles comprise a full search pipeline—*crawlers* fetch data from public sources (for example a blockchain [39, 94, 113] or P2P storage [28, 59]); *indexers* construct inverted indexes; *queriers* process queries, rank and build search results.

Instead of point-to-point communication, executors in the pipeline communicate through Kanban. Kanban also stores the data (items, indexes, and witnesses) generated by executors, and provides data integrity (but not confidentiality) by periodically creating snapshots of all current state and committing a digest of the snapshot to a public log (e.g., Ethereum [113] or Steem [39]). We call the time between two consecutive snapshots an *epoch*; we call the data corresponding to the beginning of an epoch, an *epoch snapshot*.

For privacy (§2.2), DESEARCH comprises public and private domains. Executors in the public domain access public data (like public source data) and produce shared information (like indexes). On the other hand, users' interactions with DESEARCH happen in the private domain, and their communication (for example, search requests and responses) are encrypted and kept secret.

Masters. *Masters* are the executors that provide crucial membership services: (1) a service that authenticates a TEE node who wants to join DESEARCH; (2) a job assignment service that coordinates the independent executors to form the search pipeline with minimal repeated work; (3) a key management service (KMS) that allows anyone to identify if an executor is a legitimate DESEARCH member. Regarding managing the KMS and task coordination, masters periodically (in the beginning of an epoch), release a list of public keys from legitimate executors on Kanban so that users can verify their signatures and communicate with them. This list includes the active nodes to ensure the service's availability. Masters hold the *root key* that serves as the identity of the DESEARCH system, allowing the public to recognize DESEARCH. We describe how the system is bootstrapped, how new masters join, and how the root key is generated in Section 9.

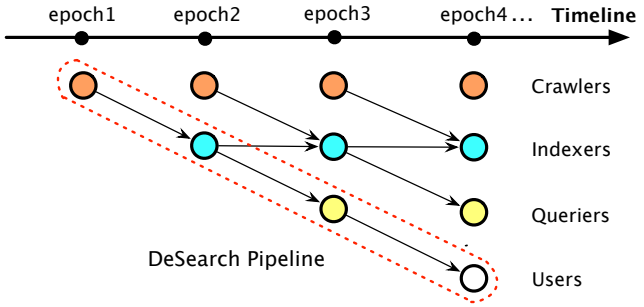


FIGURE 4—In DESEARCH, executors and users use data from the last epoch as inputs, which we call the *offset-by-one* strategy. For example, an indexer uses items of the last epoch to generate new indexes (denoted by oblique arrows), and may merge indexes from the prior epoch (denoted by horizontal arrows).

Workflow. DESEARCH's executors perform an ordinary search pipeline—crawling, indexing, and serving queries. In DESEARCH, data integrity (§2.2) is defined with respect to a particular epoch snapshot; DESEARCH uses an *offset-by-one* strategy where an executor always uses data from the last epoch in order to ensure that the flow of data is verifiable when expressed as a pipeline of tasks. Figure 4 shows an example of this process. Along every step of the pipeline, each executor generates a *witness*, a proof of what the executor has seen, has done, and how the data has been transferred. We discuss witnesses in Section 4.

To conduct a search, a user first retrieves a list of active queriers. This list is maintained by both masters and queriers: queriers update their status on the list with signed proofs specifying that they have seen the most-recent epoch; the legitimacy of the status proofs is verified by masters. Users know this by checking that the list is signed by masters.

With the active querier list, the user randomly selects one as the leader, and sends (encrypted) search keywords to the leader. The leader then seeks more peer queriers to collectively handle the request. That is, different queriers have different portions of indexes and together serve one user search request. The leader finally aggregates results by ranking based on relevance, and returns to the user a list of the k most relevant items. One item comprises a link (to the original content) and a content snippet that summarizes the item and often contains the searched keywords.

Together with the search results, the user also receives witnesses from queriers. The witnesses produced by the search pipeline (all of them) form a witness tree, which users can verify by starting from the witnesses received from the leader querier, traversing the tree, checking every node (witness), and confirming that the search has been correctly executed. We discuss this verification process next.

4 Verifiable search

In DESEARCH, search functions are outsourced to independent executors in a decentralized environment, and data is

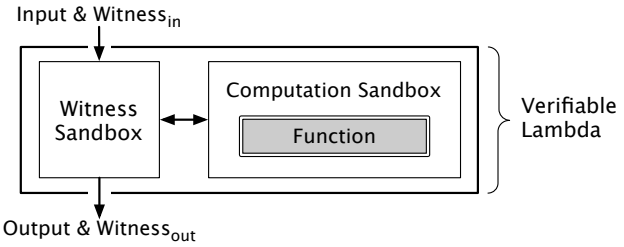


FIGURE 5—A verifiable lambda is composed of a witness sandbox and a computation sandbox. The function is one of: crawling, indexing, and querying. The witness$_{in}$ contains the hash for inputs and is from prior lambdas. The witness$_{out}$ is the witness generated by this lambda. The sandbox design is necessary because it isolates the witness processing from the function execution; the buggy or malicious function cannot tamper with the integrity of witness.

partitioned across executors. As a result, an executor only has a local view of its own computation and cannot protect the entire search pipeline even though it runs on a TEE assumed to be correct. Specifically, guaranteeing integrity (§2.2) has two parts: (a) ensuring that results are generated by the correct algorithm (*execution integrity*) despite intermediate data being transferred among executors through untrusted channels; and (b) ensuring that results are derived from the desired dataset without the absence of data records or the inclusion of bogus data (*data integrity*).

DESEARCH uses *verifiable lambdas* and *witnesses* to address challenge (a), and *Kanban* with epochs for challenge (b). We detail them below.

Verifiable lambda and witnesses. As stated earlier (§3), DESEARCH splits a search pipeline into small tasks. Each task is executed by a basic unit, called *verifiable lambda*. The concept of a verifiable lambda (short as lambda) is borrowed from serverless computing [36] and SGX sandboxing systems [77, 104]. A key difference is that DESEARCH's lambda requires a TEE enclave abstraction that yields a *witness* (we discuss it briefly, but it is a type of certificate of correct execution) after every computation, allowing the intermediate data to be verified and reused.

A lambda is composed of the two sandboxes shown in Figure 5: (1) a witness sandbox that validates the input upon loading, and generates a witness before delivering the result to the next lambda; (2) a computation sandbox which runs the main function in a self-contained execution environment that does not use any external services; the goal is to resist Iago attacks [68] (attacks in which a malicious OS causes the process to harm itself). All I/O activity of the computation sandbox must be routed to the witness sandbox, which logs the I/O data and produces a auditable record stored within the witness. This design isolates the buggy or malicious function execution from witness processing, so that the integrity of the witness is easier to reason about. For multiple inputs, a lambda can batch them to avoid generating many witnesses.

For a chain of lambdas, to ensure correct results, a seemingly straightforward solution is to encode a signed nonce in each execution of the pipeline. However, this approach does not work for search because a search pipeline often requires the data (such as crawling data and indexes) from multiple previous pipeline executions, and a signed nonce fails to capture the relationship between these multiple inputs and the output (critical for verifying data integrity such as data completeness). Hence, we introduce a certificate called *witness* for each lambda.

A lambda's witness is a tuple:

$$\Big\langle \Big[H(in1), H(in2), \cdots\Big],\ H(func),\ H(out) \Big\rangle_{signed}$$

that mirrors how an *output* is generated by performing a *function* over a list of *inputs*. The $H(in)$ and $H(out)$ are hashes of the input and output blobs, and $H(func)$ is the hash of the program binary that runs in this lambda. A witness is signed by the lambda, and anyone can verify the signature using DESEARCH's KMS (§3). A witness has a feature called "multiple input, single output" (MISO), which consumes multiple inputs and produces exactly one output. We find that search is a natural fit for MISO, as mutilple on-chain items yield an index, and multiple indexes serve a query.

Generally speaking, witnesses can be thought of as a *proof-carrying metadata* for a decentralized system, which explains how an output is being generated and with which piece of code, and enables a user to examine the computation process from the effect to the cause for a whole-pipeline verification. The notion of verifiable lambda and its witness mechanism are general enough to support other scenarios such as a decentralized and verifiable recommender system (§9).

Witness-based verification. All witnesses from a search process form a tree, which we call a *witness tree* (see an example in Figure 6). Users can verify their search results by traversing and checking the corresponding trees, the roots of which are the witnesses that users receive from queriers.

To check a single witness, a user first verifies whether the witness is signed by a legitimate executor and then checks if the hash of the executed function is as expected. This approach, combined with the integrity guarantees of the underlying TEE, ensures that the lambda which produced the witness faithfully executed the desired function.

Now consider the data transition between two adjacent lambdas in a search pipeline. The former lambda commits its output and a signed witness to Kanban; the latter lambda fetches one (or multiple) pair of data and witnesses from Kanban, checks their signatures, validates if the hash in the witness matches the data, and feeds the data to the computation function. A user can verify that the inputs of a latter lambda are indeed the outputs from a former lambda by checking whether $H(in)$ of the latter equals $H(out)$ from the former.

Finally, users check if data sources are genuine by checking whether $H(in)$ in the beginning (the crawling phase) is indeed a correct summary of the original data source. Users need

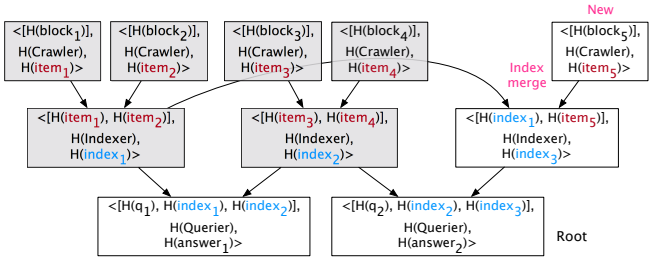


FIGURE 6—Two example witness trees. A rectangle represents a witness, edges represent data dependencies, and "H(...)" indicates hashes. This figure contains two search queries ("q_1" and "q_2"): q_1 happens first, and there is a merge update to the indexes ($index_3$ derives from $index_1$ and $item_5$), then q_2 happens. As a result, q_1 and q_2 share a subtree indicated in gray color.

to download the contents from the data source and calculate their hashes, which is a very expensive procedure (we address this issue shortly). If all the above checks pass, users confirm that their search results are faithfully produced because all steps—crawling from the data source, the intermediate data transferring between tasks, and each task in the search pipeline—are verified to be authentic and faithfully executed.

Providing efficient verification. The aforementioned verification process works in principle, but in practice, a performance challenge arises. To verify a search, a user would have to download all the witnesses, check all the signatures and hashes, and examine the data source. To lighten the burden of verification on the user side, DESEARCH uses *delegated verification*: users offload parts of their verifications to some executors dedicated for verification, which we call *verifiers*.

Beyond simplifying user-side verification, delegated verification also saves work by batching and deduplicating verifications from different users. This is based on an observation that serving different search requests uses a lot of shared indexes, hence the witnesses from the shared portion can be reused (as an example, see the gray subtree in Figure 6). Because of delegated verification, verifiers have the opportunity to batch many common witnesses, which they verify once for all. Finally, users only have to verify the final step—the querier phase's witness, significantly accelerating the verification (see delegated verification's speedup in §8.2). Because verifiers are only delegated to accelerate the verification of the witnesses in the public domain, it does not leak any information about particular users or their queries.

Data integrity. The above verification ensures that functions are executed as expected, but there is no guarantee that these functions see all data even if all functions are protected by TEEs. In fact, there is no definition of "all data" from a user's perspective because newly generated data takes time to be reflected in the search results. It is therefore unspecified what data must appear in any particular search result.

To define *data completeness* (a part of data integrity, §2.2) for searches, DESEARCH divides time into *epochs*, and executors write data to Kanban annotated with the current epoch (we elaborate on epochs in §5). We define a search that uses a complete set of data if each step (represented by witnesses) in a search pipeline (represented by a witness tree) uses *all* inputs in Kanban before the step's epoch, and each input is from the *most recent* epoch available. For example, a querier's task in epoch i satisfies our condition if the querier loads all indexes generated before epoch i, and the loaded portions of the indexes are from their latest version before epoch i.

To check the data integrity of a witness, verifiers first recognize the epoch when the witness was generated, then load the snapshot of Kanban immediately before that epoch (§5), and finally verify if the executor used all the up-to-date inputs. In practice, verifiers need not load the data in the snapshot; they load the metadata including data ids and their hashes.

5 Kanban

As mentioned in Section 4, Kanban is a storage system that provides high availability and data integrity. Kanban is hosted in public clouds for availability, but as a decentralized system, DESEARCH does not trust these cloud providers for correctness. Instead, DESEARCH creates snapshots of Kanban periodically for each epoch, called *epoch snapshots*, and commits their digests (hashes) to a public distributed log. This approach provides *epoch-based data integrity*: DESEARCH guarantees the integrity of all of the data included in a committed epoch. Of course, an alternative—using SGX-based cloud storage [100] for Kanban—works in principle, but current SGX cannot scale to a large trusted memory with integrity support (§3).

The rest of this section will introduce Kanban's usage and guarantees, and then discuss how DESEARCH uses Kanban as a storage and coordination service.

Kanban overview. Kanban serves two main purposes: storing data for the search pipeline (including items, indexes, and their witnesses), and enabling executors to communicate and coordinate their tasks. Kanban exposes APIs for each service.

As a data storage, Kanban exposes key-value-like APIs with `put(key, val)` and `get(key)`. Keys are constructed by the data types, epoch numbers, and chunk numbers. For example, "INDEX-#1000-v3" represents the 3rd chunk of the index for epoch 1000. Witnesses are also stored in the data storage, using the output hash $H(out)$ as the key. Thanks to the MISO feature of witnesses (§4), anyone can download a particular value from Kanban via `get()`, calculate its hash, and use this hash as the key to retrieve the corresponding witness in order to understand how the data was generated.

For communication, Kanban provides a mailbox for every executor with `send(mailbox, msg)` and `recv(mailbox)`, using the executor's public key as the mailbox address. Invoking `send()` allows an executor to submit a message to a specified mailbox, and `recv()` to download messages. All messages are encrypted using the mailbox owner's public key,

which can be obtained from the KMS (§3). Consequently, only owners can read message contents. For example, masters and queriers use the mailbox to negotiate the active querier list; this negotiation is encrypted in case Kanban maliciously attempts to forge or block particular executors.

Note that both storage and communication APIs are wrappers of the canonical key-value APIs, and Kanban can easily adapt to different underlying (cloud) storage systems. Our Kanban implementation uses Redis, a popular key-value store, as the underlying storage (see §7).

Epoch-based data integrity. Kanban requires executors to sign their submitted data (using Ed25519) to prevent data tampering or forgeries. Still, the underlying storage can *equivocate* and show different views of the data to different executors by omitting, rolling back, or forking the data. Detecting such divergences often requires clients (executors in our context) to synchronize out-of-band [83, 86], which is too expensive in a decentralized environment.

DESEARCH uses a loose synchronization approach: masters periodically synchronize Kanban's states with other executors. This loose synchronization works because of two observations. First, search engines are not supposed to instantly reflect newly generated data in search results because crawling and indexing takes time; as a (admittedly apples-to-oranges) comparison, Google crawls a site every 3 days or even longer [6]. Second, most of the tasks in the search pipeline are idempotent, so it is acceptable if two executors end up working on the same task. For example, it is safe for two crawlers to crawl the same data source, or two indexers to generate indexes for the same items, as the results are the same. Duplicate work sacrifices efficiency but not correctness.

To synchronize states with other executors, masters periodically create epoch snapshots of Kanban's data (excluding the data in the mailboxes which is used for coordination and is ephemeral), summarizes the snapshot as a digest, and commits the digest to a public append-only log (DESEARCH's implementation uses an EOS blockchain [11]). After the log accepts the digest, a new epoch is committed and is visible to all executors (assuming the public log is available).

DESEARCH guarantees *epoch-based data integrity*: for a committed epoch, all data included in this epoch is immutable and must be visible to all executors; otherwise, verification will fail. To see how DESEARCH guarantees this, if Kanban hides data from or returns stale data to an executor, the data integrity checks (§4) of this executor's witness will fail. This is because verifiers know the epoch of the witness (say epoch i) and the data this executor should have read (data in epoch $i-1$). If the witness missed any data or read some stale version, the verifier rejects. Before using one epoch for checking the data integrity, verifiers must ensure that the data (represented by their ids and hashes) in one epoch is consistent with the digest on the log. The verifier fetches all the data ids and hashes in one epoch, calculates their digest, and compares it with the digest on the public log.

Task coordination by epochs. DESEARCH's pipeline is coordinated through Kanban, which is based on epochs. An epoch is 15 minutes by default. Executors learn the current epoch number by querying the public log. As mentioned earlier, executors follow an offset-by-one strategy where they read data from the last committed epoch rather than the current epoch (see Figure 4). This guarantees that the DESEARCH pipeline only uses data that is already authenticated by the epoch-based digests on the log.

Our current implementation uses masters to assign jobs for crawlers and indexers. Masters also take charge of data collection in each epoch—they decide what data to include in the current epoch. DESEARCH requires other executors to put the current/correct epoch number in their outputs. If an executor in epoch i fails to do so, for example, it disregards the epoch or fails to submit its outputs on time (before masters commit epoch i), the data is discarded by the masters and the work is wasted. But this waste is acceptable as tasks are small.

Supporting multiple clouds. Though our current implementation only uses one cloud as Kanban's underlying storage, we plan to extend Kanban with multiple clouds for better availability [60], and more importantly, to lower the risk of vendor lockdown. With multiple clouds, executors write to all clouds and read from any one of them. Master executors are obligated to synchronize different clouds.

Data synchronization among clouds is challenging, which often requires running an expensive consensus protocol. However, by Kanban's epoch design, DESEARCH is able to do synchronization infrequently, only when committing an epoch. And if data diverges between clouds (note that data cannot be forged, due to signatures), master executors are in charge of merging the data. The key takeaway is that DESEARCH (or rather a search service) can tolerate infrequent synchronizations, so masters have plenty of leeway to orchestrate an epoch on which all clouds agree.

6 Oblivious search

DESEARCH guarantees integrity (§2.2) by leveraging witnesses and SGX. One might hope that SGX would also provide privacy for searches, as SGX supports confidential computing [26] and we assume that SGX works as designed (§2.4). However, DESEARCH's design in Section 4 leaks information: an adversary can learn users' keywords without breaking any of the guarantees of SGX. Below, we discuss concrete privacy violations, and then show how DESEARCH addresses these violations with ORAM and equalizing message lengths.

Privacy violations. To start a search, a user initiates an encrypted session (via TLS) to a querier selected from the active list of queriers published on Kanban by masters. Although the messages are encrypted/authenticated and computations are confidential (offered by SGX), adversaries can still conduct

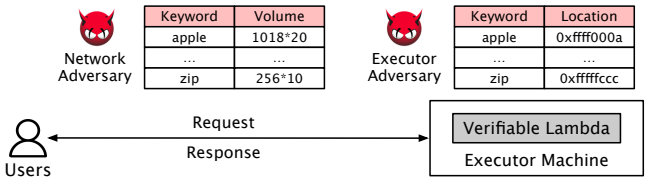


FIGURE 7—A search faces two privacy challenges: a network adversary can learn keyword information by monitoring request/response volumes, and an executor adversary can infer keywords by observing memory accesses.

two types of attacks (see examples in Figure 7).

First, an *executor adversary* that runs queriers can observe memory access patterns (both EPC and DRAM) to infer user keywords. Specifically, the adversary issues search requests with all possible keywords to the querier it hosts; by observing memory accesses, it can construct a dictionary that maps a keyword to a memory location [115]. Consequently, when a real user sends a query, the adversary can infer the user's keywords by observing which memory locations are accessed by looking them up in the dictionary.

Second, a *network adversary* can eavesdrop on the communication between users and queriers, and among queriers (which occurs when collaboratively serving one request). By monitoring the number of packets and their sizes, the adversary can learn information about keywords [66] because candidate lists for different keywords have different lengths, and returned items (e.g., a post on Steemit) also vary in size. Similar to an executor adversary, a network adversary can construct a dictionary that maps keywords to response lengths.

DESEARCH + ORAM. To prevent attacks from the executor adversaries, DESEARCH, like much prior work [89, 102], uses Oblivious RAM (ORAM) to hide memory access patterns. In particular, DESEARCH uses Circuit-ORAM [112] as follows: DESEARCH creates a key-value store where the keys are search keywords and values are lists of item ids. ORAM then guarantees that an executor adversary cannot learn anything about which object in the key-value store is being accessed (that is, which search keywords or item ids) from the memory accesses themselves. For a keyword that does not match any item, DESEARCH performs dummy accesses.

Note that Circuit-ORAM does not support concurrent accesses, so DESEARCH leverages multiple ORAM replicas for higher throughput. Specifically, DESEARCH encodes the underlying data in multiple ORAM instances, and accesses different instances to process queries. This is safe because we use ORAM exclusively for read-only workloads, and each instance is independent and has its own position map. This requires more storage space, but DESEARCH allows executors to make this trade-off.

Equalizing response lengths. Beyond ORAM, DESEARCH needs to avoid leaking information from the number of matched result items (count) and the length of each item (volume). We observe that results from search engines are highly regular: search results are displayed in multiple pages; each page contains a fixed number of items; and each item contains a link (e.g., URL) and a small content snippet highlighting the keywords. Therefore, DESEARCH equalizes result lengths by returning a fixed number of entries for each search request, and each entry has a 256-byte summary of the original contents, which we believe to be sufficient for most cases. For comparison, we find that over 80% of search results from Google are within 256 bytes. DESEARCH hides the counts and volume of keywords by padding search queries to the same length and limiting the number of keywords to 32 (the maximum supported by Google).

DESEARCH Components	Language	Total LoC
Openbazaar Crawler	Golang	178
Steemit Crawler	Python	190
Indexer	C++	701
Querier	C++	925
Master	C++	106
Verifier	C++	502
Search Engine Library (Sphinx [38])	C++	37,271 (+63)
ORAM Library (ZeroTrace [102])	C++	3,851 (+188)
Crypto Library (HACL⋆)	C/C++	4,071

FIGURE 8—Lines of code of each component in DESEARCH.

7 Implementation

System components. Figure 8 lists the components of DESEARCH's implementation. We implement our own crawlers that parse raw data from Steemit and OpenBazaar. The Steemit crawlers aggregate data from the Steem blockchain, and the OpenBazaar crawlers work as OpenBazaar peers to retrieve the online shopping items. Our indexer and querier borrow the tokenizer implementation from Sphinx v2.0 [38]. DESEARCH's indexer is implemented to support oblivious index access with ZeroTrace's ORAM implementation [102].

DESEARCH's verifiable lambda is built on Intel SGX SDK 2.13. We use Redis v6.2 as Kanban, and implement a Kanban protocol using Redis++ v1.2 [31] (a Redis client library). DESEARCH commits epoch snapshot digests to an EOS blockchain [11] testnet, which acts as an append-only log. Further, we deploy a dedicated smart contract that provides APIs to read and write the on-log data.

DESEARCH parameters. DESEARCH has several parameters that heavily influence the search performance. We elaborate on these parameters and our choices below.

We set up Circuit-ORAM with a bucket size of 2 (parameter Z) and set the stash size to 10 because they can achieve the best temporal and spatial efficiency. We use two independent ORAM instances, one for the index and another for the search result summaries, and overlap their operations to minimize the latency.

	Shard Size		
Metrics	**10K**	**100K**	**1M**
SGX Load Time	0.3s	2.9s	20.8s
ORAM Setup Time	2.0s	23.6s	267.7s
Worst Response Time	24ms	52ms	452ms
DRAM Required	121MB	285MB	2.34GB
SGX EPC Used	79MB	80MB	86MB

FIGURE 9—Execution time and memory usage of a DESEARCH querier under different shard sizes. SGX load time includes fetching the index/summary files from Kanban and in-enclave deserialization. ORAM setup time includes initializing two Circuit-ORAM instances for inverted index and search result summaries, respectively.

When the targeted dataset exceeds one executor's capacity (including both SGX EPC's and ORAM's capacity), DESEARCH has to split the dataset into shards. We experiment with several shard sizes (Figure 9), and choose 1M data items per shard because 1M-item-shard does not exceed SGX physical memory capacity and the response time is acceptable (within 1 second).

In DESEARCH, each epoch is set to 15min because most existing blockchains yield at least one block in this time frame [22]. It is also long enough for the master to summarize the epoch snapshot on Kanban. For shorter epochs, one could use an incremental hash function [58, 105] to create the digest incrementally throughout the epoch.

Side-channel defenses. DESEARCH use the Ed25519 implementation from a formally verified cryptographic library HACL⋆ v0.2.1 [120], which is resistant to digital (cache and timing) side channels [69, 82]. For ORAM block encryption, we choose AES-NI-based AES-128-GCM. AES-NI is purportedly side-channel resistant according to Intel [78]. Finally, we apply patch (commit f74c8a4) from Intel for SGX-OpenSSL [23] to mitigate hardware vulnerabilities [110].

Limitations. DESEARCH's current implementation only supports full-text search. The links of images, audio, or video, encoded in the texts may be hosted in other unverified servers. DESEARCH does not guarantee their integrity. In terms of privacy, DESEARCH implementation does not hide the frequency of ORAM accesses. Frequency smoothing techniques [76] can help at the cost of additional storage overhead.

8 Evaluation

Our evaluation answers the following questions:

- What is the overall performance of DESEARCH, in terms of end-to-end latency, throughput, and scalability? (§8.1)
- How long does it take to verify a search result? (§8.2)
- Does DESEARCH tolerate executor failures? (§8.3)

Experimental setup. We deploy DESEARCH on a small set of SGX-enabled desktop machines: three machines with 12-core Intel i7-8700, three with 8-core Intel i7-8559U, three with 8-core Intel i7-9700, and all nine machines have at least 8GB DRAM. These machines are connected by a 1Gbps local network. To simulate a large-scale decentralized environment, we also deploy DESEARCH on 1312 nodes of AWS EC2 VMs. Each node is an AWS t2.medium instance with 2-vCPU of Xeon E5-2676 and 4GB of DRAM. These nodes are spread across four geographic regions: Singapore (Asia), London (Europe), West Virginia (East America), Califonia (West America), and are connected through a wide-area network.

In the following experiments, we run DESEARCH in one of two modes: (1) normal mode, in which we run DESEARCH as-is; and (2) simulation mode, where we run DESEARCH under SGX SDK simulation mode and add ORAM latencies to each query instead of actually interacting with DESEARCH's ORAM implementation. We use the simulation mode for the large-scale scalability experiment ("Scalability" in §8.1) which requires hundreds to thousands of executors.

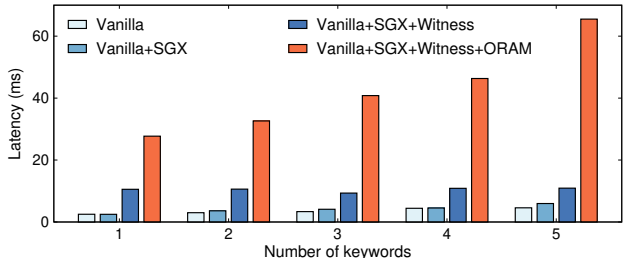
Datasets. We run DESEARCH on two datasets:

- *Steemit.* Steemit [40] is a decentralized blogging and social media service built upon the Steem blockchain [39]. DESEARCH crawlers constantly fetch the latest posts from the Steem blockchain and write them to Kanban. At the time of evaluation, we fetched a total of 81,681,388 posts, distributed across 952 epochs (nearly 10 days) leading to a 234GB dataset. The corresponding indexes contain 296K keywords (we omit 659 stopwords according to Google stopword list [42]). All of this results in 37.68GB crawling-phase witnesses and 6.25GB indexing-phase witnesses.
- *OpenBazaar.* OpenBazaar [28] is a decentralized e-commerce platform where individuals can trade goods without middlemen. The OpenBazaar frontend provides an API [5] for a customized search engine to update the shop contents by crawling IPFS [59], an append-only storage. DESEARCH crawls all OpenBazaar's shopping lists but ignores the ones that are removed by sellers (though they still exist in IPFS). At the time of evaluation, OpenBazaar has (on average) 21K listings per day, and there are 85K keywords in the indexes. Crawling and indexing witnesses are 10.26MB and 1.28MB in size.

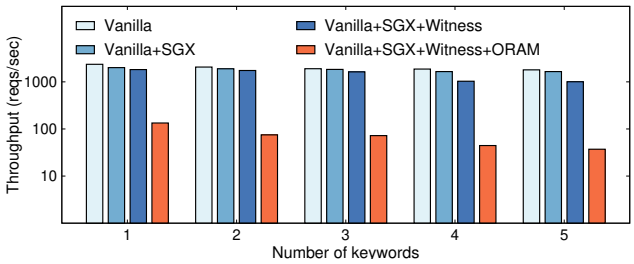
8.1 Serving performance

Querier bootstrap. To offer search for Steemit, DESEARCH splits the Steemit dataset (234GB) into 82 shards, each managing 1 million data items (Steemit posts). One Steemit querier serves one shard. It takes 333.5s for a querier to finish bootstrap, which includes establishing an SGX verifiable lambda, fetching the index and summary files from Kanban, and initializing ORAM instances. An OpenBazaar querier takes 25s to boot because it has a much smaller dataset—a querier serves 21K OpenBazaar data items.

Search throughput and latency. We run a DESEARCH querier for the two datasets on an SGX normal-mode machine and run clients on another machine. To capture a steady-state



(a) The 99th percentile latency of multi-keyword searches.



(b) The average throughput of multi-keyword searches.

FIGURE 10—Differential analysis of latency and throughput for multi-keyword searches. "Vanilla" is a native (unprotected) querier; "SGX" represents the SGX-based isolation; "Witness" represents the lambda confinement; "ORAM" represents the Circuit-ORAM protection. Note that the applied ORAM does not support concurrency while others use 8 threads.

performance, we warm up each querier by issuing 10,000 requests before the experiments. We measure the end-to-end throughput and latency by randomly searching from a list of 10,000 frequently appeared keywords (in each dataset).

For the Steemit dataset, DESEARCH has an average throughput of 133.8 requests/sec, and the 99 percentile end-to-end latency is 21.71ms. Although the measured throughput is modest, we expect that a decentralized network can achieve higher throughput as more executors join (see "Scalability" below). For OpenBazaar, DESEARCH's throughput is 581 requests/sec on average, and the 99 percentile latency is 9.19ms.

Overhead analysis. How do DESEARCH's techniques (including trusted hardware, the witness mechanism, and oblivious protections) affect search performance? To answer this question, we conduct a differential analysis for a Steemit querier, which manages 1M data items. Again, clients issue search queries by randomly picking keywords from a top-10,000 keyword list. But now we experiment with multi-keyword searches with up to 5 keywords. Figure 10 shows the average throughputs and the 99th percentile latency for different multi-keyword searches.

Figure 10 illustrates that putting a querier into SGX with DESEARCH's lambda enforcement decreases throughputs by 15.2% and adds 0.4% overhead to latency. This is because each request triggers *ecalls* (i.e., a context switch between SGX enclave and user-space) in DESEARCH's isolated sandboxes. In the future, we can optimize DESEARCH with the asynchronous call mechanism as studied in SCONE [56].

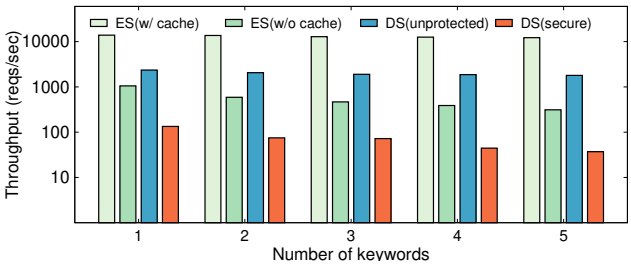


FIGURE 11—Throughput comparison between ElasticSearch and DESEARCH for multi-keyword searches. "ES" represents ElasticSearch and "DS" represents DESEARCH. Both "ES" and "DS (unprotected)" use multi-threading with 8 threads (CPU core number is 8)."DS (secure)" uses ORAM which does not support concurrent accesses, hence it does not benefit from multi-threading.

DESEARCH's witness generation imposes 7.6% overhead in throughput and 8ms in latency. Specifically, the witness generation consists of hashing the lambda's inputs and outputs with standard SHA-256, and signing this witness with Ed25519. Finally, the dominating overhead factor is Circuit-ORAM, which incurs 12.6× throughput degradation and increases latency by 4.5–11.9×. While this overhead is considerable, ORAM provides strong privacy guarantees against the executor adversary that controls the lambda.

Considering different multi-keyword searches, their latency is proportional to the number of keywords as searching multiple keywords requires fetching multiple rounds of index blocks from the Circuit-ORAM server. Historical statistics [19] show that 71.3% search queries do not exceed four keywords. Four-keyword ORAM-based searches in DESEARCH have a 46.3ms (99 percentile) latency, which is acceptable in a human interactive process.

Comparison with ElasticSearch. ElasticSearch [10] (or ES) is a popular search engine system that has been widely deployed. We compare DESEARCH with ES under an 1M-items dataset. We configure ES's Java runtime memory to 2.5GB, which is the maximum memory consumed by a DESEARCH Steemit querier. Figure 11 depicts the results.

By enabling caching, ES can achieve a throughput of 13.9K requests/sec, 4.8× faster than a DESEARCH implementation without integrity and privacy protection. But this is an unfair comparision because DESEARCH does not have caches (caches will break the security guarantee of ORAM). We further experiment with ES by disabling caching. Figure 11 shows that DESEARCH has higher throughputs than ES without caching because DESEARCH's implementation is simpler and has less functionality (e.g., complex item ranking policy). Finally, we compare ES with the full-fledged DESEARCH with SGX verifiable lambda and ORAM. As the ORAM operations limit the concurrency of the service, DESEARCH's throughput drops significantly.

We also observe that ES's throughput decreases mildly as the number of search keywords increases (around 11%

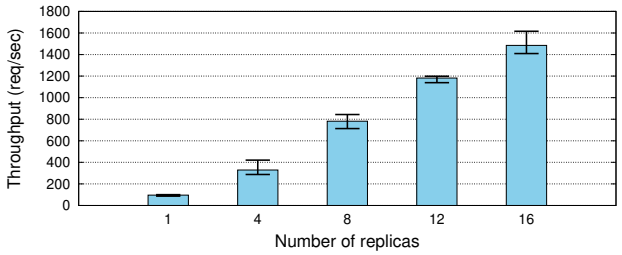


FIGURE 12—DESEARCH's throughputs for different number of replicas. Each replica consists of 82 queriers.

	Native	Delegated	Speedup
Execution Integrity			
Witness Download	517.0s	–	
Signature Verification	4h25min	–	
Data Integrity			
Witness Tree Verification	201.9s	–	
Final-Phase Verify			
Verifier Interaction	0s	1.0s	
Signature Verification	0s	0.2s	
Total Time	4h33min	1.2s	13,681x

FIGURE 13—User-side verification costs for a native (with an 8-core CPU) verification and a delegated verification.

decrease from single-keyword to five-keyword search), while the throughput of the full-fledged DESEARCH drops more rapidly (72.3% decrease). This is because multi-keyword searches require multiple rounds of ORAM item retrievals.

Scalability. To evaluate DESEARCH's scalability, we measure the overall throughput of DESEARCH with different number of *replicas*; each replica is a fully functional DESEARCH instance that consists of 82 Steemit queriers. We run DESEARCH in simulation mode, and use 82×16 2-core AWS EC2 VMs as servers to simulate a decentralized network of executors. Each virtual machine hosts a Steemit querier. From 4 to 16 replicas, we compose a geo-distributed setup where these replicas are equally deployed on four regions (i.e., Singapore, London, West Virginia, Califonia). We deploy clients on 8 other 96-core c5.24xlarge machines.

The results are shown in Figure 12. DESEARCH's throughputs increase horizontally with the number of replicas. With 16 replicas, DESEARCH can support 1484 requests/sec, that is 128 million requests per day.

8.2 Verification cost

A user verifies the final results by holding: (1) the digests of each epoch-based snapshot (retrieved from the public log), (2) executors' public keys, (3) witnesses from queriers (sent to users with search results) and other executors (stored on Kanban). Since the first two can be prefetched, a verification does not include fetching them. Figure 13 shows the *user-side* costs of native verifications and delegated verifications for a single-keyword search on the Steemit dataset.

In a native verification, a user verifies the search results on their own. To verify execution integrity, it takes 517s to download witnesses from Kanban, and 15698s to verify the signatures using Ed25519. The majority of the time is spent on checking crawlers' witnesses, as the number of witnesses is proportional to the number of items crawled. To verity data integrity, the user first ensures the witnesses from Kanban are consistent with the digests from the public log, and then breadth-first traverses the witness tree. This process takes 201.9s to complete.

In a delegated verification, the user sends the witnesses received from queriers to a verifier, and the verifier examines the execution and data integrity on behalf of the user. If the verifier accepts, the user only needs to verify the witnesses in the private domain, namely the witnesses from queriers. It takes 1.0s to interact with a verifier and 0.2s to verify the hashes and signatures of queriers' witnesses.

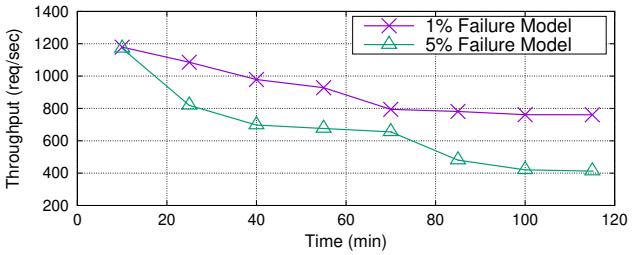


FIGURE 14—DESEARCH throughput changes under 1% and 5% per-epoch executor failure rate. The experiment uses Steemit dataset with 16 replicas and runs for 2 hours (8 epochs).

8.3 Fault tolerance

To understand how executor failures affect DESEARCH's performance, we run DESEARCH for Steemit with 16 replicas and 20 shards (4×20 queriers), and randomly kill a certain number of queriers in each epoch. In particular, we have two failure workloads: one kills 1% of the current available queriers in each epoch (15 min), and the other kills 5% per epoch. If a querier does not respond in 1 second, clients will issue the request to other queriers. Note that our experiment ensures that the remaining online queriers always comprise a complete dataset, otherwise all queries will fail (since they will only be able to provide partial results).

We run the experiments under these two workloads for 2 hours (8 epochs) each. Figure 14 shows the results. When 1% queriers fail per epoch, we observe that DESEARCH's throughput drops from 1,179 requests/sec to 761 requests/sec after 2 hours, a 35.4% decrease. As a comparison, in the last epoch, DESEARCH loses 7.7% ($= 1 - 0.99^8$) of the initial queriers. The throughput degrades significantly because every search has to get responses from all shards to make up a full result, but killed nodes do not perfectly distribute across shards. Hence, shards with fewer available queriers become a performance bottleneck. In the workload with 5% failure rate

per epoch, the throughput drops from 1,171 requests/sec to 412 requests/sec in 8 epochs (a 64.8% throughput degradation), while 33.7% ($= 1 - 0.95^8$) queriers are killed.

9 Discussion

Incentive model. As with existing decentralized systems [94, 113], DESEARCH relies on volunteers to offer service availability. To encourage TEE owners to join in DESEARCH, we discuss a possible incentive model, inspired by Teechain [85], where cryptocurrency can be securely transferred from a blockchain to a TEE enclave. We observe that current Steem Dollars (SBD) flow from readers to mostly popular post authors as rewards. Our incentive model, instead, transfers SDB from readers $\rightarrow$ queriers $\rightarrow$ indexers $\rightarrow$ crawlers $\rightarrow$ authors, where data needs to be paid for the usage. Our witness is a natural fit for proving the usage. A higher contribution of TEE computing power would then translate into a higher cryptocurrency reward.

System bootstrap. To bootstrap DESEARCH, any TEE executor can become a master, as long as it downloads the code of master and runs the code within an attestable enclave. The master's initialization will generate a pair of public/private keys. The former is recorded on the append-only log for public availability, and the latter is kept within the enclave, being DESEARCH's root key. Any party can use the TEE's remote attestation mechanism to verify the root key's genuineness. The first master then registers itself in the active list on Kanban. A set of executors (say, 21, a DESEARCH parameter) are required to join and serve as masters to start the first epoch. DESEARCH masters run independently and form a decentralized network to avoid a single point of failure. Masters can change over time via periodic selections (e.g., BFT sortition protocols [75]). Executors that join at a later time after the masters network is active will simply be assigned other roles.

Further, to resist possible supply-chain attacks [37], DESEARCH can use reproducible builds [33] to ensure verifiability from source code to lambda images.

Beyond search. DESEARCH, as a general framework, is not limited to search. As an example, it is intuitive to extend DESEARCH to a verifiable recommender system for OpenBazaar by replacing querier's ranking function with a recommendation function. A user can verify that the advertisements are chosen based on their interests, without opaque manipulation. Also, DESEARCH can be used as a "watchdog" for other search services. Users can continue to use an existing search engine, and cross-validate the results with DESEARCH.

10 Related Work

Decentralized search engines. There are several prior efforts in building decentralized search services. For example, YaCy [48] is a peer-to-peer distributed search engine since 2004. It enables decentralized index generation and supports shared indexes among peers. YaCy assumes that peers are honest, which might not be true in an open environment. Presearch [29] leverages a blockchain to provide decentralized search, but inherits blockchain's characteristics, including duplicated computations and no privacy guarantees. The Graph [20] and PureStake [30] are indexing services for decentralized storage, but neither provides privacy, and PureStake does not provide integrity either.

Verifiable search for decentralized services. There is an increasing interest in providing verifiable search for decentralized services, due to the diversified usages of decentralized applications (see Figure 1). For verifiable blockchain searches, vChain [114] adopts authenticated data structures (ADS) while GEM^2-Tree [117] explores on-chain indexes. Compared with vChain and GEM^2-Tree which only support range-based searches, DESEARCH provides full-text search and offers verifiability via witness-based dataflow tracking.

IPSE [24] provides search over IPFS [59] (a decentralized storage) and provides hash-based content verifiability. Freenet [15] is an anonymous file-sharing network (similar to BitTorrent) but is not search-oriented. Compared to DESEARCH, IPSE and Freenet lack data integrity (§2.2), which is troublesome because incomplete data sources can make the search vulnerable to censorship [41].

Private search with TEE. TEE is a hot topic for providing private search. To hide users' search intention, X-Search [92] uses a cloud-side TEE proxy while Cyclosa [98] adopts browser-side TEE proxies. X-Search and Cyclosa are metasearch engines (a proxy between users and a search engine) that reveal query keywords and results to search providers, whereas DESEARCH is a complete search engine that provides query and result privacy.

A long line of prior works (Opaque [119], OCQ [72], ZeroTrace [102], Oblix [91], Obliviate [51], etc.) have explored TEE and ORAM combination to protect search. Similar efforts have been made by combining symmetric searchable encryption and TEEs (e.g., Rearguard [108]), or private information retrieval and TEE (e.g., SGX-IR [106]). While DESEARCH uses similar primitives, DESEARCH's architecture results in the first fully built decentralized system to serve searches on real-world datasets with integrity and privacy.

Secure big-data systems with TEE. Many prior systems use TEEs for big-data computation [61, 72, 77, 101, 104, 119]. VC3 [104] secures a map-reduce framework with TEE, and Opaque [119] protects SQL queries for Spark SQL. Unlike the setting of VC3 and Opaque, DESEARCH faces a dynamic computation graph because the set of live executors is constantly changing given our decentralized environment. DESEARCH therefore employs an epoch-based snapshot and witnesses mechanism to ensure data and execution integrity. Ryoan [77] provides distributed sandboxes for private data computation. Compared with Ryoan, DESEARCH offers publicly verifiable witnesses for reusable intermediate data and

effectively reduces the verification cost (see Section 8.2).

TEE and serverless computing. The notion of verifiable lambda is inspired by serverless computing. DESEARCH extends this notion from centralized cloud-based computing to decentralized computing. S-FaaS [52], T-FaaS [63], Clemmys [109] are TEE-based serverless systems that protect serverless workloads with TEE. Instead, DESEARCH utilizes epoch-based Kanban and TEE-generated witnesses to maintain and verify the state of the (stateless) TEE lambdas.

New decentralized systems. Recently, new architectures of decentralized systems have been proposed to address the limitations (low-throughput, resource waste, lack of privacy) of conventional decentralized ledgers or blockchains. Algorand [75] and Blockene [103] propose new consensus protocols to achieve high-throughput. Omniledger [80] and Protean [53] introduce sharding to scale out the blockchain. Similarly, DESEARCH shards executors to different roles, and offloads states to Kanban to achieve high scalability.

Other TEE-based systems for decentralized services. TEEs have been used to build a provable blockchain oracle [118], off-chain smart contracts [70], Bitcoin fast payment channel [85] and lightweight clients [89], and online-service secure sharing [88]. They share the same goal towards shaping a better decentralized world but differ from DESEARCH in their target functionality.

11 Conclusion

DESEARCH is the first decentralized search engine to support existing decentralized services, while guaranteeing verifiability owing to its *witness* mechanism and offering privacy for query keywords and search results. DESEARCH achieves good scalability and minimizes fault disruptions through a novel architecture that decouples the decentralized search process into a pipeline of verifiable lambdas and leverages a global and highly available Kanban to exchange messages between lambdas. We implement DESEARCH on top of Intel SGX machines and evaluate it on two decentralize systems: Steemit and OpenBazaar. Our evaluation shows that DESEARCH can scale horizontally with the number of executors and can achieve the stringent subsecond latency required for a search engine to be widely usable.

DESEARCH 's source code will be released at:
`https://github.com/SJTU-IPADS/DeSearch`

Acknowledgments

We thank the anonymous reviewers of OSDI 2020 and OSDI 2021 for their helpful and constructive comments, and our shepherd Marko Vukolic for his guidance. We also thank Sajin Sasy for helpful discussion about ORAM. This work was supported in part by National Key Research and Development Program of China (No. 2020AAA0108500), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Sebastian Angel was funded by NSF Award CNS-2045861 and DARPA contract HR0011-17-C0047. Cheng Tan was funded by AFOSR FA9550-18-1-0421 and NSF CNS-1514422. Yubin Xia (xiayubin@sjtu.edu.cn) is the corresponding author.

References

[1] Augur. `https://www.augur.net/`.
[2] Blockstamp openbazaar explorer. `https://bazaar.blockstamp.market/`.
[3] Cam4 live-streaming adult site exposed 7tb records publicly. `https://www.2-spyware.com/cam4-live-streaming-adult-site-exposed-7tb-records-publicly`.
[4] Censorship by google. `https://en.wikipedia.org/wiki/Censorship_by_Google`.
[5] Content discovery on OpenBazaar. `https://openbazaar.org/blog/decentralized-search-and-content-discovery-on-openbazaar/`.
[6] Crawl stats report - search console help - google support. `https://support.google.com/webmasters/answer/9679690`.
[7] Data-enriched profiles on 1.2b people exposed in gigantic leak. `https://threatpost.com/data-enriched-profiles-1-2b-leak/150560/`.
[8] Dtube. `https://d.tube`.
[9] Duo search is a search engine for openbazaar. `https://bitcoinist.com/duo-search-is-a-search-engine-for-openbazaar/`.
[10] Elasticsearch. `https://www.elastic.co/`.
[11] Eosio blockchain software & services. `https://eos.io/`.
[12] Etoro. `https://stocks.etoro.com/`.
[13] Facebook is illegally collecting user data, court rules. `https://thenextweb.com/facebook/2018/02/12/facebook-is-illegally-collecting-user-data-court-rules/`.
[14] Facebook suspends cambridge analytica for misuse of user data, which cambridge denies. `https://www.cnbc.com/2018/03/16/facebook-bans-cambridge-analytica.html`.
[15] Freenet. `https://freenetproject.org/`.
[16] Giving social networking back to you - the mastodon project. `https://joinmastodon.org/`.
[17] Golem network. `https://golem.network/`.
[18] Google faces antitrust investigation by 50 us states and territories. `https://www.theguardian.com/technology/2019/sep/09/google-antitrust-investigation-monopoly`.
[19] Google search statistics and facts 2020. `https://firstsiteguide.com/google-search-stats/`.
[20] The graph is an indexing protocol for querying networks like ethereum and ipfs. `https://thegraph.com`.
[21] Graphite docs. `https://www.graphitedocs.com/`.
[22] How many bitcoins are mined everyday? `https://www.buybitcoinworldwide.com/how-many-bitcoins-are-there/`.

[23] Intel® software guard extensions ssl. https://github.com/intel/intel-sgx-ssl.
[24] Ipfs search. https://ipfs-search.com/#/search.
[25] Matrix - an open network for secure, decentralized communication. https://matrix.org/.
[26] Microsoft azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.
[27] The monopoly-busting case against google, amazon, uber, and facebook. https://www.theverge.com/2018/9/5/17805162/monopoly-antitrust-regulation-google-amazon-uber-facebook.
[28] OpenBazaar. https://openbazaar.org/.
[29] Presearch is a decentralized search engine, powered by the community. https://www.presearch.io/.
[30] Purestake: Blockchain infrastructure for proof of stake networks. https://www.purestake.com/.
[31] Redis client written in c++. https://github.com/sewenew/redis-plus-plus.
[32] Report: 267 million facebook users ids and phone numbers exposed online. https://www.comparitech.com/blog/information-security/267-million-phone-numbers-exposed-online/.
[33] Reproducible Builds. https://reproducible-builds.org/.
[34] Searchbizarre. https://searchbizarre.com/.
[35] Searching the blockchain (indexer v2) - algorand developer docs. https://developer.algorand.org/docs/features/indexer/.
[36] Serverless computing. https://en.wikipedia.org/wiki/Serverless_computing.
[37] The solarwinds orion breach, and what you should know. https://blogs.cisco.com/security/the-solarwinds-orion-breach-and-what-you-should-know.
[38] Sphinx: Open source search server. http://sphinxsearch.com/.
[39] Steem. https://steem.com/.
[40] Steemit. https://steemit.com/.
[41] Steemit censoring users on immutable social media blockchain's front-end. https://cryptoslate.com/steemit-censoring-users-immutable-blockchain-social-media/.
[42] Stop words - words ignored by search engines. https://www.link-assistant.com/seo-stop-words.html.
[43] Supporting intel sgx on multi-socket platforms. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html.
[44] A timeline of facebook's privacy issues — and its responses. https://www.nbcnews.com/tech/social-media/timeline-facebook-s-privacy-issues-its-responses-n859651.
[45] Tital-advancing the era of accelerated computing. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.
[46] Welcome to bazaar dog, your scrappy open bazaar search provider. https://www.bazaar.dog/.
[47] Wix.com: Free website builder. https://www.wix.com/.
[48] Yacy - decentralized search engine. https://yacy.net/.
[49] Zeronet. https://zeronet.io/.
[50] Zipcall.io. https://meet.questo.ai/.
[51] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
[52] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2019.
[53] Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, and Bryan Ford. Rethinking general-purpose decentralized computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.
[54] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.
[55] ARM. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone.
[56] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[57] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: An online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM Conference*, 2009.
[58] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1997.
[59] Juan Benet. Ipfs - content addressed, versioned, p2p file system. *ArXiv*, abs/1407.3561, 2014.
[60] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.
[61] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
[62] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
[63] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2019.
[64] Jo Van Bulck, M. Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In

USENIX Security Symposium, 2018.

[65] 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 *Proceedings of the USENIX Security Symposium*, 2017.

[66] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[67] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[68] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[69] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[70] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[71] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[72] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[73] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proceedings of the USENIX Security Symposium*, 2015.

[74] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[75] Y. Gilad, Rotem Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[76] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *Proceedings of the USENIX Security Symposium*, 2020.

[77] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[78] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`.

[79] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[80] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[81] Ziliang Lai, Chris Liu, Eric Lo, Ben Kao, and Siu-Ming Yiu. Decentralized search on decentralized web. *ArXiv*, abs/1809.00939, 2019.

[82] 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 *Proceedings of the USENIX Security Symposium*, 2017.

[83] Jinyuan Li, M. Krohn, David Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[84] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in amd's secure encrypted virtualization. In Nadia Heninger and Patrick Traynor, editors, *Proceedings of the USENIX Security Symposium*, 2019.

[85] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter R. Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[86] Prince Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[87] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[88] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *Proceedings of the USENIX Security Symposium*, 2018.

[89] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: bitcoin lightweight client privacy using trusted execution. In *Proceedings of the USENIX Security Symposium*, 2019.

[90] P. Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[91] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[92] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. X-search: revisiting private web search using intel SGX. In *Proceedings of the*

ACM/IFIP/USENIX International Middleware Conference, 2017.

[93] 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.

[94] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[95] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[96] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *USENIX ATC*, 2019.

[97] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[98] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. CYCLOSA: decentralizing private web search through sgx-based browser extensions. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[99] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[100] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[101] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. Sgx-pyspark: Secure distributed data analytics. In *International World Wide Web Conference (WWW)*, 2019.

[102] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[103] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A high-throughput blockchain over mobile devices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[104] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[105] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[106] Fahad Shaon and Murat Kantarcioglu. Sgx-ir: Secure information retrieval with trusted processors. In *DBSec*, 2020.

[107] Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[108] Wenhai Sun, Ruide Zhang, Wenjing Lou, and Y. Thomas Hou. REARGUARD: secure keyword search using trusted hardware. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

[109] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: towards secure remote execution in faas. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[110] 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 *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[111] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[112] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[113] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.

[114] Cheng Xu, Ce Zhang, and Jianliang Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the ACM SIGMOD Conference*, 2019.

[115] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[116] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[117] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2019.

[118] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[119] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca A. Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[120] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing

Youngseok Yang[1] Taesoo Kim[2] Byung-Gon Chun[1,3*]
[1]Seoul National University [2]Georgia Institute of Technology [3]FriendliAI

Abstract

Ethereum is the second-largest blockchain platform next to Bitcoin. In the Ethereum network, decentralized Ethereum clients reach consensus through transitioning to the same blockchain states according to the Ethereum specification. Consensus bugs are bugs that make Ethereum clients transition to incorrect blockchain states and fail to reach consensus with other clients. Consensus bugs are extremely rare but can be exploited for network split and theft, which cause reliability and security-critical issues in the Ethereum ecosystem.

We describe Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. First, Fluffy mutates and executes multi-transaction test cases to find consensus bugs which cannot be found using existing fuzzers for Ethereum. Second, Fluffy uses multiple existing Ethereum clients that independently implement the specification as cross-referencing oracles. Compared to a state-of-the-art fuzzer, Fluffy improves the fuzzing throughput by 510× and the code coverage by 2.7× with various optimizations: in-process fuzzing, fuzzing harnesses for Ethereum clients, and semantic-aware mutation that reduces erroneous test cases.

Fluffy found two new consensus bugs in the most popular Geth Ethereum client which were exploitable on the live Ethereum mainnet. Four months after we reported the bugs to Geth developers, one of the bugs was triggered on the mainnet, and caused nodes using a stale version of Geth to *hard fork* the Ethereum blockchain. The blockchain community considers this hard fork the greatest challenge since the infamous 2016 DAO hack. We have made Fluffy publicly available at https://github.com/snuspl/fluffy to contribute to the security of Ethereum.

1 Introduction

> The case is perhaps Ethereum's greatest challenge since the 2016 DAO fork, and it raises questions about Ethereum's oft-touted decentralization and the effectiveness of its developer coordination going into Ethereum 2.0.
>
> — *Coindesk, November 12th, 2020* [11]

An extremely rare consensus bug[1], which makes Ethereum [17, 54] clients transition to incorrect blockchain states and fail to reach consensus with other clients, was triggered on the Ethereum mainnet on November 11th, 2020 [2, 11, 43, 45]. This was one of the two consensus bugs we found and reported to Ethereum developers four months before that date. The bug caused Ethereum nodes using a stale version of Geth [19], the most popular Ethereum client, to hard fork the Ethereum blockchain. One of the affected nodes was Infura [15], the largest infrastructure service that allows decentralized applications (DApps) to connect to the Ethereum network without having to run their own Ethereum nodes.

Consequently, Infura went down, and with it some of most popular Ethereum applications such as Metamask, MakerDAO, Uniswap, and Compound went down [11]. Shortly after, cryptocurrency exchanges around the world including Binance, the largest exchange, halted the trading of ETH, the cryptocurrency of Ethereum [2, 11]. Infura and others quickly upgraded their Geth clients to fix the bug. Nevertheless, around 30 Ethereum blocks from block 11234873 on the forked chain were lost [23], which transferred approximately 8.6 million USD worth of ETH[2]. The blockchain community considers this hard fork the greatest challenge since the infamous DAO hack of 2016 [11, 13].

This paper describes how we found such extremely rare, high-impact consensus bugs in the heavily-tested Ethereum network through *fuzzing* [9, 30, 38], an automated software testing technique that randomly mutates inputs and tests the target program on the resulting data.

Finding new consensus bugs in actively used Ethereum clients is challenging, because consensus bugs are extremely rare. Attackers can exploit consensus bugs for network split and theft, which cause reliability (e.g., delaying transactions) and security-critical issues (e.g., stealing ETH) in the Ethereum ecosystem. To prevent such issues, Ethereum developers make preventing consensus bugs a top priority, and invest heavily in auditing, testing, and fuzzing Ethereum

*Corresponding author.

[1]We focus on consensus bugs in state management that make Ethereum clients transition to incorrect blockchain states. Bugs in blockchain consensus algorithms such as proof of work are not the focus of this paper.

[2](Total amount of ETH transferred by block 11234873 to 11234902 on the canonical chain) × (Closing price of ETH/USD on November 10th)

	EVMLab	EVM libFuzzer	EVM Fuzzer	Fluffy
Transactions per test	Single	Single	Single	**Multiple**
Contract language	Bytecode	Bytecode	Solidity	**Bytecode**
In-process fuzzing	-	✓	-	✓
Coverage-guided	-	✓	-	✓
Open source	✓	-	✓	✓
Created in	2017	2017	2019	**2020**
Impact of newly found bugs (Count)	High (1), Low (2)	Low (2)	N/A	**High (2)**

Table 1: A comparison of Fluffy and existing fuzzers for finding consensus bugs in Ethereum.

clients [18, 21, 22, 33]. Since the Ethereum network launched in July 2014, only 13 consensus bugs have been found in the most popular Geth [19] and OpenEthereum [41] clients, and only 6 of them would have been exploitable on the live Ethereum mainnet [34].

Ethereum fuzzers have found most of the consensus bugs [18, 22, 33, 34]. These existing fuzzers focus on the blockchain state, which is a set of Ethereum accounts that hold a balance of ETH. Specifically, the fuzzers model an Ethereum client as a blockchain state model, in which the blockchain state is transitioned by an Ethereum transaction. As a result, in each fuzzing iteration, they generate and test a pre-transaction blockchain state and a single transaction that transitions the blockchain state. Table 1 compares the existing fuzzers. EVMLab [18] generates random Ethereum Virtual Machine (EVM) bytecode of Ethereum smart contracts [54], and invokes the contracts with a single transaction. EVMLab has found in the most popular Geth and OpenEthereum clients one high-impact bug that was exploitable on the mainnet and two low-impact bugs that were not exploitable on a live network. EVM libFuzzer [33] is a closed source fuzzer whose details are unknown. EVM libFuzzer integrates with libFuzzer [30] and has found two low-impact bugs. EVMFuzzer [22] is a more recent fuzzer that generates Solidity [20] code of contracts.

However, the blockchain state model of existing fuzzers falls short to cover the full search space for finding consensus bugs. The full search space consists of the set of possible client program states, which are the values of program variables of Ethereum clients that can be reached after executing Ethereum transactions. For each pre-transaction blockchain state (e.g., Account A has 0 ETH), the blockchain state model can cover only a single pre-transaction program state (e.g., account_a = { ETH: 0, deleted: false}). Consequently, existing fuzzers fail to test other possible pre-transaction program states (e.g., account_a = { ETH: 3, deleted: true}) that represent the same blockchain state. This leads existing fuzzers to miss consensus bugs which are triggered only when a transaction is applied to such other pre-transaction program states.

To fully cover the search space for finding consensus bugs, we propose to model an Ethereum client as a client program state model, in which the client program state is transitioned by a transaction. Based on this model, in each fuzzing iteration, we generate and execute a sequence of multiple transactions that transition an initial client program state. This allows us to indirectly generate various intermediate pre-transaction program states, which can be reached after executing transactions and can lead to the discovery of new consensus bugs.

We embody our approach in Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. Fluffy mutates and executes multi-transaction test cases to find consensus bugs which cannot be found using existing fuzzers for Ethereum. In addition, Fluffy uses multiple existing Ethereum clients that independently implement the specification as cross-referencing oracles, similar in concept to N-versioning [5]. This technique is known as differential fuzzing [9, 36] in the testing community.

Our Fluffy design employs several new fuzzing techniques. First, we modify existing Ethereum clients to provide an execution model that enables efficient multi-transaction fuzzing. Second, we use multi-transaction test cases that encode dependencies between multiple Ethereum transactions and enable mutating transaction contexts (i.e., sequence of transactions that are executed prior to the transaction) rather than pre-transaction blockchain states. Third, we introduce new semantic-aware mutation strategies for mutating transaction contexts, transaction parameters, and EVM bytecode.

We have implemented Fluffy in Rust and Go, and made it publicly available at https://github.com/snuspl/fluffy. Our current implementation supports fuzzing Geth and OpenEthereum, which are used by 98% of nodes in the Ethereum mainnet, as of August 2020 [6]. Our implementation adopts various optimizations including in-process fuzzing and optimized fuzzing harnesses for Ethereum clients, and also provides a debugger to analyze crashes due to consensus bugs. Our evaluation on a 12-core machine shows that Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. Fluffy also improves the fuzzing throughput by 510× and the code coverage by 2.7× compared to EVMLab.

2 Background

We first provide an overview of Ethereum [17, 54], and then describe consensus bugs in Ethereum.

2.1 Ethereum

The Ethereum blockchain network consists of decentralized peer-to-peer Ethereum clients that implement the Ethereum Virtual Machine (EVM) specification [54]. EVM is a Turing-complete machine that specifies how the Ethereum blockchain

state, a set of Ethereum accounts, is altered through transactions recorded on Ethereum blocks. Accounts in the blockchain state have an address and hold a balance of ETH, the cryptocurrency of Ethereum. There are two types of accounts: the externally-owned account (EOA) owned by an Ethereum user, and the smart contract account which is owned by code and has a key-value storage. In addition to the accounts, EVM provides precompiled contracts that perform specialized operations at fixed addresses. Ethereum transactions are either message call transactions that can invoke smart contracts, or contract creation transactions that create new smart contracts.

Blockchain state. EVM transitions blockchain states through processing Ethereum bytecode instructions invoked by transactions. EVM provides around 140 distinct instructions. Each instruction makes specific changes to EVM internal states such as the EVM stack and the EVM memory, as well as the blockchain state such as the ETH balance of EOAs and the storage of smart contracts. Each instruction also has a specific gas cost, a fee that the transaction sender pays to compensate for the computational effort that it takes to execute the instruction in the network. EVM throws an out-of-gas error when the sum of the gas cost exceeds the gas limit of a transaction.

Client program state. Ethereum clients implement EVM with a specific programming language such as Go and Rust [19, 41]. As a result, a client maintains its own client program state, which are the values of program variables (e.g., Rust or Go variables) that are reached after executing Ethereum transactions. There can be multiple different client program states that represent the same blockchain state, since the client determines the blockchain state it is in by interpreting the values of a subset of its program variables.

Example. Figure 1 shows how an Ethereum client executes a sequence of two transactions to transition an initial blockchain state (State 0). Transaction 1 first creates contract C by invoking the bytecode in its data field, which returns (`RETURN`) the bytecode that becomes the code of contract C (State 1). Transaction 2 then invokes the code of C with specific value and data parameters, such that a particular key-value pair is stored (`SSTORE`) in the storage of C (State 2).

Suppose we initialize an Ethereum client implementation in two different ways. First, we initialize the client directly with the blockchain state after Transaction 1 (State 1). Second, we initialize the client with the initial blockchain state (State 0), and then execute Transaction 1 to transition the blockchain state. Although the resulting blockchain state (State 1) is the same for the two clients, the resulting client program state may not be the same depending on the client implementation. As a result, the two clients can behave differently when executing the subsequent Transaction 2.

We show why it is important to fully test such different behaviors when testing whether clients transition to incorrect blockchain states, which we describe next.

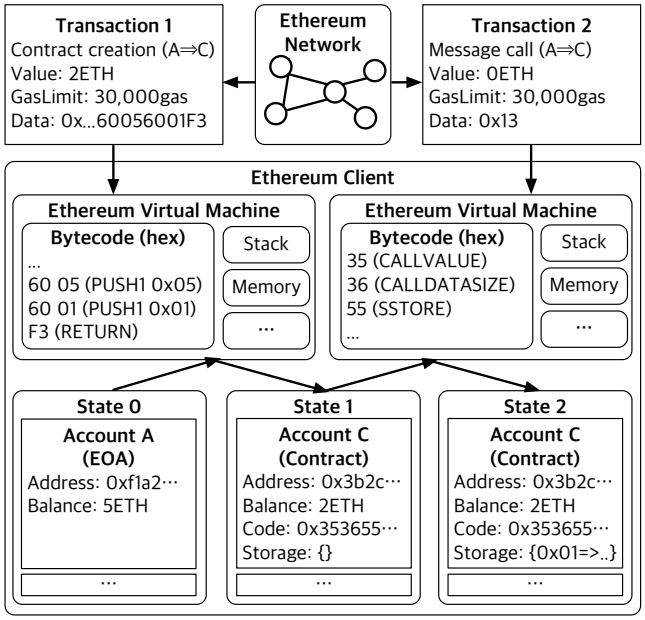


Figure 1: Multiple Ethereum transactions interact to determine the transitions of Ethereum blockchain states.

2.2 Consensus Bugs

Consensus bugs are implementation bugs that make Ethereum clients transition to incorrect blockchain states, and fail to reach consensus with other clients that transition to correct states according to the EVM specification [54]. The Ethereum community has fostered the development of diverse client implementations with a goal to verify the EVM specification and make the Ethereum network more secure. Nevertheless, only two Ethereum clients are used by 98% of nodes participating in the Ethereum mainnet, as of August 2020 [6]. Around 80% of nodes use Geth [19] written in Go, and 18% use OpenEthereum [41], previously called Parity, written in Rust. Therefore, consensus bugs that affect Geth and OpenEthereum have the most critical impacts on the Ethereum ecosystem.

Known bugs. Consensus bugs in actively used Ethereum clients are extremely rare. Table 2 shows the list of consensus bugs [34] reported to have been found in Geth, and OpenEthereum under the former name of Parity. Since the Ethereum network launched in July 2014 until 2019, only 13 consensus bugs were found. Only 6 of the 13 bugs are high-impact bugs that would have been exploitable on the live Ethereum mainnet. In addition to these 6 high-impact bugs, we were able to find 2 new high-impact consensus bugs in 2020, which we describe in detail later in the paper.

Consensus bugs are extremely rare for the following reasons. First, the Ethereum community makes preventing consensus bugs a top priority, and heavily invest in auditing, testing, and fuzzing Ethereum clients [18, 21, 22, 33]. Second, Ethereum clients are continuously being tested on the live

#	Client	Date	Consensus bug description	Tx	Impact	Finding method	Fluffy (Time)
1	Geth	Aug 2020	The balance of a deleted account is carried over to a new account	2	High	**Fluffy**	✓ (291m)
2	Geth	Jul 2020	The DataCopy precompile performs shallow rather than deep copy	1	High	**Fluffy**	✓ (386m)
3	Geth	Mar 2019	Block timestamps exceeding uint64 lead to a wrong block hash	1	High	Unknown	N/A
4	Parity	Oct 2018	The SSTORE gas refund counter does not go below zero when it should	1	Medium	Triggered-Testnet	✓ (57m)*
5	Parity	Jun 2018	Unsigned transactions are accepted and treated as valid	1	Medium	Triggered-Testnet	N/A
6	Geth	Feb 2018	Subgroups in elliptic curve pairings are not validated properly	1	High	Unknown	N/A
7	Parity	Oct 2017	CREATE in static context without enough balance throws a wrong error	1	High	EVMLab	✓ (41m)*
8	Geth	Oct 2017	CALL in static context with less than three stack elements crashes	1	Low	EVM libFuzzer	✓ (38m)
9	Parity	Oct 2017	The gas for the ModExp precompile overflows for certain inputs	1	Low	Manual auditing	Timeout-12h
10	Parity	Oct 2017	RETURNDATACOPY overflows during addition of offset and length	1	Low	EVM libFuzzer	✓ (14m)*
11	Parity	Oct 2017	The gas for the ModExp precompile overflows for large numbers	1	Low	EVMLab	✓ (15m)
12	Parity	Oct 2017	RETURNDATASIZE from a precompile returns a non-zero size	1	Low	EVMLab	✓ (2m)
13	Geth	Feb 2017	The EVM stack underflows for SWAP, DUP, and BALANCE	1	High	Unknown	✓ (6s)
14	Geth	Jan 2017	Undisclosed	-	High	Unknown	N/A
15	Geth	Nov 2016	Fails to revert the deletion of touched accounts on out of gas	1	High	Triggered-Mainnet	✓ (5m)

Table 2: The list of consensus bugs [34] found in Geth and Parity, which is the former name of OpenEthereum. The first two bugs are new bugs found by Fluffy. The number of transactions (Tx) indicates the minimum number of depending transactions required to trigger the bug. High-impact and medium-impact bugs are bugs that would have been exploitable on the live Ethereum mainnet and testnet respectively. Low-impact bugs are bugs that were fixed before they became exploitable on a live Ethereum network. The last column shows the time it takes Fluffy to find the bugs, which is explained in § 7.1.

mainnet for consensus bugs with real-world transactions. On the mainnet, multiple versions of multiple Ethereum clients have reached consensus on a total of more than 800 million transactions as of August 2020.

Ethereum fuzzers. All consensus bugs have been found with fuzzing [9,36], except for the bugs triggered on a live network, found with manual auditing, or whose finding method is unknown. Existing Ethereum fuzzers [18,22,33] focus on the blockchain state. Specifically, the fuzzers model an Ethereum client as a blockchain state model, in which the blockchain state is transitioned by an Ethereum transaction. As a result, in each fuzzing iteration, they generate and test a pre-transaction blockchain state and a single transaction that transitions the blockchain state.

However, the blockchain state model of existing fuzzers falls short to cover the full search space for finding consensus bugs. The full search space consists of the set of possible client program states that can be reached after executing Ethereum transactions. For each pre-transaction blockchain state (e.g., Account A has 0 ETH), the blockchain state model can cover only a single pre-transaction program state (e.g., account_a = { ETH: 0, deleted: false}). Consequently, existing fuzzers fail to test other possible pre-transaction program states (e.g., account_a = { ETH: 3, deleted: true}) that represent the same blockchain state. This leads existing fuzzers to miss consensus bugs which are triggered only when a transaction is applied to such other pre-transaction program states.

Bug impacts. Attackers can exploit consensus bugs for network split and theft. First, an attacker can trigger a consensus bug to make buggy client nodes create a fork of the blockchain, which only they agree on. The transactions recorded on the forked chain are eventually nullified, when the consensus bug is fixed and the fork is abandoned [23,51]. Second, an attacker can steal ETH from certain vulnerable smart contracts on buggy client nodes. Even if the business logic of a smart contract is perfectly secure, a consensus bug can alter how the buggy client executes the contract and allow the theft of ETH.

Attackers have strong incentives to find and exploit consensus bugs. Attackers can short ETH on cryptocurrency exchanges after triggering a consensus bug, with the expectation that the price of ETH will fall, when investors learn about the attack and lose trust in Ethereum. Attackers also can trade their ETH with an off-chain item (e.g., other cryptocurrency such as Bitcoin [39]) on the forked chain after triggering the bug, such that the traded ETH is given back to them when the bug is fixed and the forked chain is abandoned. Finally, attackers can steal ETH from vulnerable smart contracts.

3 Overview

We describe Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. Unlike existing fuzzers which use the blockchain state model, Fluffy fully covers the search space for finding consensus bugs, by modelling an Ethereum client as a client program state model. Using this client program state model where the client program state is transitioned by a transaction, Fluffy tests a sequence of multiple transactions in each iteration. In addition, Fluffy uses different Ethereum clients as cross-referencing oracles.

Figure 2 illustrates an overview of Fluffy. First, Fluffy

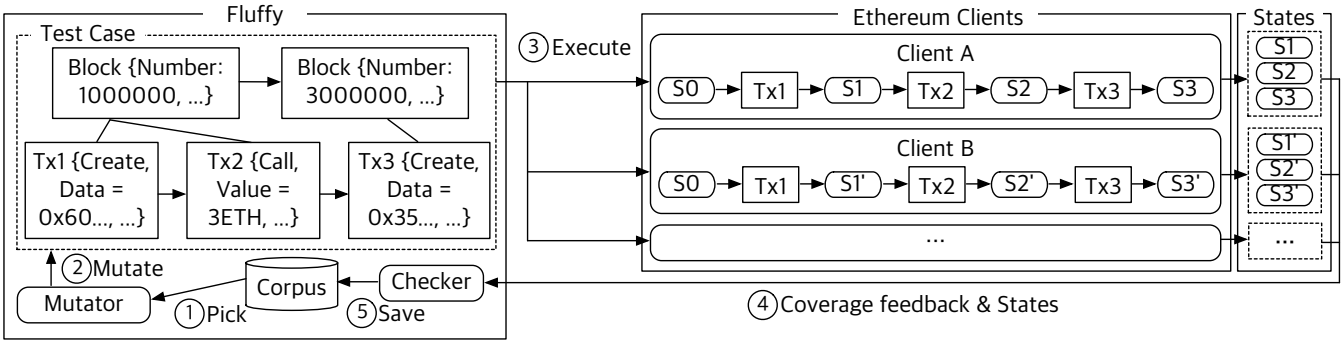


Figure 2: An overview of Fluffy. Fluffy first selects a test case from the corpus (①). Next, Fluffy mutates transactions in the test case using a semantic-aware mutation strategy (②). Fluffy then executes the new test case on multiple Ethereum clients (③). The clients transition the initial blockchain state to new states, while executing the transactions in the test case. When the execution completes, Fluffy collects the new states and coverage feedback (④). Fluffy saves the new test case if new code paths are discovered (⑤). Fluffy proceeds to the next iteration if the clients transitioned to the same states, and crashes otherwise.

selects a test case from the corpus of previously executed test cases (①). Each test case contains multiple transactions, and information about dependencies between the transactions. Fluffy then generates a new test case by mutating the transactions in the selected test case using a semantic-aware mutation strategy (②). Fluffy then executes the new test case on multiple Ethereum clients (③). The clients transition the initial blockchain state (S0) to new states (Client A: (S1, S2, S3), Client B: (S1′, S2′, S3′)), while executing the transactions (Tx1, Tx2, Tx3) in the test case. When the execution completes, Fluffy collects the new blockchain states and code coverage feedback from the clients (④). Fluffy saves the new test case in the corpus if new code paths are discovered (⑤). Fluffy cross-checks the blockchain states collected from different clients, and crashes if the clients transitioned to a different state during execution (S1 != S1′ || S2 != S2′ || S3 != S3′). Otherwise, Fluffy proceeds to the next fuzzing iteration.

4 Design

We describe the design of Fluffy, focusing on the execution model, the test case, and the mutation algorithm.

4.1 Execution Model

We modify existing Ethereum clients to provide an execution model that enables efficient multi-transaction fuzzing.

Genesis account. We modify clients to use an initial blockchain state that contains the genesis account, which we define as an EOA that has a large balance of ETH. The genesis account serves as the starting point for creating new smart contracts and invoking the code of the contracts (i.e., we set the sender of transactions to the genesis account).

Activated addresses. We modify clients to convert 20-byte Ethereum addresses to activated addresses, which we define as an address either owned by a precompiled contract, or owned by a contract created in previous EVM execution. The rationale is that it is extremely inefficient to explore the 20-byte Ethereum address space, especially given that we rewrite blockchain history and almost all of the addresses are not used by a contract. Moreover, we cannot know in advance which addresses will be activated and used by smart contracts in the future, since new contracts can be created dynamically during the execution of EVM bytecode (`CREATE`).

4.2 Test Case

We use test cases that are tailored to multi-transaction fuzzing. Figure 3 shows the data structure of test cases used in Fluffy. We execute test cases on Ethereum clients through iterating over the blocks and the transactions of each block in the list order, and applying each transaction to the blockchain state. Our design has several unique characteristics that enable efficient multi-transaction fuzzing.

First, we enable mutating the context of transactions, which we define as the ordered sequence of transactions that are executed prior to the transaction. Our approach is in contrast to existing approaches [18, 22, 33] that directly generate a pre-transaction blockchain state and test a single transaction. The blockchain state mutation strategy of existing approaches are limited to testing only a single pre-transaction client program state for each pre-transaction blockchain state. In contrast, our approach is able to generate and test various pre-transaction client program states (e.g., (account_a = { ETH: 0, deleted: false}), (account_a = { ETH: 3, deleted: true})) for each pre-transaction blockchain state (e.g., Account A has 0 ETH). This is because the values of program variables of Ethereum clients can change in various ways depending on which sequence of transactions are executed. This lets us find bugs like the transfer-after-destruct bug (§ 6.2), which requires testing particular pre-transaction client program states that the

```
class FluffyTestCase:
 Block[] blocks

class Block:
 Transaction[] transactions
 int versionNumber    // hard-fork upgrades
 int timestamp        // between prev/next block
 // Constants: author, gasLimit, ...

class Transaction:
 int gasLimit         // minimum to threshold
 int value            // 0, 1, or random
 byte[] data          // bytes
 // Constants: signature, gasPrice, ...

class CreateContract extends Transaction:
 byte[] constructor   // invoked bytecode
 byte[] codeToReturn  // code of new contract

class MessageCall extends Transaction:
 int receiver         // activated address
```

Figure 3: The data structure of test cases used in Fluffy.

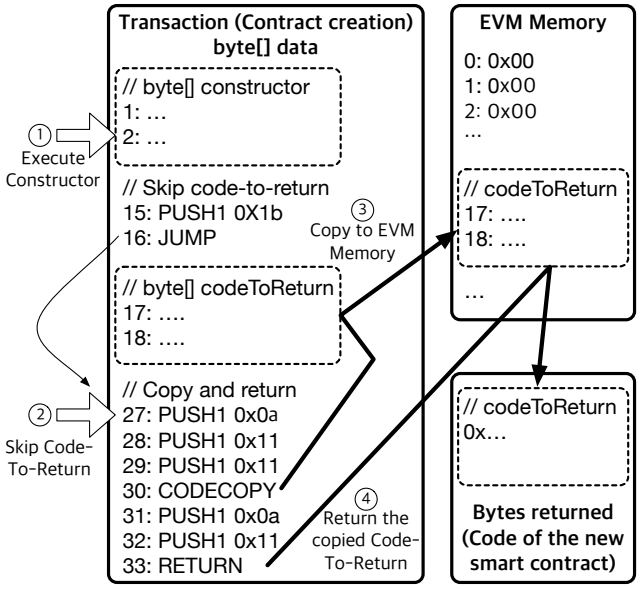


Figure 4: Fluffy mutates the data field of contract creation transaction by mutating the constructor field and the code-to-return field that become part of the data field along with a number of injected bytecode instructions. When the transactions are executed, the constructor is executed and the code-to-return is returned.

blockchain state mutation strategy is unable to generate.

Second, we introduce the constructor and the code-to-return fields for contract creation transactions. These fields, along with a number of injected instructions which we describe later, become part of the data field. Our approach enables directly mutating the code of the newly created smart contracts, which is set to the code-to-return when the transaction completes. Existing Ethereum fuzzers [18, 22, 33] do not consider this approach, since they execute a single transaction per fuzzing iteration and do not invoke the code of smart contracts created by transactions.

Third, we limit the possible values of transactions and block parameters to reduce wasting CPU cycles in meaningless mutations and executions. We use constant values for parameters that have limited effects on how clients execute transactions. Our approach reduces the overhead of mutating and executing multiple transactions.

4.3 Mutation

We use three mutation strategies to mutate test cases: context mutation, bytecode mutation, and parameter mutation. Our bytecode and context mutation strategies have not been employed in existing Ethereum fuzzers [18, 22, 33]. Minor parts of the parameter mutation strategy, such as setting the timestamp of a block within a certain range, share some similarities with existing fuzzers.

Context mutation. We randomly mutate the list of blocks and the list of transactions to mutate transaction contexts. We use four strategies: add, delete, clone, copy. We add a new block or a new transaction to the list, or delete an existing one. We also clone an existing block or a transaction, or copy its contents to another block or transaction.

Bytecode mutation. We mutate the constructor and the code-to-return fields of contract creation transactions to mutate bytecode. Specifically, we randomly add, delete, mutate, and copy bytecode instructions in the fields. Among the various EVM instructions, we do not add the `PUSH` instructions (`PUSH1-PUSH32`), which make EVM push a number of following bytes (1B-32B) in the fields onto the EVM stack, rather than interpreting the bytes as bytecode instructions and executing them. Our approach enables preserving the semantics of bytecode instructions that are not directly modified by Fluffy across mutations.

We then update the data field using the mutated constructor and code-to-return, as illustrated by Figure 4. We concatenate the constructor, instructions to skip the execution of the code-to-return (`JUMP`), the code-to-return, instructions to copy the code-to-return to the EVM memory (`CODECOPY`), and instructions to return the copied bytecode (`RETURN`). When the transaction is executed and the bytecode of the data field is invoked, the constructor is executed and the code-to-return is returned.

In contrast, it is difficult to generate smart contract constructors that return appropriate bytecode, if we treat the data field as a single sequence of instructions and mutate the data field as a whole. The reason is that the generated constructor is likely to prematurely terminate before invoking `RETURN`

to return bytecode. Moreover, appropriate bytes should be stored in the right region of the EVM memory before invoking `RETURN`, and a small mutation is likely to completely alter the stored bytes.

Nevertheless, we do note that in Fluffy the code of a smart contract is not always equal to the code-to-return field of the transaction that creates the contract. This is because errors such as EVM stack underflow can still occur during the execution of the constructor field of the transaction, and prevent the following injected instructions to copy and return the code-to-return.

Parameter mutation. We mutate transaction parameters as follows. We simply set the transaction receiver address to a random integer, assuming that the target clients convert it to an activated address. We set the gas limit to the sum of the minimum gas required for EVM to not reject the transaction before invoking any bytecode, and a randomly generated number in the range between 0 to a threshold to avoid long sequences of meaningless instructions such as an infinite while loop. For example, we can set the threshold to 1.6 million gas that allows executing the `CREATE` instruction 50 times, which is the most expensive instruction that costs 3.2 thousand gas. 1.6 million gas costs only around 0.8 ETH on the Ethereum mainnet as of August 2020. In case of value, which determines the amount of ETH transferred by the transaction, we randomly choose 0, 1, or a random integer.

We also randomly mutate the parameters of blocks. Most notably, we mutate the block version number, which determines the version of EVM that executes the transaction. Since Ethereum launched in 2014, there has been around 10 non-backward compatible EVM hard-fork upgrades that came into effect at particular block version numbers. We use the version numbers that mark the start of a new EVM hard-fork upgrade, rather than covering all of the block version numbers used in mainnet, which are more than 10 million as of August 2020.

5 Implementation

We implement Fluffy on top of libFuzzer [30] using Rust and Go. We adopt the basic infrastructure of libFuzzer, including the code coverage bitmap and test case scheduling, but introduce several key components. We replace the default mutator of libFuzzer with our multi-transaction mutator. We also introduce fuzzing harnesses for OpenEthereum and Geth to enable in-process fuzzing and several other optimizations. Finally, we implement a crash debugger for analyzing crashes due to consensus bugs. The rest of the section describes notable implementation details.

5.1 Fuzzing Harnesses

We implement fuzzing harnesses as long-running processes that integrate with the transaction processing components of Ethereum clients.

In-process fuzzing. We reuse fuzzing harnesses across fuzzing iterations to avoid having to spawn new Ethereum client processes for every new test. We link the mutator with the OpenEthereum harness to mutate test cases and collect code coverage statistics in the same process, and run the Geth harness in a separate process. We use Linux FIFO files for exchanging test cases and execution results between the two processes.

Initial blockchain state. In addition to the genesis account, we add to the initial blockchain state accounts with a balance of 1 Wei (1×10^{-18} ETH) under the addresses of precompiled contracts. These non-zero balance accounts let us avoid triggering a false positive consensus bug in Geth related to a bug that was previously exploited in the live Ethereum mainnet (Bug #15 in Table 2) [51].

Activated addresses. We maintain a list of activated addresses in the harnesses. While executing transactions, we add addresses of newly created contracts to the list and convert Ethereum addresses to activated addresses (i.e., activatedList[bigInteger(address) % activatedList.length()]). To test deleted addresses, we do not remove addresses from the list when contracts invoke `SELFDESTRUCT` and destroy themselves.

Transaction verification. Ethereum clients use the secp256k1 ECDSA algorithm to verify the signature of transactions [54]. This requires signing and verifying each of the many transactions that Fluffy generates, which is costly considering that most of EVM bytecode instructions consume few CPU cycles. We skip these procedures in our harnesses, since we do not focus on signature verification.

Jump destinations. EVM throws an error if the destination of `JUMP` and `JUMPI` is not `JUMPDEST`, which marks a valid jump target. This increases the chance of Fluffy terminating prematurely due to an error when testing loops and conditional branches. To address this issue, we disable the checking of `JUMPDEST` and allow jumping to non-`JUMPDEST` instructions in our harnesses.

Number of transactions. Fluffy uses its mutator and the test case scheduler to determine the number of transactions it should use per test case. Fluffy randomly generates test cases with few transactions to build the initial corpus. If new code paths are not easily discovered with few transactions, Fluffy gradually generates test cases with more transactions through the transaction context mutations, adding the new test cases to the corpus if they discover new code paths. Fluffy provides an option to configure a libFuzzer parameter (-len_control), which determines whether to prefer to generate small test cases over large test cases. Fluffy also provides an option to set a hard limit on the number of transactions in a fuzzing iteration. We note that the search space of Fluffy, which is the Ethereum client program state model, is constant regardless of the number of transactions that Fluffy executes for each test case.

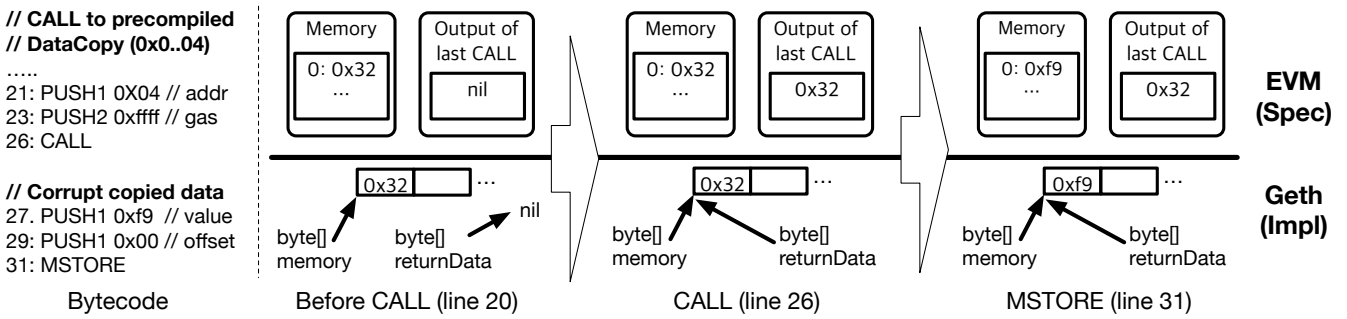


Figure 5: A minimal test case for the shallow copy bug in Geth. An attacker can exploit this bug to corrupt data copied through the precompiled DataCopy contract, making Geth deviate from the EVM specification.

5.2 Crash Debugger

The crash debugger enables analyzing crashes due to consensus bugs.

Analyzing the root cause. We find the first states that the clients output differently while processing the test case, which we call triggering states. We also find the last state that can reach the triggering states when used as the starting point of EVM execution, which we call the starting state. We then find which EVM bytecode instruction invoked during the execution of transactions between the starting state and the triggering states cause different behaviors in Ethereum clients. Finally, we use tools like Delve [14] on corresponding code in Ethereum clients to analyze the root cause.

Validating exploitability. We compare the latest blockchain state in the Ethereum mainnet with the starting state of the bug. We check whether an Ethereum user can transition the latest blockchain state to a new state that includes the accounts in the starting state. We also check whether the transactions between the starting state and the triggering states are processed by the latest version of EVM used in the mainnet. Finally, we convert the transactions into new transactions that can reproduce the bug on vanilla Ethereum clients. In particular, we convert active addresses to Ethereum addresses through examining EVM traces [19,41], and insert `JUMPDEST`s where appropriate.

6 New Consensus Bugs

Fluffy found two new consensus bugs in Geth which were exploitable on the live Ethereum mainnet: shallow copy bug and transfer-after-destruct bug. Existing ethereum fuzzers [18, 22, 33] that test only a single transaction per iteration are not able to find the transfer-after-destruct bug, because finding it requires testing particular pre-transaction client program states which the fuzzers are unable to generate. Although the shallow copy bug can be found by testing a single transaction, existing fuzzers failed to reach deep states of Ethereum clients and failed to find the bug during the time from when the bug became exploitable in the live Ethereum mainnet in November 2019 (Geth v1.9.7 release) to when we found and reported the bug in July 2020 [43]. In this section we describe the bugs using minimal test cases, and discuss the impact of the bugs. We also explain how the bugs were reported, fixed, and triggered, with a focus on vulnerability disclosure issues that occurred.

6.1 Shallow Copy Bug

The root cause of this bug is that the implementation of the precompiled DataCopy contract (address: 0x0..04) in Geth performs a shallow copy upon invocation, although the contract should perform a deep copy according to the EVM specification.

Figure 5 shows a minimal test case that triggers the bug. Suppose that a message call transaction is issued to a contract account that contains bytecode instructions shown in the figure. The figure shows the inner workings of the EVM specification (top) as well as the Geth implementation (bottom) when processing the bytecode of the contract invoked by the transaction. Geth implements the EVM memory and the output of the last `CALL` to external contracts with `byte[]memory` and `byte[]returnData` respectively, which are pointers to a byte buffer.

The following steps trigger the bug. Between line 1 to line 20, the contract stores a byte 0x32 in the EVM memory at offset 0. Geth carries out the execution by storing the byte 0x32 in the byte buffer that `byte[]memory` points to. At this point `byte[]returnData` is nil, since no `CALL` has been made from the contract yet.

Next, the contract `CALL`s the DataCopy contract at address 0x0..04, passing in the 1-byte data at the EVM memory offset 0 as the argument. This leads to the execution of the DataCopy implementation in Geth, which is a single line of code that simply returns the `byte[]` pointer that is given to it. In this case the pointer points to the 1-byte data in the byte buffer that `byte[]memory` is pointing to. Geth then sets `byte[]returnData` to this pointer.

The contract corrupts the copied data by simply storing

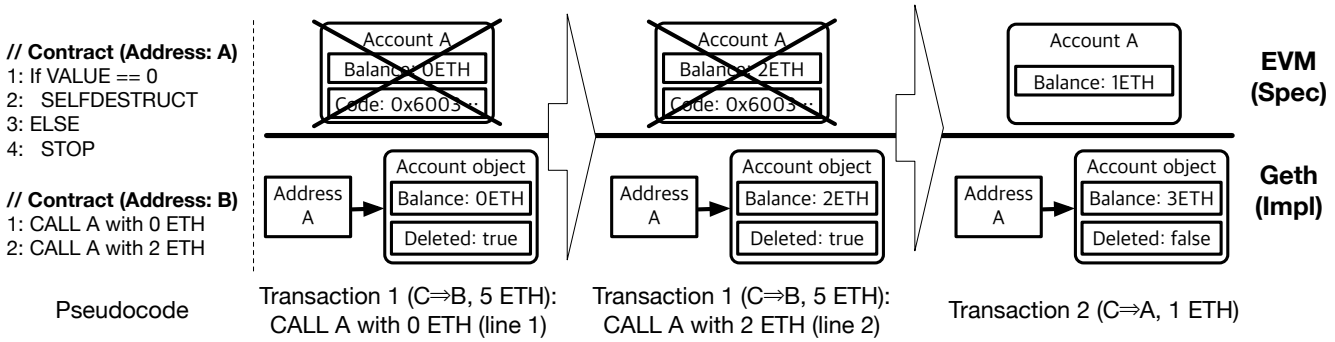


Figure 6: A minimal test case for the transfer-after-destruct bug in Geth. Transaction 1 invokes B, which leads to two CALLs to A. Transaction 2 invokes A. An attacker can exploit this bug to carry over the balance of a deleted account to a new account under the same address, making Geth deviate from the EVM specification.

new data, 0xf9, in the EVM memory at offset 0 (MSTORE). According to the specification, storing data in the EVM memory should never affect the data copied through DataCopy. However, in Geth, the contents of byte[]returnData changes from 0x32 to 0xf9.

Now, the contract can corrupt the blockchain state, for example through a sequence of bytecode instructions that stores the corrupted data in the storage of the contract (RETURNDATACOPY, MLOAD, SSTORE). After the transaction, Geth expects that 0xf9 is stored in the storage, whereas OpenEthereum and all other clients that comply with the specification expect 0x32.

Although this shallow copy bug is conceptually simple to understand, it is difficult to detect through code reviews, and tools such as static checkers which the Ethereum developers are actively using. The reason is that in the actual Go code of Geth it is not straightforward to track how pointers move across multiple files, classes, and functions.

Impact. When exploited, this bug can trigger a network split where Geth client nodes create a fork of the blockchain that stores 0xf9 in the storage. Furthermore, this bug can be exploited for smart contract theft. For example, suppose a contract invokes DataCopy by passing in an Ethereum address, overwrites the address with user input, and then transfers ETH to the copied address returned from DataCopy. Attackers can withdraw ETH from this contract to their account by specifying the address of their account as the user input, while others believe that the result of DataCopy should always be equal to the original address. Existing techniques for finding smart contract vulnerabilities [26, 32, 40, 49, 50, 57] are not capable of finding such vulnerability because they assume that the underlying Ethereum clients faithfully carry out the semantics of EVM bytecode instructions and precompiled contracts.

6.2 Transfer-After-Destruct Bug

The root cause of this bug is that Geth carries over the balance of a deleted account object to the newly created account object under the same Ethereum address, although it should not according to the EVM specification.

Figure 6 shows a minimal test case that triggers this bug. The initial blockchain state consists of two contracts. If the value of the transaction that invokes the contract is 0 ETH, the contract under address A destroys itself by invoking SELFDESTRUCT. If not, contract A simply terminates execution with STOP. The contract under address B issues two CALLs to A. To trigger the bug, we send a transaction to B, and then a transaction to A. The figure illustrates how the EVM specification (top) and the Geth implementation (bottom) handle the two transactions.

The first transaction (Transaction 1) is a message call transaction sent to B. The code of contract B is then executed as follows. First, we CALL A with 0 ETH (Contract B, line 1), which results in destroying contract A with SELFDESTRUCT. Geth carries out SELFDESTRUCT by marking the account object under address A as deleted, rather than destroying the account object as a whole. Geth also sets the balance of the account object to 0 ETH. Next, we CALL A with 2 ETH (Contract B, line 2). This makes Geth look up the account object under address A, and add 2 ETH to the balance of the object.

When the first transaction finishes, Geth transitions to a blockchain state where the account under address A is nil, through recognizing that the account object is marked as deleted and ignoring the balance of 2 ETH. This lets Geth comply with the EVM specification, which specifies that all information associated with the addresses of SELFDESTRUCTed accounts should become nil after a transaction is processed [54].

However, Geth fails to comply with the EVM specification when processing the second transaction (Transaction 2). The second transaction is a message call transaction sent to A with 1 ETH. According to the specification, the balance of the account under address A should become 1 ETH after this transaction, since the balance of the account was nil, and thus was 0 ETH before the transaction. However, Geth mistakenly thinks that the account has 3 ETH after the transaction. When

processing the transaction, Geth does recognize that the account object under address A is marked as deleted. Geth thus attempts to replace the old object with a new account object, but mistakenly carries over the balance of the old object to that of the new object during the process. This results in adding to the balance of the new account object 2 ETH from the old object, as well as 1 ETH from the transaction.

Impact. Similar to the shallow copy bug, this bug can be exploited for network split and theft. Moreover, this bug makes the total supply of ETH in circulation inconsistent between Geth and other Ethereum clients, which adds to the argument that the total supply of Ethereum is impossible to calculate [1, 53].

6.3 Responsible Vulnerability Disclosure

Responsible vulnerability disclosure in cryptocurrencies is hard because decentralized systems give no single party authority to push code updates [7]. To ensure that Ethereum mainnet nodes update securely, we privately reported the bugs to the Geth developers through the Ethereum bug bounty program [21]. Geth developers confirmed that the bugs are exploitable on the live Ethereum mainnet, and silently fixed the bugs in new versions of Geth to reduce the risk of an attacker exploiting the bugs. Ethereum mainnet nodes upgraded organically over time, thereby fixing the bugs.

Unfortunately, not all mainnet nodes upgraded, and this caused nodes using Geth v1.9.7 to v1.9.16 to hard fork the Ethereum block chain when the shallow copy bug was triggered four months later, on November 11th, 2020 [2, 11, 15, 43, 45]. Affected Ethereum infrastructure services and decentralized applications (DApps) went down, and cryptocurrency exchanges halted the trading of ETH. Around 30 Ethereum blocks from block 11234873 on the forked chain were lost [23], which transferred approximately 8.6 million USD worth of ETH. The blockchain community considers this hard fork the greatest challenge since the infamous DAO hack of 2016 [11, 13].

The hard fork sparked an active discussion on vulnerability disclosure protocols [11, 43, 52]. As a result, the Geth team created a public transparency policy for disclosing bugs [19]. The Geth team also revealed security advisories, including an advisory on the shallow copy bug (CVE-2020-26241) [35].

7 Evaluation

We evaluate Fluffy to answer the following questions.

- Does Fluffy effectively find real-world consensus bugs in Ethereum? (§ 7.1)
- Does Fluffy cover deep code paths that lead to consensus bugs in Ethereum clients? (§ 7.2)
- Does Fluffy efficiently test many instances of multiple transactions that rewrite blockchain history? (§ 7.3)
- Does Fluffy enable analyzing crashes triggered by consensus bugs? (§ 7.4)

We evaluate Fluffy on Intel(R) Xeon(R) CPU E5-2680 v3 (12 cores) with 128 GB memory. We compare Fluffy with EVMLab [18], which is an open-source, state-of-the-art differential fuzzer that is maintained by Ethereum developers. EVMLab is also the only existing fuzzer that found a high-impact consensus bug that would have been exploitable on the live Ethereum mainnet [34].

Unless noted otherwise, we configure the fuzzers as follows. For Fluffy, we configure libFuzzer parameters to run 24 parallel fuzzing instances (-fork=24), and prefer generating and executing small inputs rather than large inputs (-len_control=100). For EVMLab, we run 24 instances of the fuzzer in parallel. We run the fuzzers without any seed corpus for each experiment.

We use OpenEthereum v3.0.0 and Geth v1.9.14 as the target programs. EVMLab uses the vanilla version of the Ethereum clients. Fluffy uses the modified version that also fixes the two new bugs Fluffy found, since without the bug fixes Fluffy crashes due to the bugs during experiments.

7.1 Bug Finding Capability

We measure the time it takes for Fluffy to find the consensus bugs that occurred in Geth and OpenEthereum including the two new bugs Fluffy found, which are listed in Table 2. For each bug, we port the bug to OpenEthereum v3.0.0 or Geth v1.9.14, run Fluffy for 12 hours, and check if Fluffy finds the bug. We do not experiment with Bug #3 and Bug #5 which are associated with block mining and signature verification that Fluffy and existing fuzzers for Ethereum [18, 22, 33] do not focus on, Bug #6 which was fixed by switching to a different external library, and Bug #14 whose details are undisclosed [34].

Table 2 presents the result of the experiment. Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. Among the 10 bugs, Fluffy finds Bug #7 and Bug #10 with a configuration that bounds the number of transactions and the length of the data of transactions, and finds Bug #4 with an earlier implementation of the multi-transaction mutator. The only bug Fluffy fails to find within 12 hours is Bug #9, which was originally found with manual auditing.

Result. Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. The result shows that Fluffy is able to effectively find consensus bugs in Ethereum.

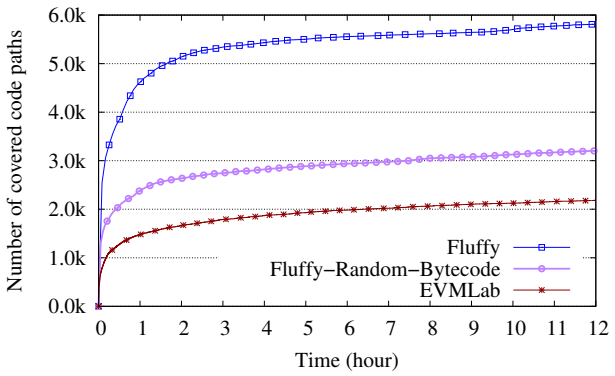


Figure 7: The number of code paths covered by Fluffy, a modified version of Fluffy called Fluffy-Random-Bytecode, and EVMLab over time.

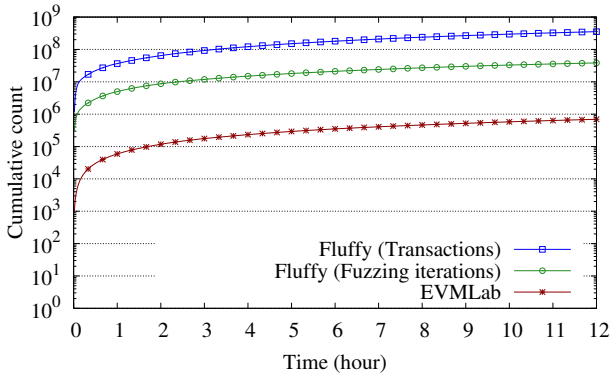


Figure 8: The total number of transactions and fuzzing iterations processed by Fluffy and EVMLab over time. Fluffy processes a varying number of transactions across iterations, whereas EVMLab processes 1 transaction per iteration.

7.2 Code Coverage

We measure the number of code paths covered by Fluffy, a modified version of Fluffy called Fluffy-Random-Bytecode, and EVMLab. Fluffy-Random-Bytecode simply generates random bytecode instructions rather than using the sophisticated bytecode mutation strategy (§ 4.3) of Fluffy. In case of Fluffy and Fluffy-Random-Bytecode, we measure the size of the corpus, which represents the number of code paths, over time. In case of EVMLab, which does not use and report code coverage, we replay the corpus it generates using libFuzzer on OpenEthereum.

Figure 7 shows the covered code paths over time. The result shows that Fluffy covers more code paths than EVMLab at all times. In the first 1 hour, Fluffy quickly covers more than 4,000 code paths, whereas EVMLab covers less than 2,000 paths. After 12 hours, Fluffy covers 5,809 code paths, which is 2.7 times as many code paths as 2,185 code paths that EVMLab covers.

Fluffy-Random-Bytecode performs better than EVMLab, but worse than Fluffy at all times. After 12 hours, Fluffy-Random-Bytecode covers 3,202 code paths, which is close to half of the paths covered by Fluffy and 1.5 times as many code paths as those covered by EVMLab. This show that the bytecode mutation strategy of Fluffy contributes significantly to the effectiveness of Fluffy.

Result. Fluffy explores 2.7 times as many code paths as EVMLab. The result shows that Fluffy is able to cover deep code paths in Ethereum clients that lead to consensus bugs.

7.3 Throughput

We evaluate whether Fluffy efficiently tests many instances of multiple transactions. For this evaluation we measure metrics such as the number of processed transactions, the number of executed fuzzing iterations, and the CPU usage.

Figure 8 shows the total number of Ethereum transactions and fuzzing iterations processed by Fluffy and EVMLab over time. Fluffy processes a varying number of transactions across iterations, whereas EVMLab processes exactly 1 transaction in every iteration. After 12 hours, Fluffy processes more than 350 million transactions and more than 38 million fuzzing iterations. On average, Fluffy processes 9.2 transactions per fuzzing iteration. In contrast, EVMLab processes less than 700 thousand transactions and fuzzing iterations after 12 hours. In total, Fluffy processes 510 times as many transactions and 55 times as many iterations as EVMLab.

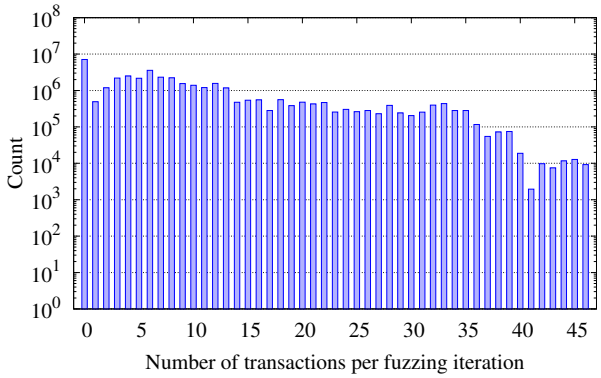


Figure 9: The distribution of the number of transactions that Fluffy processes per fuzzing iteration in 12 hours.

We then examine the number of transactions that Fluffy processes in each fuzzing iteration. Figure 9 shows the distribution of the number of transactions. 3 to 8 transactions are processed most frequently, except for the test cases with zero transaction. The largest number of transactions that Fluffy tests in an iteration for this specific 12-hour fuzzing experiment is 46. As a comparison, the number of transactions that have produced the blockchain state in the live Ethereum mainnet is more than 800 million transactions, as of August 2020. Therefore, this result shows that Fluffy tests Ethereum clients

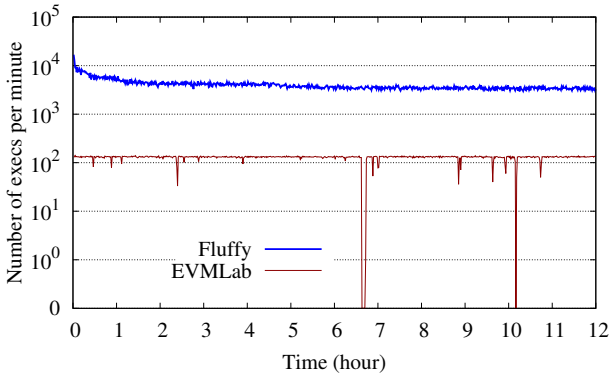


Figure 10: The number of fuzzing iterations executed per minute over time. We run the fuzzers in sequential mode without parallel fuzzing for this experiment.

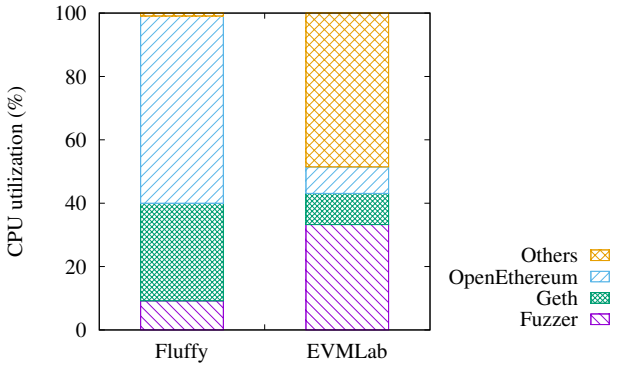


Figure 11: The CPU usage breakdown when running Fluffy and EVMLab.

using a small number of transactions that rewrite blockchain history.

Although the largest number of transactions is 46 for this experiment, Fluffy can test more or fewer number of transactions in a different experiment that runs for the same 12 hours, since how Fluffy determines the number of transactions is nondeterministic (§ 5.1). Furthermore, there can be consensus bugs that require more than 46 transactions to trigger.

Next, we examine the throughput of a single fuzzing instance through running Fluffy and EVMLab in sequential execution mode without parallel fuzzing for 12 hours. Figure 10 shows the number of fuzzing iterations executed per minute over time.

Fluffy achieves and sustains an order of magnitude higher throughput compared to EVMLab. In the beginning, Fluffy executes up to 16,805 fuzzing iterations per minute. We observe that in the beginning Fluffy generates and executes transactions that invoke a small number of bytecode instructions, which allows Fluffy to quickly execute more iterations. The throughput decreases gradually over time, as Fluffy discovers new inputs that invoke many bytecode instructions and executes mutations of those inputs. After 12 hours, Fluffy executes around 3,500 fuzzing iterations per minute.

In contrast, the throughput of EVMLab is overall flat, but fluctuates wildly at certain times during fuzzing. In particular, EVMLab fails to complete a fuzzing iteration for more than a minute between 6 and 7 hours, and 10 and 11 hours after fuzzing. We observe that such fluctuations occur when EVMLab is stuck in processing an excessively long sequence of bytecode instructions. Fluffy does not experience this, because Fluffy limits the number of bytecode instructions that are executed through limiting the gas limit of transactions.

We now measure the CPU usage to analyze why Fluffy achieves higher throughput, as both Fluffy and EVMLab are CPU-bound. Figure 11 shows the CPU usage breakdown when running Fluffy and EVMLab. We obtain the numbers through running the Linux perf command for one minute while running each fuzzer.

In case of Fluffy, most of the CPU time is spent in executing the code of OpenEthereum and Geth. We also observe that the majority of the CPU time spent in the clients is used to execute the core EVM interpreter logic, where most of the consensus bugs have been found.

In contrast, EVMLab suffers from the overhead of executing its own code, which is written in Python, and handling other tasks. We observe that much of the other tasks are associated with Docker [37], although EVMLab reuses Docker containers when spawning new OpenEthereum and Geth processes to test new inputs. We also observe that a large portion of the CPU time for executing the clients is spent in parsing test inputs and initializing the clients, where consensus bugs are unlikely to be found.

We emphasize that Fluffy does not outperform EVMLab by simply using a more efficient programming language and using specific configurations. In Figure 11, excluding the overhead of the fuzzer code (Fuzzer) and other tasks (Others), the CPU used for the Ethereum clients (OpenEthereum and Geth) is 18.2% for EVMLab and 89.9% for Fluffy, which is 4.9×. However, Fluffy processes 510× transactions and 55× fuzzing iterations compared to EVMLab, which is much larger than 4.9×. This shows that Fluffy executes Ethereum clients much more efficiently, even if the overhead external to Ethereum clients is the same as EVMLab.

Result. Fluffy processes 510 times and 55 times as many transactions and fuzzing iterations as EVMLab. The result shows that Fluffy efficiently tests many instances of multiple transactions that rewrite blockchain history.

7.4 Debugging

We used the crash debugger of Fluffy to analyze crashes. We encountered a false positive bug, when we set the initial blockchain state to a state that contains only the genesis account. We analyzed that the latest mainnet blockchain state

cannot transition to a state that contains the accounts in the starting state of the bug, because the starting state contains a zero-balance account under the address of a precompiled contract. Creating such account was possible only before a previous EVM upgrade (related to Bug #15 in Table 2). To avoid triggering this false positive bug, we added to the initial blockchain state non-zero balance accounts under the addresses of precompiled contracts. We also used the crash debugger to analyze the shallow copy bug and the transfer-after-destruct bug, and create minimal test cases.

Result. Fluffy enables analyzing crashes triggered by consensus bugs.

8 Discussion and Limitations

We discuss future research directions, and limitations of the current design of Fluffy.

Smart contract blockchains. Our idea of multi-transaction differential fuzzing can be applied to other blockchains that provide smart contract capabilities like Ethereum. Smart contract blockchains have a total market capitalization of 95 billion USD, as of December 2020, and include popular blockchains such as Ethereum, Cardano, Stellar, EOS, Tron, Tezos, and Neo [12]. Like Ethereum, multiple depending transactions which create and invoke smart contracts determine the transitions of blockchain states in these blockchains. Therefore, techniques of Fluffy such as multi-transaction test cases and semantic-aware mutation strategy can be applied to find consensus bugs in these other blockchains.

Many-client fuzzing. Another research direction is to fuzz multiple versions of many Ethereum clients in addition to a single version of OpenEthereum (Rust) and Geth (Go). Although the two clients are used by 98% of nodes participating in the Ethereum mainnet, as of August 2020 [6], the Ethereum community is becoming more aware of the benefits of using multiple different clients since the shallow copy bug we reported was triggered in the live mainnet [15]. Examples of other Ethereum clients are Aleth (C++), Trinity (Python), Besu (Java), Nethermind (.NET), and EthereumJS (Javascript) [16]. Moreover, it is worthwhile to fuzz not only the latest version of the clients but also previous versions, since many of the decentralized Ethereum nodes in the mainnet do not immediately upgrade when new versions are released, and keep on using a previous version [6]. While it is straightforward to extend the current implementation of Fluffy to fuzz many clients, it remains a challenge to achieve high fuzzing throughput while executing multiple transactions on many clients.

Mutating client program states. Fluffy models an Ethereum client as a client program state model, rather than an EVM state model. Nevertheless, Fluffy mutates the Ethereum client program state indirectly through setting the initial program state to a corresponding initial EVM state and executing multiple transactions. This is because, like other fuzzers, Fluffy does not directly mutate internal states of the target programs. An alternative approach is to directly mutate client program states. However, it would be challenging to directly generate valid client program states that are reachable with transactions, such that the found bugs are exploitable on the Ethereum mainnet.

Limitations of differential fuzzing. Similar to existing differential fuzzers for Ethereum, Fluffy is unable to find bugs when the Etherum clients transition to the same incorrect blockchain state due to the same consensus bug. A practical solution to this limitation is to fuzz many different versions of different client implementations, since it is unlikely that the same bug exists in all of these different clients. A more fundamental solution is to utilize the EVM specification itself as an oracle, similar to how Hydra [27] implements an emulator along with a fuzzer. However, this approach reduces the number of input generation and testing as well as requires extensive engineering efforts unlike differential testing. We also note that, to our knowledge, there has been no previous case of the same bug occurring in multiple Ethereum client implementations.

Limitations of fuzzing. Like existing fuzzers for Ethereum, Fluffy inherits the limitations of fuzzing. Although fuzzing is good at exploring code paths with loose branch conditions (e.g., x > 0), fuzzing struggles to drive the target program into paths with tight branch conditions (e.g., x == 0xdeadbeef) [9, 48, 56]. This limitation is demonstrated by Fluffy failing to find Bug #9 in Table 2 within 12 hours, which requires specific inputs that satisfy tight branch conditions to trigger. We can address the limitation by combining fuzzing with concolic execution [9, 25, 48, 56], which interprets target program variables as symbolic variables and uses constraint solving to generate specific inputs that satisfy branch conditions.

9 Related Work

Fluffy is the first multi-transaction differential fuzzer for finding consensus bugs in Ethereum. In this section we describe existing works that are related to Fluffy.

Consensus in blockchains. Consensus in blockchains are increasingly becoming important as blockchains such as Bitcoin [39] and Ethereum [17] are becoming increasingly used. Researchers have proposed various techniques related to consensus in blockchains to improve the scalability, security, and usability of blockchains [4, 24, 28, 29, 31, 44, 47]. Our work complements these works by focusing on the implementation aspects of consensus in blockchains, and finding consensus bugs in Ethereum clients that lead to network split and theft.

Differential testing for consensus bugs. Differential testing is an effective software testing method that has been applied to various systems [3, 8, 10, 36, 42, 55]. Several fuzzers have been proposed to apply differential testing techniques to find consensus bugs in Ethereum [18, 22, 33]. These fuzzers gen-

erate a blockchain state and a transaction that transforms the state. Fluffy is also a differential fuzzer for finding consensus bugs, but Fluffy generates and runs multiple transactions that rewrite blockchain history and adopts various optimizations to improve the fuzzing throughput and the code coverage.

Coverage-guided fuzzing. Coverage-guided fuzzers such as libFuzzer [30] and AFL [38] leverage code path statistics to mutate test inputs. Fluffy extends such coverage-guided fuzzing mechanisms through extending libFuzzer. Leveraging more sophisticated mechanisms like gradient-guided techniques [46] is left as future work.

Smart contract vulnerabilities. Existing techniques for finding smart contract vulnerabilities [26, 32, 40, 49, 50, 57] focus on vulnerabilities in the business logic of smart contracts and transactions, whereas Fluffy focuses on vulnerabilities in the underlying Ethereum client implementations. For example, TxSpector [57] replays transaction history to extract logic relations, and applies user-specific logic rules to uncover vulnerabilities such as the re-entrancy vulnerability. In contrast, Fluffy generates and tests transactions which have never occurred in blockchain history to trigger consensus bugs in Ethereum clients that alter how the vulnerable clients execute the business logic of smart contracts.

10 Conclusion

Consensus bugs in Ethereum are extremely rare but can be exploited for network split and theft, which cause reliability and security-critical issues in the Ethereum ecosystem. Our fuzzer, called Fluffy, shows how to find consensus bugs hidden in deep states of Ethereum clients. Unlike existing fuzzers for Ethereum, Fluffy supports multi-transaction tests and uses different Ethereum clients as cross-referencing oracles. Fluffy also greatly improves the fuzzing throughput and the code coverage with various optimizations: in-process fuzzing, fuzzing harnesses for Ethereum clients, and semantic-aware multi-transaction mutation that reduces erroneous test cases. Fluffy found two new consensus bugs in the most popular Geth client which were exploitable on the live Ethereum mainnet. Fluffy is publicly available at https://github.com/snuspl/fluffy.

11 Acknowledgements

We thank our shepherd Ding Yuan and the anonymous reviewers for their insightful feedback. We thank the members of the Software Platform Lab at Seoul National University for their valuable input. This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Adriana Hamacher. So, what is the Ethereum (ETH) total supply?, August 2020. https://decrypt.co/38271/so-what-is-the-ethereum-eth-total-supply.

[2] Andrey Shevchenko. Binance briefly pauses Ethereum withdrawals as network suffers 'minor hard-fork', November 2020. https://cointelegraph.com/news/binance-pauses-ethereum-withdrawals-as-network-suffers-minor-hard-fork.

[3] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A Device for Secure Transaction Approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[5] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 1985.

[6] Bitfly. Ethereum Mainnet Statistics. https://ethernodes.org, Accessed August 2020.

[7] Rainer Böhme, Lisa Eckey, Tyler Moore, Neha Narula, Tim Ruffing, and Aviv Zohar. Responsible Vulnerability Disclosure in Cryptocurrencies. *Commun. ACM*, 2020.

[8] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[9] George Candea and Patrice Godefroid. Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances. In *Computing and Software Science - State of the Art and Perspectives*. Springer, 2019.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[11] Colin Harper. Ethereum's 'Unannounced Hard Fork' Was Trying to Prevent the Very Disruption It Caused, November 2020. https://coindesk.com/ethereums-hard-fork-disruption.

[12] CryptoSlate. Smart Contracts Coins: Protocols intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. https://cryptoslate.com/cryptos/smart-contracts, Accessed December 2020.

[13] David Siegel. Understanding The DAO Attack, June 2016. https://coindesk.com/understanding-dao-hack-journalists.

[14] Derek Parker. Delve: A Debugger for the Go Programming Language. https://github.com/go-delve/delve, Accessed August 2020.

[15] Eleazar Galano. Infura Mainnet Outage Post-Mortem 2020-11-11, November 2020. https://blog.infura.io/infura-mainnet-outage-post-mortem-2020-11-11.

[16] Ethereum. Clients, tools, dapp browsers, wallets and other projects. https://github.com/ethereum/wiki/wiki/Clients,-tools,-dapp-browsers,-wallets-and-other-projects, Accessed December 2020.

[17] Ethereum. Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/, Accessed August 2020.

[18] Ethereum. EVM lab utilities: Utilities for interacting with the Ethereum virtual machine. https://github.com/ethereum/evmlab, Accessed August 2020.

[19] Ethereum. Go Ethereum: Official Go implementation of the Ethereum protocol. https://geth.ethereum.org, Accessed August 2020.

[20] Ethereum. Solidity: An object-oriented, high-level language for implementing smart contracts. https://solidity.readthedocs.io/en/develop, Accessed August 2020.

[21] Ethereum. The Ethereum Bounty Program. https://bounty.ethereum.org, Accessed August 2020.

[22] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[23] Geth team. Geth security release: Critical patch for CVE-2020-28362, November 2020. https://blog.ethereum.org/2020/11/12/geth_security_release.

[24] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[26] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium (CSF)*, 2018.

[27] 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)*, 2019.

[28] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[29] 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.

[30] LLVM Project. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, Accessed August 2020.

[31] 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.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[33] Martin Holst Swende. One year of Ethereum Security, November 2017. Devcon 3.

[34] Martin Holst Swende. Protecting The Baselayer - from Shanghai to Osaka, October 2019. Devcon 5.

[35] Martin Holst Swende. Shallow copy in the 0x4 precompile could lead to EVM memory corruption, November 2020. https://github.com/ethereum/go-ethereum/security/advisories/GHSA-69v6-xc2j-r2jf.

[36] William M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 1998.

[37] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, March 2014.

[38] Michał Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl, Accessed August 2020.

[39] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. https://bitcoin.org/bitcoin.pdf.

[40] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[41] OpenEthereum. OpenEthereum: Fast and feature-rich multi-network Ethereum client. https://github.com/openethereum/openethereum, Accessed August 2020.

[42] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Péter Szilágyi. Geth v1.9.17 Post Mortem, November 2020. https://gist.github.com/karalabe/e1891c8a99fdc16c4e60d9713c35401f.

[44] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A High-throughput Blockchain Over Mobile Devices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[45] Scott Chipolina. How a Dormant Bug Briefly Split the Ethereum Blockchain, November 2020. https://decrypt.co/47891/how-a-dormant-bug-briefly-split-the-ethereum-blockchain.

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[47] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High Throughput Cryptocurrency Routing in Payment Channel Networks . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[49] Christof Ferreira Torres, Mathis Steichen, et al. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[50] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[51] Vitalik Buterin. Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2, November 2016. https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2.

[52] William Foxley. Developers Debate Disclosure Protocols After 'Accidental' Ethereum Hard Fork, November 2020. https://coindesk.com/developers-debate-disclosure-protocols-accidental-ethereum-hard-fork.

[53] William Foxley. How Much Ether Is Out There? Ethereum Developers Create New Scripts for Self-Verification, August 2020. https://coindesk.com/how-much-ether-is-out-there-ethereum-developers-create-new-scripts-for-self-verification.

[54] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. https://gavwood.com/paper.pdf.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), 2011.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, 2018.

[57] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *Proceedings of the 29th USENIX Security Symposium (Security)*, 2020.

MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation

Sam Kumar, David E. Culler, and Raluca Ada Popa
University of California, Berkeley

Abstract

Secure Computation (SC) is a family of cryptographic primitives for computing on encrypted data in single-party and multi-party settings. SC is being increasingly adopted by industry for a variety of applications. A significant obstacle to using SC for practical applications is the memory overhead of the underlying cryptography. We develop MAGE, an execution engine for SC that efficiently runs SC computations that do not fit in memory. We observe that, due to their intended security guarantees, SC schemes are inherently *oblivious*—their memory access patterns are independent of the input data. Using this property, MAGE calculates the memory access pattern ahead of time and uses it to produce a memory management plan. This formulation of memory management, which we call *memory programming*, is a generalization of paging that allows MAGE to provide a highly efficient virtual memory abstraction for SC. MAGE outperforms the OS virtual memory system by up to an order of magnitude, and in many cases, runs SC computations that do not fit in memory at nearly the same speed as if the underlying machines had *unbounded physical memory* to fit the entire computation.

1 Introduction

Secure Computation (SC) refers to cryptographic primitives that allow computation on encrypted data. An example of SC is secure multi-party computation, which allows two parties to perform a collaborative computation on private input data. Advances in cryptography over the years have steadily brought SC closer to practice. Recently, the use of SC in industry—in particular, homomorphic encryption (HE) and secure multi-party computation (SMPC)—has burgeoned. Companies offer services based on SC [12,19,27,38,46,75] (from secure collaborative learning to decentralized authentication and custody), large financial enterprises have added SC-based products [64], and cryptocurrencies secure billions of dollars with SC [91].

SC not only has high CPU overhead, but also requires high memory usage and, in the case of SMPC, high network usage. For example, a 64-bit integer, which requires only 8 B of memory when computing in plaintext, takes up 1 KiB of memory when using a garbled circuit (a type of SMPC). Efficiently running SC requires careful attention to not only CPU efficiency, but also memory and network demands.

CPU overheads can be reduced using hardware accelerators (e.g., GPUs, FPGAs) or specialized hardware (e.g., AES-NI). Network bandwidth continues to grow exponentially according to Nielsen's Law [62], and recent cryptographic improvements have relaxed network bandwidth demands for some SC protocols [10, 15]. But memory management remains problematic. Many recent cryptographic systems based on SC report that SC systems quickly run out of memory [66,79,94,95]. Once they do, the computation becomes prohibitively slow because the OS inefficiently swaps the large memory footprint to secondary storage. For example, the authors of Conclave [79] report that Obliv-C, an SMPC framework, can run a join on only 30,000 records before running out of memory, and state that SMPC "in practice only scales to a few thousand input records." Similarly, Senate [66], a secure collaborative analytics engine based on SMPC, can run a 16-party private set intersection on only 10,000 integers per party.

In this context, we address the research question: **Can SC execute efficiently when it does not fit in memory?** We answer this in the affirmative with our system MAGE.[1]

A natural starting point for MAGE is to specialize the OS page replacement policy to SC workloads. Unsurprisingly, this design suffers from some of the same pitfalls as classic virtual memory systems. Pages may not be fetched until a page fault occurs, requiring multiprogramming to keep the CPU busy [26]. Furthermore, classic page replacement algorithms perform poorly on some workloads [3], and a policy specialized to SC would likely be no different.

To mitigate these issues, we observe that SC is inherently *oblivious*. In particular, many SC protocols have *no data-dependent memory accesses*. This is because an SC protocol must not leak any information about the data contents via its memory access pattern. Our key insight in MAGE is that SC's inherent obliviousness allows us to calculate the access pattern for any computation *in advance* and use it to manage memory in a fundamentally more efficient way than classic OS paging. Unlike paging, which typically responds to page faults *reactively*, MAGE can *proactively* produce a memory management plan based on the program's memory access pattern. To highlight this distinction, we call our approach *memory programming* and the resulting plan a *memory program*. MAGE preplans the exact sequence of memory-storage transfers to issue at runtime, given a target memory consumption. Enabled by memory programming and the compute-to-memory ratio of SC workloads, MAGE executes certain SC programs that do not fit in memory at nearly in-memory speeds, as if memory were unbounded—in effect, virtual memory at nearly zero cost.

To understand the power of MAGE's preplanning based on SC's obliviousness, consider Belady's theoretically optimal

[1] MAGE stands for **M**emory-**A**ware **G**arbling **E**ngine.

paging algorithm (MIN) [3]. MIN, being a clairvoyant algorithm, is not realizable in the classic formulation of paging; it is typically used as a point of comparison to other realizable heuristics. But in the context of memory programming, MAGE can use MIN directly, because it knows the access pattern in advance. Memory programming allows MAGE to use an algorithm that is well-grounded in theory, instead of a heuristic (e.g., LRU or LFU) that sometimes performs poorly.

Yet memory programming also raises the bar for possible memory management strategies. For example, although MIN is an optimal paging algorithm, it unfortunately does not produce an optimal memory program. The reason is that MIN, like other paging algorithms, brings a page into memory only at the moment it is needed, thereby causing the program to stall while transferring the page. We can overcome this by leveraging SC's obliviousness once again, to prefetch according to the access pattern (i.e., with no false positives or false negatives) so that the program never stalls.

At its core, our approach to memory management is quite simple: MAGE optimizes storage bandwidth by evicting pages using MIN, and optimizes latency via prefetching and asynchronous eviction. Whereas classic paging algorithms typically rely on heuristics and empirical observations of what works well in practice [9], our memory programming approach is simple, well-grounded, robust, and performant.

While conceptually simple, the above strategy is challenging to instantiate efficiently. The reason is that MIN requires the entire memory access pattern to be materialized at once; it cannot be applied in a streaming fashion. Using Intel Pin [54], we found that an SC workload that runs in under an hour can issue *trillions* of memory accesses. Thus, materializing the access trace could require *terabytes* of space.

To address this, we leverage the strong precedent for using DSLs to specify SC programs [34, 78]. MAGE's planner represents the program as a bytecode recording higher-level operations specified in the DSL program. This is more succinct than recording individual memory accesses. For example, consider a program that adds two integers using garbled circuits, an SMPC protocol. Garbled circuits support only AND and XOR operations on encrypted *bits*, so the integer addition is ultimately decomposed into encrypted AND and XOR operations, each of which comprises many memory accesses. Yet, MAGE records the entire addition operation as a *single* entry in the bytecode. This works well because most of the addition operation's memory accesses are "uninteresting"—they are accesses to temporary variables (e.g., on the stack) that fit easily in memory, or to SC protocol state that should remain in memory for the entire program. The only consequential accesses for memory management—reading the two input integers and writing the output integer—are captured in the single entry MAGE records.

Once MAGE allows SC to efficiently expand beyond the physical memory limit, another limited resource (e.g., storage/network bandwidth or CPUs) of a single machine could become the bottleneck. Thus, we design MAGE to support *parallel* SC execution across multiple network flows, CPU cores, or machines. To do so, we observe that a distributed memory programming model allows SC to be parallelized in this way, without requiring MAGE's planner to reason about threads executing concurrently in the same address space.

Finally, we aim to support a variety of applications and protocols, including new ones that may emerge in the coming years. The challenge is that different SC protocols may be very different cryptographically and may support different operations efficiently. Fortunately, our memory programming approach allows us to build MAGE entirely in userspace on a Linux system, helping to make MAGE *extensible* to new applications and protocols. We carefully design a layered architecture for MAGE so that the DSL, bytecode, and interpreter can be extended for new SC protocols.

We implemented MAGE in C++ and apply it to two SC protocols: (1) garbled circuits, a type of SMPC, and (2) CKKS, a type of HE. We evaluated MAGE using 10 workloads, sized such that they do not fit in memory. MAGE outperforms the operating system's virtual memory for all 10 workloads, and outperforms it by 4–12× for 7 of them. Additionally, MAGE executes all 10 workloads at within 60% of in-memory speeds, and runs 7 of them at within 15% of in-memory speeds.

Even with our techniques, SC remains orders of magnitude slower than plaintext computation due to CPU and network overheads. That said, various applications like federated data analytics [1, 66], coopetitive machine learning [94], and privacy-preserving recommendation [63] *require* SC. Due to privacy constraints, running these applications in plaintext is not an option. By bringing memory management overhead for SC to nearly zero, MAGE helps make SC more practical and potentially enables more SC-based applications.

2 Secure Computation Background

2.1 Circuit Representation

As explained earlier, SC is inherently oblivious, meaning that any function f computed using SC cannot have data-dependent memory accesses. Thus, it is natural to describe the function f as a circuit C [13, 23, 37, 55]. C is a combinational circuit that accepts the arguments to f as inputs and produces the result of f applied to those arguments as its output. We write $C = (W, G)$, where W is a set of wires and G is a set of gates. Each *wire* represents a datum whose type is the unit of computation in the SC scheme (in most cases, it is the information stored in a single ciphertext). We denote the subset of W storing C's input as I, and the subset of W storing C's output as O. Each *gate* represents a computation supported by the SC scheme. We will typically assume that each gate has exactly one output wire, and that each $w \notin O$ is the input wire of at least one gate. Thus, $|W| = |G| + |I|$.

The particular data types represented in the wires and the types of supported gates depend on the particular SC scheme of interest. For the CKKS homomorphic encryption

scheme [16], each wire represents a *vector of real numbers* and each gate represents an element-wise *addition or multiplication* of those vectors. For garbled circuits [88], each wire represents a *single bit* and each gate represents a binary *AND operation or XOR operation* on those bits. Other SC schemes can be similarly formulated this way. Below, we explain CKKS and garbled circuits in greater depth.

2.2 CKKS Homomorphic Encryption

In the CKKS scheme [16], each ciphertext encodes a vector of real or complex numbers (stored with limited precision). Given ciphertexts $c_1 = \mathsf{Enc}(\vec{v}_1)$ and $c_2 = \mathsf{Enc}(\vec{v}_2)$, one can compute $\mathsf{Enc}(\vec{v}_1 + \vec{v}_2)$ and $\mathsf{Enc}(\vec{v}_1 \circ \vec{v}_2)$ (where $\circ$ is element-wise multiplication). The dimension of each vector depends on parameters chosen during key generation. Each ciphertext is assigned a level, which is a nonnegative integer. When performing the element-wise multiplication operation, both input ciphertexts must have the same level; the level of the output ciphertext is one less than the level of the inputs. Performing element-wise addition does not reduce the ciphertext level the way element-wise multiplication does. A ciphertext at level 0 cannot be used for element-wise multiplication. The maximum level of a ciphertext depends on the parameters chosen during key generation. While one can run a bootstrapping procedure to increase the level of a ciphertext, it is very expensive, and therefore not implemented by all libraries.

2.3 Garbled Circuits

Yao's garbled circuit protocol [88] (referred to simply as *garbled circuits*) allows two parties, called the *garbler* and the *evaluator*, to jointly compute a function f over their private inputs x_1 and x_2. The protocol requires f to be represented as a *boolean circuit* C. Unlike CKKS, there are no restrictions on C's depth. However, *both* parties have to execute the circuit.

First, the two parties run a protocol called *oblivious transfer* to obtain the (encrypted) wire values for their inputs without revealing their inputs. Then the garbler encrypts C in a special way called *garbling* to obtain $\widetilde{C}$, called a *garbled circuit*. The process of garbling is analogous to executing the circuit; a gate cannot be garbled until the (encrypted) values of both input wires are obtained, and garbling a gate produces, as a side effect, the (encrypted) value of the output wire. Then, the garbler sends $\widetilde{C}$ to the evaluator. The evaluator executes the circuit, executing each gate using the gate's garbled information in $\widetilde{C}$. Finally, the two parties communicate to decipher the plaintext values of the output wires.

If the parties would like to repeat the computation again with different inputs, they must re-garble C. It is insecure to reuse the same garbled circuit $\widetilde{C}$ with different sets of inputs.

More comprehensive explanations of garbled circuits, their underlying cryptography, and their state-of-the-art optimizations are available in other resources [6, 69, 86].

2.4 Efficiently Executing Circuits

In this section, we give background on existing techniques for efficiently executing cryptographic circuits. Although many of these techniques were developed for garbled circuits, they mostly apply to homomorphic encryption as well.

2.4.1 Naïve Baseline

Early garbled circuit systems, like Fairplay [55], JKS [41], and PSPW [65], allocate memory for all wires and store the entire garbled circuit in memory. The memory overhead is $\mathcal{O}(|W| + |G|)$. Because, for a well-formed circuit, $|G| + |I| = |W|$, this is equivalent to $\mathcal{O}(|W|)$.

2.4.2 Pipelining Garbling and Evaluation

After the garbler garbles a gate to include in $\widetilde{C}$, the garbler does not use that gate's garbled data. Similarly, once the evaluator evaluates a gate, it never again uses that garbled gate. Based on this observation, the HEKM system [37] operates without keeping the entire garbled circuit in memory, as follows. The garbler and the evaluator first agree on an order in which to execute the gates in C. Then, the garbler garbles each gate and streams the garbled gates to the evaluator, who evaluates the gates in the same order. In this way, all gates are garbled and evaluated, without materializing the full set of garbled gates at any one time. Because space is allocated for all wires in the circuit, the memory overhead is still $\mathcal{O}(|W|)$.

2.4.3 Reclaiming Wire Memory

When executing a circuit, one can discard the memory for a wire once all gates it feeds into have been executed. Only wires whose values have been computed and will be used in the future—the *live* wires—must be kept in memory. The KSS system [49] takes advantage of this by dynamically attaching a reference count to each wire; PCF [48] statically calculates when to reuse wire memory. Using interpretation techniques developed in PCF [48] and refined in Frigate [60], not even the plaintext circuit is materialized in memory. TinyGarble [73], EMP-toolkit [82] (for semi-honest SMPC), and EVA [23] also use variants of this technique. With this optimization, the memory demand is $\mathcal{O}(w)$, where w is the size of the largest set of live wires when executing the circuit. MAGE builds on this line of work by exploring how to efficiently swap to storage when w wires do not fit in memory.

3 Memory Overhead of Secure Computation

First, we discuss the memory overhead of SC. Then, we discuss the memory overhead for collaborative applications.

3.1 Analysis of the Memory Demand

The size of the circuit, for a computation, is proportional to the size of the computation. But in many cases, the memory demand is substantially smaller than the circuit size; only w wires need to be stored, where w is the size of the largest set of live wires when executing the circuit (§2.4.3).

In practice, circuits are often described in a programming language [34, 78] and gates are executed in the same order as the program is interpreted. In this execution order, live wires correspond to in-scope variables in the program. Thus, the memory usage of running an SC program has the same order of growth as running the same algorithm in plaintext.

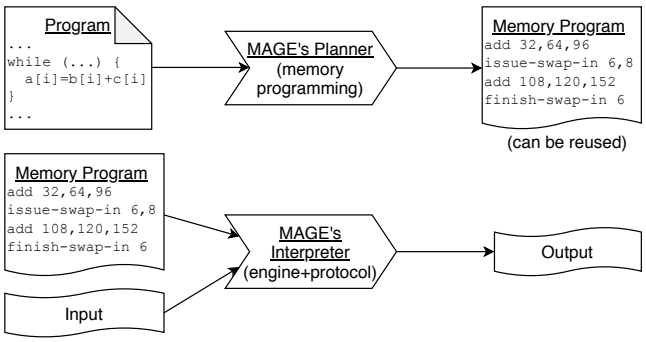


Figure 1: Overview of MAGE. It consists of two phases: planning (top) and execution (bottom)

The memory cost of SC lies in the constant factors. When executing a secure computation protocol, the wire values are encrypted. Thus, a key parameter is the expansion factor of the encryption. In garbled circuits using a 128-bit block cipher, including state-of-the-art optimizations (Point-and-Permute [2], Free XOR [47], Half Gates [90], and Fixed-Key Block Cipher [5, 31]), each wire value is 16 bytes. Each wire represents only 1 bit of plaintext, so this is a 128× expansion factor. For CKKS, ciphertexts at higher levels are larger than ciphertexts at lower levels. For the parameters we used in our evaluation, each ciphertext is hundreds of kilobytes and encodes a vector of dimension up to 4,096.

3.2 Scaling Collaborative Applications

SMPC supports *collaborative applications* over secret data, such as federated data analytics [1] and cooperative machine learning [59]. A common technique to reduce SMPC's overhead is to use SMPC in a *minimal way*. For example, some approaches aim to use SMPC for only a small part of the overall computation [1, 43, 53, 79, 94]. Others carefully choose algorithms that can be executed efficiently in SMPC or use approximations that incur less overhead [58, 59, 68]. But even with these approaches, the SMPC computation often has high memory demands [66]. Thus, it remains important to efficiently execute SMPC computations that do not fit in memory.

4 Overview of MAGE

SC workloads are oblivious by nature. Thus, MAGE can work out the program's memory access pattern in advance, and use this information to produce a memory management plan, called a *memory program*, tailored to the particular access pattern. Importantly, obliviousness is not merely an artifact of certain existing SC schemes; it is inherent to SC. Otherwise, an adversary could potentially infer information about secret data based on the memory access pattern.

To support this paradigm, MAGE's workflow has two phases, as shown in Fig. 1. An SC application is written in a DSL internal to C++. MAGE's planner unrolls the DSL code to produce a bytecode, and then performs transformations on the bytecode to produce a memory program. In MAGE, the memory program is a bytecode that includes *swap directives* describing when to transfer data between storage and memory. Finally, the memory program is given to MAGE's interpreter, which executes it using the SC protocol.

For multi-party protocols, the parties run separate instances of MAGE's interpreter. In the case of garbled circuits, garbled gates are streamed from the garbler to the evaluator, as described in §2.4.2. Both the garbler and evaluator use MAGE to follow a memory program and run with constrained memory.

Our approach of including swap directives in the memory program relies on the planner knowing how much memory will be available at runtime. An alternative approach is for memory programs to be agnostic to the amount of available memory. This would add runtime overhead, as MAGE's interpreter would need to decide which pages to evict. In contrast, our approach moves this overhead to the planning phase, keeping the execution phase as lightweight as possible.

4.1 Address Translation in MAGE

The application programmer should not have to manage paging, so it is natural to write DSL programs in a virtual address space that is, in effect, infinitely large. Central to designing MAGE is deciding at which point in Fig. 1 to translate this address space into a physical address space that fits in RAM.

One possibility (which MAGE does not use) is to perform address translation at runtime, using standard operating system mechanisms for prefetching and address translation. At runtime, swap directives in the memory program would ask the operating system to page parts of the virtual address space out to storage or in to RAM. Unfortunately, the existing way for a Linux process to do this—the `madvise` system call—is too limited. As of Linux 5.10, pages brought into RAM using the `MADV_WILLNEED` hint are not mapped in the page table, so a minor page fault is incurred on the first subsequent access. Similarly, the `MADV_PAGEOUT` hint merely marks pages as inactive; it does not swap out pages immediately.

In contrast, MAGE does not rely on OS address translation for demand paging. MAGE's engine moves data between memory and storage via explicit I/O operations, so that its resident set size never exceeds the available RAM. At the surface, this is similar to buffer management in a DBMS. But unlike a DBMS, MAGE's planner can be viewed as solving an address translation problem in advance. The DSL variables declared by the programmer exist in a *MAGE-virtual address space*, and the final memory program output by the planner references data (i.e., wire values) in a *MAGE-physical address space* that fits within RAM. MAGE's planner creates these address spaces and performs their translation in software during the planning phase. It includes swap directives in the memory program so that the interpreter does not run out of RAM.

To avoid confusion, we will refer to the addresses created by the OS and sent over the memory bus as *OS-virtual addresses* and *OS-physical addresses*. At runtime, MAGE's interpreter stores the program's memory in an array, and each MAGE-physical address in the memory program is treated as an index into this array. Thus, MAGE-physical addresses roughly correspond to the OS-virtual addresses of MAGE's interpreter.

MAGE's approach to address translation has several advantages. First, in contrast to an `madvise`-based approach, MAGE's planner has nearly complete control over when pages are brought into memory and evicted to storage. Second, by translating addresses in the planner, MAGE avoids address-translation-related overheads at runtime. In contrast, relying on OS address translation would mean minor page faults, page table updates, and TLB invalidations at runtime.

MAGE's approach also has a few drawbacks, however. First, the planning phase takes longer because MAGE's planner must translate all addresses in software. Second, memory programs are considerably larger because they must contain not only swap directives, but also a copy of the program translated to operate on MAGE-physical addresses. In particular, the memory program's length is proportional to the program's execution time because a variable local to a function or loop could be assigned different physical addresses each time the function is called or on each iteration of the loop.

Overall, we felt that the advantages of this design outweighed its drawbacks. Longer planning times seemed reasonable because planning can happen offline and the resulting memory program can be used repeatedly. The larger memory program size was an acceptable tradeoff because MAGE's planner materializes an unrolled form of the program anyway to run Belady's algorithm. Meanwhile, MAGE's planner is afforded nearly full control of page eviction and replacement and MAGE's runtime overheads remain relatively small.

4.2 MAGE's Bytecode Representation

Recall that MAGE's planner expresses the program as an unrolled (branch-free) bytecode, and performs transformations on it to compute the memory program bytecode. What operations should the bytecode instructions support?

One possibility would be for the bytecode to describe low-level operations similar to those supported by a CPU, excluding control flow instructions. Unfortunately, such a bytecode includes the raw memory trace of the program, which, as discussed in §1, can be impractically large.

One alternative, used by PCF [48] and Frigate [60][2] (but not MAGE), is to have each instruction correspond to a gate in the circuit C being executed. This approach would require a *protocol driver* in MAGE's interpreter that executes each gate using the SC protocol. To understand why this is inefficient, consider garbled circuits, for which gates are binary and wires represent bits. The programmer specifies the circuit in terms of operations on high-level types such as integers, which are then compiled into bit-level operations. Thus, each time the program performs a high-level operation (e.g., adding two integers), the same subcircuit (e.g., describing integer addition in terms of binary gates) is repeated in the bytecode.

To eliminate this repetition, MAGE has each instruction describe a high-level operation directly. This requires not only a *protocol driver*, but also an *engine* in MAGE's interpreter that expands each instruction into the relevant subcircuit at runtime. MAGE's planner does not need to materialize the subcircuits because wires internal to the subcircuits are very short-lived and therefore can be ignored.

[2]Unlike MAGE, these systems also include control flow operations.

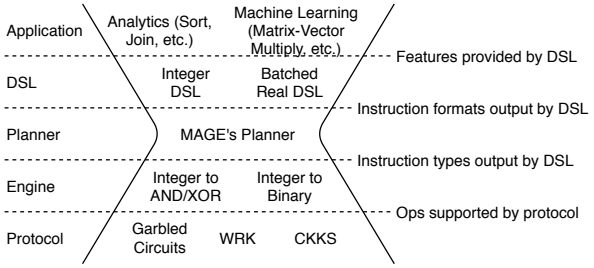


Figure 2: MAGE's envisioned ecosystem, with planning as the narrow waist

4.3 MAGE's Ecosystem and its Extensibility

An important consideration in MAGE's design is to be applicable to a range of SC protocols. For example, garbled circuits and homomorphic encryption (CKKS) have quite different computation models, yet we show how MAGE captures both. MAGE's envisioned ecosystem can be understood as a set of layers with a narrow waist, as shown in Fig. 2. The narrow waist is MAGE's planner; MAGE's core planning algorithms can be used with a variety of applications and interpreters.

MAGE's interpreter has two layers. The upper layer, called the *engine*, decomposes each instruction into a subcircuit of gates supported by the target SC protocol (§4.2). The lower layer, called the *protocol driver*, evaluates gates with the SC protocol. For example, when using a protocol that supports only binary AND and XOR operations (e.g., garbled circuits), one must use an engine that decomposes each instruction into a circuit of AND and XOR gates. In contrast, when using a protocol that supports all types of binary gates (e.g., TFHE [17]), one can use an engine that uses all types of gates.

One must choose compatible implementations at each layer. For example, once one has selected an SC protocol, one should choose an engine that executes each instruction using operations supported by that protocol. Then, one should select a DSL that outputs instructions that the chosen engine understands. Finally, one must write the application in that DSL.

MAGE's planner, however, is universally compatible, allowing it to be the "narrow waist" of the ecosystem. The first reason is that MAGE's planner does not have to understand what each instruction *does*, only what memory it accesses. Thus, even if a new instruction is introduced into a DSL, extending a header file to specify its format (which includes which fields are memory addresses) is enough for the planner to understand that instruction. The second reason is that MAGE's planner does not introduce any new instructions except for swap directives, which all engines understand. Thus, if an engine understands the instruction types output by MAGE's DSL, then the engine will also be able to interpret the planner's output (i.e., the memory program).

A number of frameworks and DSLs for SC [34, 78] aim to make it easier for non-SC-experts to use SC. In contrast, MAGE is an efficient SC execution engine; its DSLs are not necessarily geared toward non-experts, do not optimize the resulting circuit, and might expose low-level SC operations. We discuss how these frameworks fit into Fig. 2 in §9.

5 MAGE's Engine

MAGE's execution engine is an interpreter for the final memory program. First, it allocates an array to store the program's data. Each MAGE-physical address is an index into this array. To execute an instruction, MAGE reads the instruction's arguments from this array, makes calls to the protocol layer to compute the output, and writes the output back to the array. Each instruction in the memory program references its input and output data directly by MAGE-physical address; the engine sees no MAGE-virtual addresses. Some instructions, such as those requesting pages to be transferred between storage and memory, are handled directly by the engine, without calling the protocol. We call such instructions *directives*.

5.1 Parallel/Distributed Engine

SC is resource-intensive, so it is natural to scale SC by executing the protocol in a distributed fashion across *multiple CPU cores* or *multiple machines*. The multiple-machine case is useful to overcome resource constraints associated with a single machine such as limited CPU cores, limited storage I/O, or, in the case of SMPC, limited network bandwidth. This is different from having multiple parties in SMPC. Here, we are parallelizing a single trust domain—for example, a single logical party in SMPC may execute using multiple machines.

MAGE's engine supports distributed execution across multiple *workers*. Each worker is a thread of computation, running MAGE's engine, operating on its own memory region (a MAGE-physical address space). Workers differ from OS processes as follows: (1) each worker contains exactly one thread, (2) workers are not necessarily isolated by hardware such as an MMU—multiple workers in a MAGE computation could, in principle, run within the same process, and (3) memory is statically partitioned among the workers.

MAGE's planner does not automatically infer how to parallelize the computation. Rather, the programmer writes DSL code in a distributed memory model, explicitly indicating asynchronous network operations to transfer data among the different workers. The resulting memory program bytecode contains *network directives* that the engine interprets. Similarly, the protocol driver must be written to function properly when the computation is distributed over multiple workers.

Programs for MAGE are parameterized by the Worker ID. MAGE's planner is run once *for each worker*. To generate the memory program for a worker, the planner processes only the accesses for that worker—it does not need to consider other workers' accesses, because each worker can only access its own memory region. Thus, the workers' memory programs can be generated independently and in parallel.

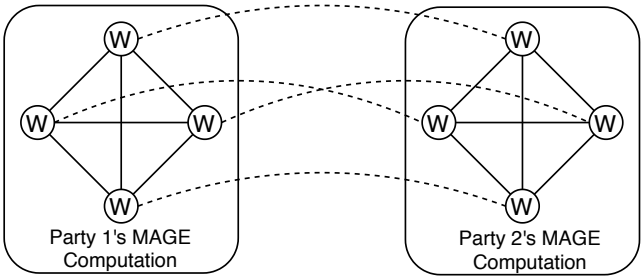


Figure 3: Example of distributed SMPC with MAGE. Workers are denoted as circles with W. Solid lines indicate connections managed by MAGE's engine; dashed lines indicate connections managed by the protocol driver

Using a distributed memory model provides two benefits. First, it allows MAGE to be agnostic to whether workers are placed on a single machine or across multiple machines. Second, it guarantees that the access pattern for each region of memory consists of a single well-defined sequence, simplifying planning. To ease the difficulty of explicitly specifying network transfers, one can build easier-to-use DSL libraries for common communication patterns (e.g., our implementation provides a `ShardedArray<T>` abstraction).

5.2 Distributed SMPC

Some SC protocols, like SMPC, require interaction over the network between mutually distrusting parties. For such protocols, each party runs a separate MAGE computation, with its own set of workers. Whereas the MAGE engine handles *intra-party communication* between workers in the same party, the protocol implementation handles *inter-party communication* among workers in different parties. The inter-party topology is up to the protocol driver; our protocol driver for garbled circuits uses a one-to-one inter-party topology (Fig. 3).

6 MAGE's Planner

Our memory programming approach is to calculate the memory access pattern in advance and use it to preplan memory management. One can potentially preplan the following:

- **Placement.** How should we divide up a circuit into pages?
- **Ordering.** In what order should we evaluate the gates in the SC circuit to result in the best memory behavior?
- **Scheduling.** When should pages that will be used in the future be swapped in from storage?
- **Replacement.** How should we choose pages to evict when making room for pages from storage?

MAGE produces an approximate solution, using a heuristic for placement and optimizing scheduling and replacement. Note that MAGE does not optimize ordering; it evaluates gates in the order implicit in the DSL program for the circuit.[3]

6.1 Organization of MAGE's Planner

We organize MAGE's planner into stages (Fig. 4):

[3] Optimizing ordering may be NP-hard [76]. A system that does so would be very powerful—for example, it would automatically block a loop join or tile a matrix multiplication. It is beyond the scope of this work.

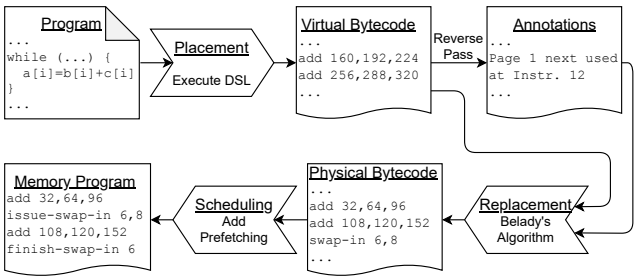


Figure 4: MAGE's planner's workflow, with its three stages

1. **Placement.** This stage accepts a DSL program and organizes wires into MAGE-virtual pages. It outputs instructions referencing wires by MAGE-virtual address.
2. **Replacement.** This stage adds instructions to swap pages to/from storage, deciding which pages to evict. It outputs instructions referencing wires by MAGE-physical address.
3. **Scheduling.** This stage moves swap instructions within the instruction stream and relocates wires to mask the latency of moving data between memory and storage.

For a parallel/distributed program, MAGE's planner is invoked separately for each worker, with separate MAGE-virtual and MAGE-physical address spaces. Network directives in the program transfer data among those address spaces.

MAGE's planner does not benefit from MAGE's memory programming techniques, so it is important that planning does not consume an unreasonable amount of memory. We keep the planner's memory usage lightweight by (1) writing/reading the intermediate bytecodes to/from files instead of keeping it all in memory, (2) designing the DSLs to be lightweight, and (3) keeping track of pages instead of individual bytes.

6.2 MAGE's First Stage: Placement

MAGE's placement module is, in effect, a page-aware memory allocator for the DSL. It unrolls the DSL, allocating space for each variable and intermediate value in the MAGE-virtual address space. It outputs a bytecode for the program in which each variable is referenced by its MAGE-virtual address.

6.2.1 Unrolling the DSL Code

MAGE's DSLs are internal to C++. This means that the DSL is a set of convenient C++ APIs to specify the program's behavior, often involving operator overloading. The program is specified as a C++ function that uses these APIs.

Fig. 5 shows a program that solves Yao's Millionaire's problem [87]. `Integer<width>` describes an Integer datum with the specified width in bits. `Bit` is an alias for `Integer<1>`.

MAGE's planner does not parse the DSL program's source code or manipulate its AST. Instead, it simply calls the C++ function containing the DSL program. As the DSL code executes, it produces a bytecode describing the computation. For example, the overloaded `+` operator for `Integer` emits an `Add` instruction in the output bytecode; it does not actually add integers using secure computation. Each output instruction references its operands by MAGE-virtual address. Thus, the DSL (e.g., the `Integer` class) calls MAGE's placement module to allocate memory in the MAGE-virtual address space for intermediate results, including those stored in variables.

```
void millionaire(const ProgramOptions& args) {
    Integer<32> alice_wealth, bob_wealth;
    alice_wealth.mark_input(Party::Garbler);
    bob_wealth.mark_input(Party::Evaluator);
    Bit result = alice_wealth >= bob_wealth;
    result.mark_output();
}
```

Figure 5: Example code in an Integer-based DSL internal to C++ to solve Yao's Millionaire's problem

For example, see Fig. 5. On the `mark_input` and `>=` operations, an allocation request is made to MAGE's placement module to obtain a MAGE-virtual address, and an instruction is emitted to perform that operation (obtain input or integer comparison) and store the result at that MAGE-virtual address. Once an `Integer`'s destructor is called, or if an `Integer` is reassigned to a new MAGE-virtual address, a deallocation request is made to MAGE's placement module for the MAGE-virtual address previously held by that `Integer`.

For a parallel/distributed program, the worker ID and total number of workers are provided via the `ProgramOptions` structure. The C++ code can branch on these variables, to have each worker operate differently and exchange data appropriately to perform the parallel/distributed computation.

Each `Integer` object contains only the MAGE-virtual address of its contents; other attributes, such as width, are template arguments and do not consume memory. Thus, `Integer`s and other DSL-provided data types are typically smaller than the encrypted data items they represent. For example, a 32-bit integer encrypted for the garbled circuit protocol is 1 KiB in size, whereas an `Integer<32>` object used during planning is just 8 B (a single MAGE-virtual pointer). This helps keep the memory cost of the planning phase small.

6.2.2 Memory Allocation Strategy

When MAGE's placement module allocates memory for a variable, it ensures that the variable is contained in a single MAGE-virtual page; a variable must never straddle two pages. The reason is that two adjacent MAGE-virtual pages may not be adjacent in the OS-virtual address space at runtime.

A key issue in designing the placement module's memory allocator is internal fragmentation [25, 67]. Some fragmentation, which we call *classic fragmentation*, arises from the inability to pack variables onto pages (e.g., part of a page's space cannot store any variable). Another type of fragmentation, which we call *effective fragmentation*, arises from the page's lifetime exceeding some of the variables it stores; if even one wire on a page is alive, the entire page remains alive.

To reduce classic fragmentation, MAGE's placement stage uses techniques from slab allocators [8]. Each page contains only variables of a particular size. When a variable goes out of scope in the DSL, its "slot" in its page is marked as free.

When a space for a variable must be allocated, MAGE's placement module look for a free slot in a page containing variables of that size; if no such pages have free slots, it allocates a new page for variables of that size. The slab size is one MAGE-virtual page. This ensures that no variable will straddle a page boundary. Just as in slab allocators, some leftover space at the end of a page may be unusable, but this can be controlled by tuning the page size. Unlike slab allocators, MAGE's placement module does not preserve object state across allocations.

To reduce effective fragmentation, MAGE's placement stage uses the following heuristic when allocating memory for a variable. If multiple pages, for the specified variable size, have free slots available, then MAGE uses the candidate page with the fewest free slots. This allows the number of live pages to decrease if the number of live variables decreases, by giving a chance for all variables on a page to die.

6.3 MAGE's Second Stage: Replacement

We apply Belady's MIN algorithm [3]. MIN is theoretically optimal in the number of SWAP-IN operations, but it does not minimize the number of swap operations if SWAP-OUT operations are also considered. The reason is that only dirty pages need to be written back to storage (i.e., "swapped out"). Minimizing the number of swaps when taking this into account is NP-hard [28]. Regardless, MIN produces a solution with at most $2\times$ as many swaps as the theoretical optimum,[4] so it is useful in MAGE's replacement stage.

To use MIN, we first make a backward pass over the program to determine, each time a page is used, the time (instruction ID) at which it is used next. Then we make a forward pass over the program, using the annotated next use time to determine which page to swap out. This requires us to maintain a priority queue of resident pages, so that we can quickly identify which one's next use is farthest in the future. Each instruction, even if its arguments are already resident, requires us to also perform a `decrease_key` operation on the priority queue to adjust pages' next use time. Therefore, if N is the number of instructions and T is the number of pages that fit in memory, applying Belady's MIN algorithm is $\mathcal{O}(N \log T)$.

This stage outputs an instruction stream that contains swap directives and references wires by MAGE-physical address. To support this, MAGE's planner maintains a data structure that maps MAGE-virtual page numbers to MAGE-physical frame numbers, similar to a page table.

When planning a parallel/distributed program, the planner must be careful to not steal a page that is currently being used for network I/O. Thus, MAGE's replacement phase reads the network directives to infer the outstanding asynchronous network operations. When stealing pages, it issues *network barrier* directives, as necessary, to ensure that the engine waits for the relevant network I/Os to complete.

[4] This occurs in the worst case where it evicts only dirty pages, but there is an optimal solution that evicts the same number of clean pages.

6.4 MAGE's Third Stage: Scheduling

We introduce a parameter ℓ called the *lookahead*. To prefetch data, MAGE's scheduling algorithm attempts to move SWAP-IN directives ℓ instructions earlier in the instruction stream. However, this does not work if one of the ℓ intervening instructions uses the page frame into which we are bringing in data. We solve this by budgeting B extra physical page frames, called the *prefetch buffer*; the replacement stage is now run with a capacity of $T - B$ frames, not T frames. Data is brought asynchronously into a free slot in the prefetch buffer. Only when it is finally needed is it copied from the prefetch buffer into its destination physical page frame. Instead of SWAP-IN directives, the memory program contains ISSUE-SWAP-IN directives, which initiate the transfer of a page into memory, and FINISH-SWAP-IN directives, which block execution until a swap operation has completed. Ideally, swap operations will be scheduled such that FINISH-SWAP-IN never blocks, but it serves as an important fallback to prevent old/corrupt data from being used if the transfer is unpredictably delayed.

We use the prefetch buffer similarly to swap out pages. The page to be swapped out is copied into a free slot in the prefetch buffer and then swapped out to storage with an ISSUE-SWAP-OUT directive while execution of subsequent instructions continues. Unlike SWAP-IN operations, there is no clear deadline by which the write to storage must complete. Thus, we delay issuing a FINISH-SWAP-OUT directive for as long as possible; we only issue it when allocating a slot in the prefetch buffer fails. In such a situation, we identify the oldest ISSUE-SWAP-OUT operation, issue the FINISH-SWAP-OUT directive for it, and reclaim its page in the prefetch buffer.

One could eliminate the copying of pages to/from the prefetch buffer by rewriting future instructions. We did not implement this optimization because it would introduce additional complexity and MAGE performs well without it.

A natural question is how large B must be. SSDs have bandwidths less than 10 GB/s and latencies that are usually less than 1 ms. Based on these measurements, Little's Law gives: $B = 10\text{ GB/s} \cdot 1\text{ ms} = 10\text{ MB}$. For server-class machines, this is $< 1\%$ of physical memory. In practice, we use 16–32 MiB to account for burstiness/queuing, still only a small fraction of available memory. Thus, MAGE's scheduling promises to mask storage latency with only a small memory penalty.

7 Implementation

We implemented a prototype of MAGE in C++, including support for two protocols: garbled circuits and CKKS. Using `cloc`, we found that our implementation is $\approx 11{,}000$ lines of code, excluding comments and blank lines, broken down as follows: $\approx 2{,}800$ for common libraries used throughout MAGE (e.g., data buffering for I/O, configuration file parsing, etc.); $\approx 1{,}300$ for MAGE's planner; ≈ 900 for protocol drivers (not including the underlying cryptography); $\approx 1{,}000$ for MAGE's DSLs and libraries for those DSLs (e.g., for sharding data); $\approx 1{,}100$ for MAGE's engines; $\approx 1{,}600$ for SC

programs written in MAGE's DSLs, used for testing and evaluating MAGE; $\approx 1,900$ for the underlying cryptography for garbled circuits, much of which is based on EMP-toolkit [82]; and ≈ 400 for in-progress (not yet complete) support for a third protocol. We build MAGE using `clang++` version 10.0.0 with the optimization flags `-Ofast -march=native`. MAGE runs as a Linux process, with no changes to kernel code.

7.1 MAGE's Interpreter

Engine. The `Engine` class implements common functionality for the engine layer, including support for directives. It establishes pairwise TCP connections among workers within a single party, to support network directives. Swap directives are implemented using the `aio` facility provided by the kernel (not to be confused with POSIX `aio`); the swap file/device is `open`ed with the `O_DIRECT` flag. MAGE engines are implemented as class templates that extend (inherit from) the `Engine` class. The protocol driver class is provided to the engine as a template argument, so the engine can make calls to it. We avoided using virtual functions for this, as their overhead can be significant (e.g., for free XORs).

Protocol Driver. The protocol driver exposes the SC protocol's native operations to the engine as a set of methods. When the engine invokes these methods, it provides pointers to data to operate on, stored in a large array representing the MAGE-physical address space. The protocol driver specifies the type of entries in the engine's array, in effect dictating what each MAGE-physical address actually corresponds to for its protocol (plaintext bits, ciphertext bytes, etc.), and provides a plugin to the DSL so it can allocate MAGE-virtual memory accordingly. The protocol driver must not store pointers to dynamically allocated memory in the array. The reason is that the engine swaps out only the contents of the array, not including any dynamically-allocated memory it points to. In addition to the SC protocol's cryptographic routines, the driver manages all protocol-specific operations. This includes sharing protocol-specific state among workers within a party, obtaining input data, writing output data, and managing intra-party communication where necessary (e.g., sending garbled gates from the garbler to the evaluator).

7.2 Extending MAGE with New Protocols

To extend MAGE with a new protocol, one must, at minimum, write a protocol driver to support it. If the operations exposed by the new protocol driver are identical to those exposed by an existing protocol driver, then one can use the same engine that works with the existing protocol. Otherwise, one must implement a new engine or modify an existing engine. This involves deciding which instruction types the new engine will be compatible with. If the supported instruction types differ from what existing DSLs produce, then one may have to implement a new DSL or modify an existing DSL.

We implemented protocol drivers for garbled circuits and CKKS. Garbled circuits and CKKS support different operations, so we implemented a separate DSL (Integers vs. Batches) and engine (AND-XOR vs. Add-Multiply) for each protocol. This conveniently allows us to showcase MAGE's ability to support different implementations of each layer. That said, it is not uncommon for related SC protocols to expose similar interfaces. For example, the WRK protocol [83, 84] exposes the same interface as garbled circuits (AND-XOR), so support for WRK, if added, could reuse our Integer DSL and AND-XOR engine.

7.3 Garbled Circuit Protocol Driver

For garbled circuits, wires have uniform size, so we allow MAGE address spaces to be wire-addressed; the DSL is unaware of the size of wires in bytes. Some subcircuits used by the AND-XOR engine are based on those used by Obliv-C [89]. Our garbled circuit driver uses cryptographic kernels from EMP-toolkit [82]. We implement oblivious transfer (OT) using multiple background threads. Concurrently with our work, EMP-toolkit was updated to use the MiTCCRH hash function [31]; our implementation is based on an older version of EMP-toolkit based on fixed-key AES [5]. When we compare MAGE to EMP-toolkit in §8, we use the older version of EMP-toolkit so the comparison is fair. This is not a limitation of MAGE; our driver could be changed to use MiTCCRH.

7.4 CKKS Protocol Driver

CKKS ciphertexts vary in size depending on their level, so for CKKS' DSL and engine, MAGE address spaces are byte-addressed. The protocol driver provides a plugin to the DSL describing the particular wire sizes in bytes. It uses the CKKS implementation in Microsoft SEAL [71]. We chose parameters for CKKS that allow a multiplicative depth of 2. A challenge was that SEAL ciphertext objects contain pointers and dynamically-allocated memory. MAGE cannot swap such objects to storage (see §7.1). Thus, TE protocol driver serializes ciphertexts using SEAL's built-in serialization methods when they are not in use; each operation (e.g., add, multiply) deserializes the arguments, computes the result, and then serializes the result. We quantify the cost of serialization in §8. This overhead is not fundamental; CKKS ciphertexts could be implemented as flat buffers, or homomorphic operations could be implemented to operate directly on serialized ciphertexts.

After a multiplication, CKKS ciphertexts are typically relinearized and rescaled before the next multiplication. But if two products are added (e.g., $ab + cd$), one can perform relinearization once for the overall result instead of for each multiplication separately (e.g., ab and cd). MAGE's DSL supports this optimization, which is crucial to achieve good performance on **rstats** and the linear algebra workloads.

8 Evaluation

8.1 Workloads

We now establish a set of SC workloads for our evaluation. Garbled circuits and CKKS support different operations—bitwise operations for garbled circuits, and add-multiply circuits of low multiplicative depth for CKKS—so we design

separate workloads for each protocol. These workloads are data-intensive "kernels" that may be used as part of larger SC applications. We discuss larger SC applications in §8.8.

8.1.1 SMPC Collaborative Applications

One application of SMPC is federated data analytics [66, 79]. Aggregations (`GROUP BY` operations) and joins are particularly memory-intensive. A federated data analytics system may express equi-joins as set intersections (SI) and aggregations as set unions (SU), both of which can be implemented by merging sorted lists [66]. This inspires our first benchmark, **merge**: *merging sorted lists of records*. In some cases, the input lists may not be already sorted. This inspires our second benchmark, **sort**: *sorting a list of records*. For joins other than equi-joins, the system must fall back to a classic loop join. This is our third benchmark, **ljoin**: *loop join*. For concreteness, we assume that each record is 128 bits long, and that the first 32 bits are the key used for sorting or joining; the problem size n is the number of records per party.

Privacy-preserving machine learning applications inspire our fourth benchmark, **mvmul**: *matrix-vector multiply with 8-bit integers*. A recent proposal for secure neural network inference, XONN [68], suggests *binarizing* the neural network. This inspires our fifth benchmark, **binfclayer**: *binary fully-connected layer*. It consists of a series of XNOR and PopCount operations similar to multiplying a binary matrix by a binary vector, followed by a binary activation function. For simplicity, we do not include batch normalization.

8.1.2 CKKS Homomorphic Encryption

We restrict ourselves to workloads for which CKKS is efficient—workloads that can be expressed as arithmetic circuits of low multiplicative depth. The sixth workload is **rsum**: *sum of a list of real numbers*, which requires no multiplications. The seventh workload is **rstats**: *computing the mean and variance of real numbers*, which requires a multiplicative depth of 2. These represent simple data analytics workloads; the problem size n is the number of elements.

Our remaining workloads are inspired by machine learning and linear algebra. The eighth workload is **rmvmul**: *matrix-vector multiply with real numbers*. Finally, we consider two variants of matrix multiplication. The ninth workload is **n_rmatmul**: *matrix-matrix multiply with a naïve nested `for` loop*. The tenth workload is **t_rmatmul**: *tiled matrix-matrix multiply*. The problem size n is the length of one side of the matrix (also for **mvmul** and **binfclayer**).

8.1.3 Implementation of Workloads

For simplicity, our implementations of some of these workloads only support power-of-two sizes and power-of-two number of workers, but this is not a fundamental limitation of MAGE. Some workloads can, in principle, be optimized through streaming. For example, **rsum** could read each input one at a time, add the result to an accumulator, and then output the accumulator, instead of holding the entire input dataset in memory. We deliberately avoided such "optimizations," as they would not be possible if the workload were part of a larger computation whose intermediate results are held in memory. Thus, each workload operates in three non-overlapping phases: (1) the inputs are read into memory, (2) the computation is performed, materializing the output in memory, and (3) the output is written to a file.

For the parameters we chose, the CKKS scheme encrypts vectors of dimension 4096. Thus, each of our workloads for CKKS could be applied to 4096 instances of the problem in a SIMD fashion with no additional overhead. There are ways to use the 4096 slots in the vector to speed up a *single* problem, for example, by vectorizing matrix multiplication [42]. Our workloads, for simplicity, do not apply such techniques, but MAGE is not incompatible with them.

8.2 Empirical Methodology

We compare MAGE's performance to an upper bound and a lower bound. The upper bound, *OS Swapping*, is the speed when relying on the operating system's paging. The lower bound, *Unbounded*, is the speed when the entire computation fits in memory. We measure these three scenarios as follows:
1. Unbounded. MAGE's planner is run assuming enough memory to fit the program. Thus, MAGE's planner does not insert swap directives in the memory program. Finally, MAGE's engine executes the memory program outside of any `cgroup`.
2. OS Swapping. A memory program is generated in the same way as for the *Unbounded* solution. However, it is executed in a `cgroup` that limits physical memory to a fixed amount.
3. MAGE. MAGE's planner is run assuming a fixed physical memory capacity, minus the prefetch buffer and the interpreter's overhead. The resulting plan is run within a `cgroup` that limits physical memory to 1 GiB or 16 GiB, to ensure that the memory overhead fits in the limit.

Except where stated otherwise, we used `D16d_v4` instances on Microsoft Azure [57]. We chose this instance type for a few reasons. First, it has enough memory to fit the entire computation for most experiments, necessary for the Unbounded scenario. Second, it contains a local "temporary" SSD. We use it for swap space (one of its recommended uses [20]) and for the file containing the memory program. Third, it provides enough network bandwidth so as not to be a bottleneck for garbled circuits (we explore the WAN setting in §8.7).

We set MAGE's parameters as follows. For garbled circuits, we used a page size of 64 KiB, lookahead ℓ of 10,000 instructions, and prefetch buffer size B of 256 pages. For CKKS, we used a page size of 2 MiB, lookahead ℓ of 100 instructions, and a prefetch buffer size B of 16 pages. Because CKKS ciphertexts are large, we used a larger page size (slab size) than for garbled circuits to reduce external fragmentation. Additionally, we left an additional 32–64 MiB of memory unused, to accommodate the memory used by MAGE's interpreter.

8.3 Comparison to Existing Frameworks

We compare MAGE's garbled circuits performance to that of EMP-toolkit. Our goal is to demonstrate that MAGE's techniques do not limit the performance of garbled circuits

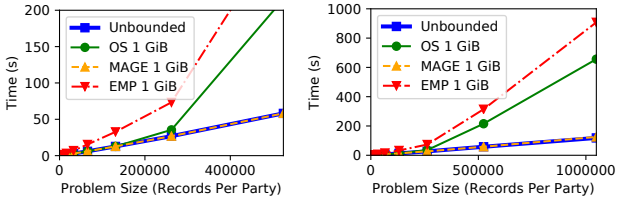


Figure 6: Comparison of MAGE and EMP-toolkit

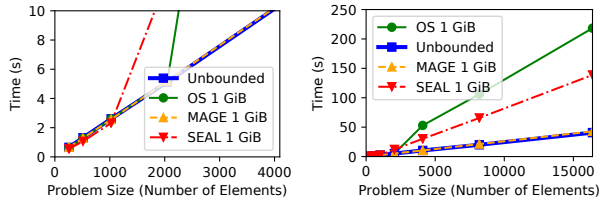


Figure 7: Comparison of MAGE and SEAL

compared to an existing system. We use **merge** for the comparison. We implemented **merge** in EMP-toolkit's DSL, and used EMP-toolkit's library for merging sorted arrays.

We discovered that EMP-toolkit is an order of magnitude slower than MAGE. This was because EMP-toolkit performs a separate invocation of OT extension, which involves a network round-trip, each time an Integer input is read for the evaluator. Our garbled circuits implementation for MAGE does not have this problem because it performs OTs in larger batches using background threads, regardless of the units by which the program reads the input. To eliminate this effect, we exclude the time to read the input, for both EMP-toolkit and MAGE, for this experiment only; we measured the time to merge the two arrays once they are materialized in memory.

We also compare MAGE's CKKS performance on **rstats** to a C++ program that uses SEAL directly. The main source of overhead in MAGE is the need to deserialize the input ciphertexts and serialize the output ciphertext, for each instruction.

The results are shown in Fig. 6 and Fig. 7. The graphs on the left are zoomed in to smaller problem sizes to show the point where memory demand exceeds available physical memory. "OS" refers to scenario 2 in §8.2; "EMP" and "SEAL" refer to those systems similarly running in a `cgroup`. EMP performs about 3× worse than OS when the problem fits in memory; when it does not, the relative overhead is small (≈ 33%). We found that EMP performs worse than OS primarily due to (1) the overhead of its "real-time circuit optimization" feature, (2) inefficient data buffering when using the network, and (3) virtual function overhead when executing the circuit. OS uses MAGE's runtime, so it does not have these issues. SEAL is faster than OS when the problem fits in memory, but only slightly (less than 20%), indicating that the serialization overhead is not large. When the problem size does not fit in memory, SEAL improves further compared to OS, but remains less than 2× faster than OS.

8.4 Overhead of Swapping Pages

We ran the three scenarios on all 10 workloads, using a 1 GiB memory limit. The results are shown in Fig. 8. We ran 8 trials on different Azure instances (8 different pairs of instances, for garbled circuits) and plot the median; error bars are the quartiles. We additionally ran experiments using a 16 GiB memory limit. We increased the problem sizes so that their memory use exceeded 16 GiB (necessary for the OS scenario) but fit within the 64 GiB available on the virtual machines (necessary for the Unbounded scenario). Our methodology is the same as for the 1 GiB memory limit. We do not include **sort** in our results for the 16 GiB memory limit, because the intermediate bytecodes produced while planning were too large for the local SSD. The results are shown in Fig. 9. MAGE outperforms OS swapping by at least 4× on 7 of the workloads, with improvements of ≈ 12× for **ljoin** and ≈ 10× for **rsum**. Its performance is within 15% of Unbounded for 7 of the workloads (including **sort** from Fig. 8).

MAGE's improvement compared to OS is higher for **binfclayer** and **rmvmul** than for **mvmul**; although all three have similar access patterns, **mvmul** has lower memory intensity because multiplying integers in a garbled circuit has high overhead. For complex access patterns, like **merge** and **sort**, MAGE's improvement is not markedly higher than for simple scans like **ljoin**, **rsum**, and **rstats** (note that both input tables for **ljoin** fit in memory; it is the *output*, populated in order, that does not fit). MAGE is less affected by high memory intensity than OS, allowing it to perform well.

8.5 Overhead of Planning

The time and peak memory use for planning each workload for the MAGE scenario in Fig. 8 and Fig. 9 is shown in Table 1. Note that MAGE's planning is outside of the critical path: for a given circuit, MAGE's planner can be run before the parties' inputs are known. For garbled circuits, although the garbled circuit $\widetilde{C}$ cannot be reused if the computation is re-run, MAGE's memory program *can* be safely reused.

The planning time and final memory program size are linear in the size of the *computation* (size of C), not in the size of the memory demand. Nevertheless, the planning times are generally less than the time to perform the execution and the planner's memory consumption is significantly smaller than the available memory at runtime for all experiments.

Generating memory programs for CKKS is more efficient than for garbled circuits. This is because each instruction for CKKS operates on more memory than for garbled circuits, which means that the problem sizes that fill a given physical memory size tend to require smaller bytecodes for CKKS than for garbled circuits. For example, an instruction operating on integers in a garbled circuit program may operate on a few kilobytes of memory (each bit of each integer is 16 bytes), but for CKKS, each instruction operates on a *vector* of real numbers, whose encrypted size is hundreds of kilobytes.

For CKKS, the final memory programs were < 100 MiB for Fig. 8 and < 1 GiB for Fig. 9. For garbled circuits other than **sort**, they were < 5 GiB for Fig. 8 and < 65 GiB for Fig. 9. For **sort**, it was less than < 25 GiB for Fig. 8. MAGE's planner requires about 4–5× times more storage space than

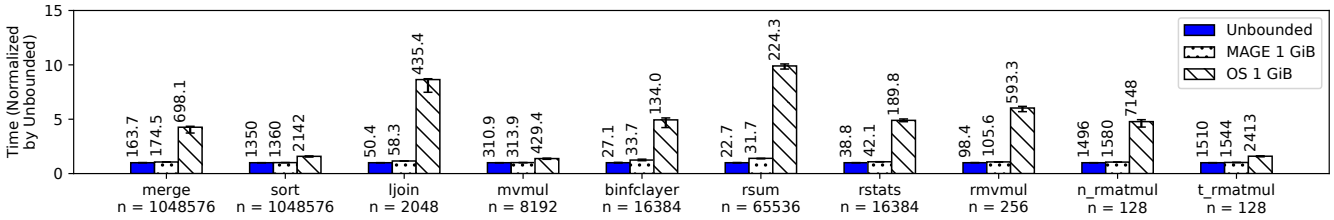


Figure 8: Performance of Unbounded, OS Swapping, and MAGE, normalized by the time for Unbounded; absolute times, in seconds, are printed at the upper left corner of each bar

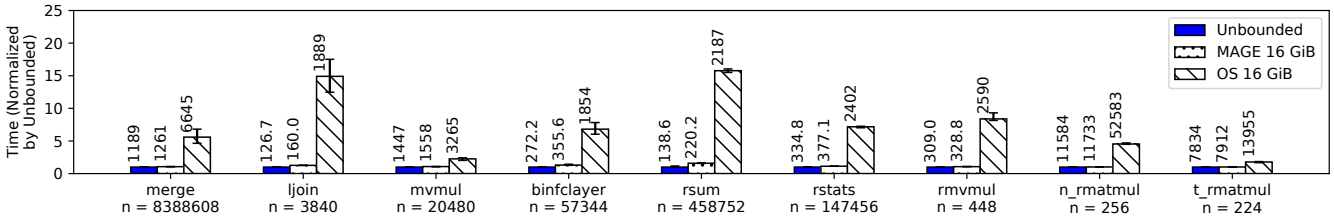


Figure 9: Repeat of Fig. 8, with larger problem sizes and a 16 GiB memory limit (note the larger y-axis scale)

Problem	Time (8)	Mem. (8)	Time (9)	Mem. (9)
merge	38.0	42.6	291.6	299.4
sort	367.3	42.7	N/A	N/A
ljoin	6.7	121.0	23.6	411.4
mvmul	56.0	527.5	298.2	3268
binfclayer	77.2	19.1	1041	165.7
rsum	0.04	9.6	0.29	30.2
rstats	0.04	10.9	0.34	48.5
rmvmul	0.09	16.4	0.24	36.9
n_rmatmul	2.2	246.1	18.6	1927
t_rmatmul	2.3	246.5	12.9	1246

Table 1: Planning times (s) and peak memory use of the planner (MiB) for workloads in Fig. 8 and Fig. 9

the final memory program due to the need to materialize intermediate bytecodes of similar size, but this could be optimized by pipelining stages of MAGE's planner where it is possible to do so (e.g., replacement and scheduling in Fig. 4).

8.6 Impact of Parallelism

We now explore how the relative performance of Unbounded, OS, and MAGE are affected by parallelizing the computation. We did experiments parallelizing the computation across four workers (per party, for garbled circuits). We place each worker on a separate VM instance, each with a separate SSD.

We ran each experiment three times, using the same cluster of machines for all trials, and report the median in Fig. 10. Most experiments follow a similar pattern as Fig. 8, indicating that MAGE's performance gains persist when we parallelize the computation. For two experiments, **merge** and **sort**, MAGE's improvement over OS Swapping visibly increases. Whereas the other workloads are parallelized by splitting the input among the workers in a communication phase at the beginning and then computing independently thereafter, **merge** and **sort** have a communication phase in the *middle* of the computation (several such phases in the case of **sort**). That OS Swapping performs worse for these workloads, but MAGE does not, suggests that the OS virtual memory system might be introducing jitter, which interacts poorly with the communication phase and induces stragglers.

8.7 SMPC in Wide-Area Networks

SC does not always require significant data transfer over the wide area. In HE, computation is done by a single logical party. Even in SMPC, there may be ways for multiple parties to co-locate for an SMPC computation while remaining physically and logically distinct. But in some cases, it is desirable to run SMPC over a wide-area network. We explore this below.

We measure performance of garbled circuits with the two parties hosted on different cloud providers. The garbler was always on Azure in the US West 2 region (Oregon). The evaluator was on Google Cloud (`n2-highcpu-2` [30]). We compare two setups: one where the evaluator was in `us-west1` (Oregon) and one where it was in `us-central1` (Iowa).

Initially, higher latencies and limited single-flow bandwidth limited performance. For example, the round-trip time in the Oregon setup was ≈11 ms, which made OTs a bottleneck.

First, we tuned the local TCP stack, increasing the maximum window size to 32 MiB. Then, we increased the number of OT rounds performed concurrently, pipelining multiple OT rounds over a single connection, which significantly improved performance (Fig. 11a). Additionally, we explore parallelizing the computation, assigning multiple workers to the same machine, so that multiple TCP flows are used. The results are in Fig. 11b. The dashed line at the bottom is the time to run the experiment with both the garbler and evaluator on Azure (taken from Fig. 8). For the Oregon setup, we can come close to the Local performance using two flows. The Iowa setup is more challenging because less bandwidth is available per flow. Using multiple parallel flows helps, but the performance improvement in the Iowa setup is limited by variation in wide-area flow performance, which induces stragglers.

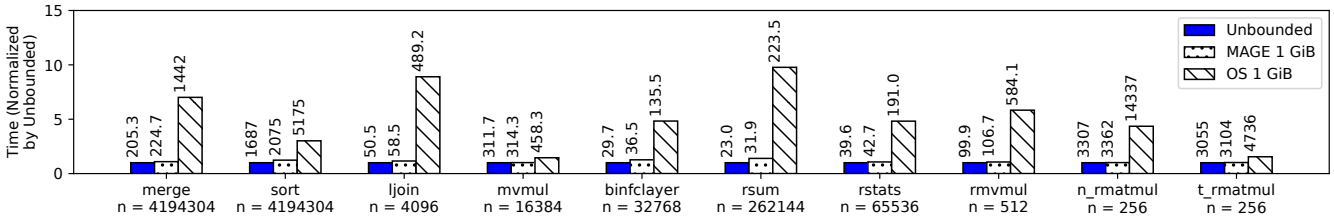
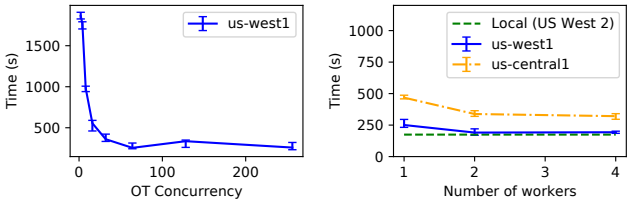


Figure 10: Normalized performance of Unbounded, OS Swapping, and MAGE, parallelized over $p = 4$ workers (per party)



(a) Time to run **merge** vs. number of concurrent OTs

(b) Time to run **merge** vs. number of workers

Figure 11: Wide-area garbled circuit performance in MAGE

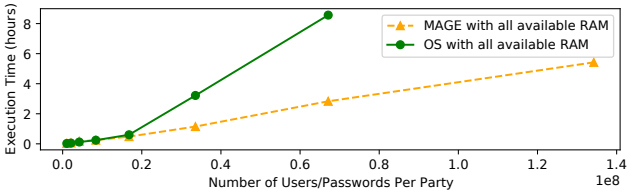


Figure 12: Scaling password reuse detection with MAGE

In both cases, the performance overhead of operating in the wide area is less than the performance overhead of swapping (Fig. 8), indicating that MAGE's techniques confer substantial benefit even in wide-area settings.

8.8 Applications

For these experiments, we did not use `cgroups` to limit RAM. The OS and MAGE setups ran using all of the available RAM.

8.8.1 Detecting Password Reuse

When users reuse a password across multiple websites, they become prone to "credential stuffing" attacks, in which an attacker uses a user's password leaked by one site to compromise that user's account on other sites. To address this problem, sites may wish to identify which of their users reuse their passwords on other sites [81]. Senate [66, Query 2 in §2] proposes a protocol for this. First, the sites arrange to assign user IDs and hash passwords such that they will match *across* sites. Then, they use SMPC to detect which user IDs are shared between the sites and have the same password hash. Note that user IDs and password hashes cannot be shared directly, since they are sensitive (the hashes can be reversed).

We write a two-party version of the password reuse program in MAGE's DSL for garbled circuits, based on Senate's password reuse program. Senate uses a different SMPC protocol, so its results are not directly comparable to ours.

We use MAGE to scale the password reuse program to 2^{27} users per party, which requires 1.125 TiB on each party. A single `D16d_v4` instance does not have enough swap space. Thus, we use four `D16d_v4` instances on Azure for the garbler party, and four `n2-highmem-4` instances on Google Cloud [30] for the evaluator party. As explored in §8.7, we use two workers per instance (total of eight workers per party) to efficiently use wide-area network bandwidth. The results are shown in Fig. 12. For a given time budget, MAGE increases the number of user-password records by $\approx 3\times$. This improvement may have been larger had we been able to obtain `Ddv4`-series instances with a greater swap-space-to-RAM ratio.

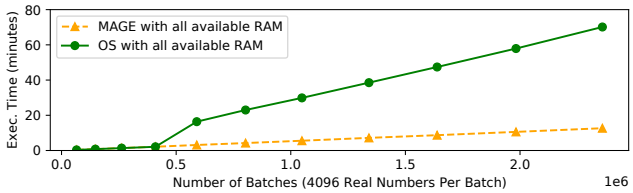


Figure 13: Scaling computational PIR with MAGE

8.8.2 Private Information Retrieval

Private Information Retrieval (PIR) is a family of protocols that allow a user to retrieve a data item at a particular index from a database without the database learning which item was accessed. PIR can be used to support public queries on private data [80]. We evaluate MAGE by using CKKS to instantiate the classic Kushilevitz-Ostrovsky single-server computational PIR scheme [50, §3]. PIR's access pattern is particularly simple—a linear scan over the database—so ad-hoc approaches to prefetching, or multi-threading to improve swap performance, may be quite effective. Our focus is on what MAGE optimizes *automatically*, so we do not include such ad-hoc optimizations in the OS baseline. We use a single worker (thread) to compute the PIR. The database consisted of plaintext data pre-encoded into batches to use with CKKS. We wrote a DSL program that populates the database (with hardcoded elements) and then performs a PIR query on it; the reported measurements are the time to perform the PIR query, not including the time to populate the database. The results are in Fig. 13. For a given time budget, MAGE allows for $\approx 5\times$ as many database elements to be processed.

9 Related Work

Much existing work has looked at high-performance algorithms for SMPC [21, 22, 44, 45, 84] and HE [17, 29]. These works focus on the cryptography, not how to manage a computer's resources to perform large computations efficiently.

A complementary line of work explores tailoring SMPC

computations to a specific application [15,43,68,94]. The goal of MAGE is to perform the same computation more efficiently, so its techniques generalize across different applications. For an application, one may first simplify the computation using application-specific observations, and then execute the resulting computation as efficiently as possible.

Research works including Fairplay [55], HEKM [37], KSS [49], MLB [61], PCF [48], and TinyGarble [73] are frameworks for garbled circuit execution. We described many of them in §2.4. One work [11] explores parallelizing execution of a garbled circuit, using programming language tools to automatically extract parallelism. None of them explore how to efficiently swap memory to storage, as MAGE does.

There already exist many DSLs and compilers for SMPC [34, 36, 51, 60, 82, 89, 93] and HE [13, 23, 78]. These tools often aim to make SC more accessible to non-expert developers, by automatically optimizing the SC program. MAGE addresses the complementary problem of executing the resulting SC circuit more efficiently. To use an existing tool with MAGE (as in Fig. 2), one could modify it to output its optimized circuits in one of MAGE's DSLs, and then run MAGE's planner on that DSL code. Alternatively, one could modify the tool to output a bytecode directly usable by MAGE's planner (e.g., the "Virtual Bytecode" in Fig. 4).

AIFM [70] uses similar C++ language features as MAGE's DSLs. AIFM uses them at runtime for fine-grained memory management. In contrast, MAGE (1) executes DSL programs only to extract the memory access pattern during the planning phase and (2) manages memory at the granularity of pages.

There is an extensive literature concerning memory management in traditional operating systems [3, 4, 24–26]. A related line of work looks at how operating systems can give memory-intensive applications, such as scientific simulations, more control over paging [32]. While these works focus primarily on paging in the classic sense, our work explores memory programming. Additionally, our work, unlike scientific simulations, is capable of *general* computations within SC. Scheduling page movement according to real-time constraints imposed by computation also draws from the real-time scheduling literature [52]. These techniques do not manage memory directly and are complementary to ours.

Some systems in other domains, like neural network training, formulate memory management problems as an integer linear program and use an exponential-time solver [40]. This approach exploits the high-level structure of the application to coarsen the dataflow graph. For MAGE, the dataflow graph is much larger because *general* SC computations do not conform to any particular high-level structure. By operating on a program representation of the circuit (§4.2), MAGE does coarsen the graph, but it nevertheless remains enormous. Thus, we use our staged approach (§6) to find a good approximation.

Some systems use observations of past memory accesses or past working sets (e.g., from prior invocations of a program) to perform targeted prefetching [33, 35, 56, 77, 92] and approximate Belady's algorithm (MIN) [72]. SC's obliviousness and our memory programming approach allow MAGE to compute the memory access pattern without first running the program, and then apply these techniques using the access pattern itself.

The recent DEMAND-MIN [39] algorithm combines MIN with prefetching. DEMAND-MIN tells which item to evict given an access pattern sequence and prefetch sequence fixed in advance. It is not directly applicable to MAGE because MAGE's prefetch sequence is not fixed in advance.

At a technical level, MAGE's planning is similar to register allocation in compiler theory [14, 18, 74, 85]—variables, registers, and memory in register allocation correspond to wire values, slots in memory, and storage swap space in the context of MAGE. The key difference is that register allocators must deal with conditional branches whose outcomes cannot be predicted at compile time. From the perspective of register allocation, the entire circuit that MAGE operates on would be viewed as a single basic block. We discussed a result from register allocation theory for a single basic block in §6.3. Another result is that, for a *fixed* number of registers, there is a linear-time algorithm that can reorder instructions within a structured program to optimize its register allocation [7, §3.2] (though the time is exponential in the number of registers).

10 Conclusion

This paper explores how to efficiently execute SC computations that do not fit in memory. Our key observation is that SC is inherently oblivious. This enables memory programming, in which one computes the access pattern of an SC program in advance and uses it to produce a memory management plan. By using memory programming to preplan data transfers between memory and storage, MAGE runs SC up to an order of magnitude faster than the OS virtual memory system and can execute some SC programs at nearly in-memory speeds.

Some non-SC programs, like plaintext neural network inference and programs designed for hardware enclaves like Intel SGX, are also oblivious. Applying memory programming to such workloads is an interesting direction for future work.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nadav Amit, for their helpful feedback. We would also like to thank Katerina Sotiraki and other students/postdocs from the RISELab Security Group for their feedback on early drafts.

This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Bakar Fellows Program, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This research is also supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Bater, G. Elliott, V. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure querying for federated databases. *VLDB*, 10(6), 2017.

[2] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*. ACM, 1990.

[3] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5(2), 1966.

[4] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *CACM*, 12(6), 1969.

[5] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*. IEEE, 2013.

[6] O. Biçer. Efficiency optimizations on Yao's garbled circuits and their practical applications. Master's thesis, Istanbul Şehir University, 2017. Chapters 3 and 4.

[7] H. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers. In *SODA*. SIAM, 1998.

[8] J. Bonwick. The slab allocator: An object-caching kernel. In *USENIX Summer Technical Conference*. USENIX Association, 1994.

[9] D. P. Bovet and M. Cesati. Page frame reclaiming. In *Understanding the Linux Kernel*, chapter 17, page 679. O'Reilly Media, 2006.

[10] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. Cryptology ePrint Archive, Report 2019/448, 2019. https://eprint.iacr.org/2019/448.

[11] N. Buescher and S. Katzenbeisser. Faster secure computation through automatic parallelization. In *USENIX Security*. USENIX, 2015.

[12] Cape Privacy. https://medium.com/dropoutlabs.

[13] S. Carpov, P. Dubrulle, and R. Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *SCC*. ACM, 2015.

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1), 1981.

[15] H. Chen, M. Kim, I. P. Razensteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously secure matrix multiplication with applications to private deep learning. 2020. https://eprint.iacr.org/2020/451.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*. Springer, Cham, 2017.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*. Springer, Berlin, Heidelberg, 2016.

[18] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *CC*. Springer, Berlin, Heidelberg, 1998.

[19] Curv. Curv | digital asset security infrastructure. https://www.curv.co/.

[20] D. McDaniel. Virtual machines best practices: Single VMs, temporary storage and uploaded disks. https://azure.microsoft.com/en-us/blog/virtual-machines-best-practices-single-vms-temporary-storage-and-uploaded-disks/, 2014.

[21] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In *ESORICS*. Springer, Berlin, Heidelberg, 2013.

[22] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

[23] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. ACM, 2020.

[24] P. J. Denning. Thrashing: its causes and prevention. In *AFIPS*. ACM, 1968.

[25] P. J. Denning. Virtual memory. *CSUR*, 2(3), 1970.

[26] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, SE-6(1), 1980.

[27] Duality. https://dualitytech.com/.

[28] M. Farach and V. Liberatore. On local register allocation. In *SODA*. SIAM, 1998.

[29] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2012.

[30] Google Cloud. Machine types. https://cloud.google.com/compute/docs/machine-types.

[31] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu. Better concrete security for half-gates garbling (in the multi-instance setting). Cryptology ePrint Archive, Report 2019/1168, 2019. https://eprint.iacr.org/2019/1168.

[32] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS*. ACM, 1992.

[33] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *ICML*, 2018.

[34] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *S&P*. IEEE, 2019.

[35] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. I/O acceleration with pattern detection. In *HPDC*. ACM, 2015.

[36] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *CCS*. ACM, 2012.

[37] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. USENIX, 2011.

[38] Inpher. https://inpher.io/.

[39] A. Jain and C. Lin. Rethinking belady's algorithm to accommodate prefetching. In *ISCA*. ACM/IEEE, 2018.

[40] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, 2020.

[41] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. IEEE, 2008.

[42] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. ACM, 2018.

[43] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*. USENIX, 2018.

[44] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*. ACM, 2016.

[45] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*. Springer, Cham, 2018.

[46] Keyless. Keyless | zero-trust passwordless authentication. https://keyless.io/.

[47] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*. Springer, Berlin, Heidelberg, 2008.

[48] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security*. USENIX, 2013.

[49] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*. USENIX, 2012.

[50] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single databse, computationally-private information retrieval. In *FOCS*. IEEE, 1997.

[51] C. Liu, X. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*. IEEE, 2015.

[52] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973.

[53] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS*. ACM, 2017.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*. ACM, 2005.

[55] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security*. USENIX, 2004.

[56] H. Al Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In *ATC*. USENIX, 2020.

[57] Microsoft Azure. Ddv4 and Ddsv4-series. https://docs.microsoft.com/en-us/azure/virtual-machines/ddv4-ddsv4-series, 2020.

[58] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*. USENIX, 2020.

[59] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P*. IEEE, 2017.

[60] B. Mood, D. Gupta, H. Carter, K. R. B. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P*. IEEE, 2016.

[61] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *FC*. Springer, Berlin, Heidelberg, 2012.

[62] J. Nielsen. Nielsen's law of Internet bandwidth. Accessed: May 26, 2020.

[63] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S&P*. IEEE, 2013.

[64] T. Peng. Shared machine learning: Ant financial's solution for data privacy. https://medium.com/syncedreview/shared-machine-learning-ant-financials-solution-for-data-privacy-8069cffe7bb6.

[65] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*. Springer, Berlin, Heidelberg, 2009.

[66] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *USENIX Security*. USENIX, 2021.

[67] B. Randell. A note on storage fragmentation and program segmentation. *CACM*, 12(7), 1969.

[68] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*. USENIX, 2019.

[69] M. Rosulek. A brief history of practical garbled circuit optimizations, 2015. https://simons.berkeley.edu/talks/mike-rosulek-2015-06-09, https://www.youtube.com/watch?v=FTxh908u9y8.

[70] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*. USENIX, 2020.

[71] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, 2020. Microsoft Research, Redmond, WA.

[72] Z. Song, D. S. Berger, K. Li, and W. Lloyd. Learning relaxed Belady for content distribution network caching. In *NSDI*. USENIX, 2020.

[73] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*. IEEE, 2015.

[74] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN*. ACM, 1998.

[75] Unbound. https://www.unboundtech.com/.

[76] Laakeri (https://cs.stackexchange.com/users/95646/laakeri). Is there an algorithm to minimize working set during a topological traversal? Computer Science Stack Exchange, 2020. https://cs.stackexchange.com/q/120274.

[77] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*. ACM, 2021.

[78] A. Viand, P. Jattke, and A. Hithnawi. SoK: Fully homomorphic encryption compilers. In *S&P*. IEEE, 2021.

[79] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: secure multiparty computation on big data. In *EuroSys*. ACM, 2019.

[80] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*. USENIX, 2017.

[81] K. C. Wang and M. K. Reiter. How to end password reuse on the web. In *NDSS*. Internet Society, 2019.

[82] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[83] X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*. ACM, 2017.

[84] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *CCS*. ACM, 2017.

[85] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE*. ACM, 2005.

[86] S. Yakoubov. A gentle introduction to Yao's garbled circuits, 2017. http://web.mit.edu/sonka89/www/papers/2017ygc.pdf.

[87] A. C.-C. Yao. Protocols for secure computations. In *FOCS*. IEEE, 1982.

[88] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*. IEEE, 1986.

[89] S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://eprint.iacr.org/2015/1153.

[90] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2015.

[91] Zcash. Parameter generation. https://z.cash/technology/paramgen/.

[92] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *VEE*. ACM, 2011.

[93] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In *USENIX Security*. USENIX, 2021.

[94] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *S&P*. IEEE, 2019.

[95] R. Zhu, D. Cassel, A. Sabry, and Y. Huang. NANOPI: Extreme-scale actively-secure multi-party computation. In *CCS*. ACM, 2018.

A Artifact Appendix

Abstract

Our artifact consists of a MAGE prototype and scripts to use it to run our experiments from §8. The MAGE prototype can execute SC efficiently even when the computation does not fit in memory. It does so by using memory programming to provide a very efficient virtual memory abstraction. Our prototype supports distributing an SC computation across workers that communicate over the network, allowing for parallel and distributed SC execution. The MAGE prototype presently supports two SC protocols: garbled circuits and CKKS. It follows the layered architecture described in §4.3.

Scope

Our artifact can be used to validate our central claim that, using memory programming, MAGE can execute SC computations that do not fit in memory at nearly in-memory speeds. Specifically, our artifact can be used to validate the performance claims made in the figures and table in §8. Our submitted artifact package allowed the artifact evaluation committee to reproduce those results present in our submitted paper; we have since added support for reproducing the measurements we have added since the original submission.

Our artifact can also be used to run SC computations unrelated to our evaluation of MAGE in §8. The user can describe a custom SC computation using a DSL internal to C++, and then use our MAGE prototype to generate a memory program for it and execute it efficiently.

Contents

Our artifact comprises (1) a prototype of MAGE and (2) scripts to run experiments from §8.

Prototype. Our MAGE prototype includes:

- The planner and interpreter for the MAGE system.
- A utility program to read the bytecode format used by our implementation and print a memory program in human-readable form.
- Implementations of the workloads used in our evaluation (§8.1) in MAGE's DSLs.
- Utility programs to prepare inputs for these workloads.
- A wiki page that walks the user through using our MAGE prototype to perform a computation.

Scripts. Our scripts to run our experiments include:

- A program, `magebench.py`, that can spawn cloud instances on Microsoft Azure and Google Cloud and run experiments on the resulting cloud setup. The command line parameters passed to this program can be used to specify the cloud setup and experiments to run; the user can change these command line parameters to change aspects of the setup (e.g., number of workers, memory size, problem size, etc.).
- A README file that describes how to use `magebench.py` to run our experiments from §8 and obtain log files containing the results.
- An IPython notebook to produce graphs from the log files output by `magebench.py`.
- Utility scripts to help automate invoking `magebench.py` to run experiments from §8.

Hosting

Our artifact is available on GitHub. Our MAGE prototype is available at https://github.com/ucbrise/mage and our scripts to run our experiments are available at https://github.com/ucbrise/mage-scripts. The version we provided to the artifact evaluation committee is marked in both repositories using the `osdi21ae` tag. However, we encourage users to use the latest versions of each repository (on the `main` branch), as they include the newest features and bug fixes, including scripts for additional experiments in §8.

Requirements

We developed and tested our artifact on Intel x86-64 systems running Ubuntu 20.04. We used `clang++` 10.0.0 to compile our MAGE prototype. The `magebench.py` script spawns cloud instances with an environment appropriate for building and running our MAGE prototype. Spawning those cloud instances requires a subscription to Microsoft Azure and Google Cloud. The particular software dependencies for our artifact are specified in the README files of our two GitHub repositories.

Workflow

To use our MAGE prototype, the user first writes a configuration file in YAML describing the execution setup (e.g., network information and swap file for each worker, number

of concurrent OTs for garbled circuits, etc.). For SMPC, information needed only by other parties (e.g., the swap file for other parties' workers) can be omitted from the configuration file. Next, the user writes a program in a DSL internal to C++ specifying the computation to run. Then, the user runs MAGE's planner, which accepts the DSL program and configuration file as input, for each worker the user will run, and outputs a file containing a memory program for each worker. The user prepares a file for each worker describing that worker's input for the computation. Finally, the user runs MAGE's interpreter for each worker, which accepts files containing the memory program, configuration, and input data and writes a file containing the program's output. Further details are given in the README file and wiki pages of the `mage` repository on GitHub.

To use our script to run experiments, the user invokes `magebench.py` to spawn cloud virtual machines. The user can then invoke `magebench.py` to run MAGE on those cloud virtual machines, copy the resulting log files to the machine where `magebench.py` is run, and finally, deallocate the cloud virtual machines. Further details are given in the README file of the `mage-scripts` repository on GitHub.

Zeph: Cryptographic Enforcement of End-to-End Data Privacy

Lukas Burkhalter*, Nicolas Küchler*, Alexander Viand, Hossein Shafagh, Anwar Hithnawi

ETH Zürich

Abstract

As increasingly more sensitive data is being collected to gain valuable insights, the need to natively integrate privacy controls in data analytics frameworks is growing in importance. Today, privacy controls are enforced by data curators with full access to data in the clear. However, a plethora of recent data breaches show that even widely trusted service providers can be compromised. Additionally, there is no assurance that data processing and handling comply with the claimed privacy policies. This motivates the need for a new approach to data privacy that can provide strong assurance and control to users. This paper presents Zeph, a system that enables users to set privacy preferences on how their data can be *shared* and *processed*. Zeph enforces privacy policies cryptographically and ensures that data available to third-party applications complies with users' privacy policies. Zeph executes privacy-adhering data transformations in real-time and scales to thousands of data sources, allowing it to support large-scale low-latency data stream analytics. We introduce a hybrid cryptographic protocol for privacy-adhering transformations of encrypted data. We develop a prototype of Zeph on Apache Kafka to demonstrate that Zeph can perform large-scale privacy transformations with low overhead.

1 Introduction

The availability of rich data and the advancement of tools and algorithms to process data at scale has enabled tremendous innovations in various fields ranging from health and retail to agriculture and industrial automation [68, 79, 81]. However, the accumulation of sensitive data has made service providers hosting data lakes a desirable target for attacks. In addition, a surge of incidents of unauthorized data monetization, instrumentation, and sharing has raised societal concerns [50, 85]. This has pushed regulatory bodies to enact data privacy regulations to prevent misuse of private data and ensure the privacy of personal data [2, 3]. Today, the most integral parts of existing data protection systems are security controls such as authentication, authorization, and encryption which protect data by guarding it and limiting unnecessary exposure. Security controls alone, however, are not sufficient. We ultimately need to ensure that user's privacy is respected even by entities authorized to use the data. Thus, privacy solutions that control the extent of what can be *inferred* [15] from data and protect *individuals' privacy* [45] are crucial if we are to continue to extract utility from data safely.

Today's Data Privacy Landscape: The advent of new data privacy regulations such as GDPR and CCPA, coupled with the increasing importance of data, has led to a growing demand for privacy solutions that protect sensitive data while retaining its value. Despite recent advancements in privacy enhancing technologies [41, 78, 84], privacy frameworks [26, 43, 44, 61, 76, 82] remain shaped by regulatory requirements that predominately focus on the notion of *notice and consent* [5, 11, 59]. Though an essential step towards transparency and user control, it is important to emphasize that user consent is not the answer to data privacy. Bad practices in data use and sharing remain pervasive in consent-based systems [48, 57, 67], and often consent does not adequately express the complexities of real-world privacy preferences. The status quo has three shortcomings that we aim to address with this work: *(i) Trusted data curators:* In the current model, privacy controls are implemented and enforced by data curators who have full access to data in the clear. Frequent data breaches [25, 38, 66] have shown that even trusted providers can be compromised or fall prone to data misuse temptations. Additionally, there are no assurances that data processing actually complies with the stated privacy policies. Consequently, there is a need for built-in data privacy mechanisms that do not require data curators to access data in the clear. *(ii) Lack of user control:* Though privacy regulations mandate services to grant users more control over their data, the materialization of this has been disappointing in practice. Services have been drafting privacy policies that unilaterally dictate how users' data will be used. Users have no option

*These authors contributed equally to this work.

to exert their data privacy preferences except to give blanket consent if they choose to use the service [49, 59]. *(iii) End-to-end privacy:* Privacy solutions today are mostly ad hoc efforts [14] rather than an integral part of the data processing ecosystem. We need a cohesive end-to-end approach to data privacy that follows data from source to downstream. Such solutions should integrate with existing data processing and analytics frameworks and coexist with data protection mechanisms already in place.

Zeph: In this work, we propose Zeph, a new data privacy platform that provides the means to safely extract value from encrypted data while ensuring data confidentiality and privacy by serving only privacy-compliant data. Zeph addresses the above shortcomings with two key ideas: *(i)* a user-centric privacy model that enables users to express their privacy preferences. In Zeph, a user can authorize services to access raw data or privacy-compliant data securely. This aligns with data sharing practices claimed in privacy policies today: e.g., "we share or disclose your personal data with your consent" or "we only provide aggregated statistics and insights" [6, 13]. In addition to this commonly referenced aggregation policy, Zeph supports more advanced privacy-compliant data transformations. For example, transformations that restrict what can be inferred from the data (e.g, generalization techniques [24, 72, 78]) or ensure differential privacy – a mathematically rigorous definition of privacy. *(ii)* Zeph cryptographically enforces privacy compliance and executes privacy transformations on-the-fly over encrypted data, ensuring that the generated transformed views conform to users' privacy policies.

The design concepts underpinning Zeph are generic and could be adapted to other systems. In this work, we specifically target data stream analytics/processing pipelines and build on the typical structure of such systems. Hence, we focus on cryptographic building blocks that optimize efficiency for this type of data. Streaming compute tasks are increasingly relevant in various privacy-sensitive sectors [17, 35, 47, 55]. The online nature of stream processing makes low latency and high throughput critical requirements for privacy-preserving stream processing solutions.

Cryptographically Enforced Privacy Transformations. There are three key challenges in designing a data platform that enables privacy-compliant data transformations on encrypted data. First, we need to ensure compatibility with the data flow of existing data processing pipelines (e.g., storage and compute) and meet their strict performance requirements. Second, the platform must enable a wide range of existing privacy transformations and allow for different transformations to be applied to the same underlying data. Finally, in addition to single-source privacy transformations, we need to support transformations that require combining data from multiple users (e.g., aggregate private data releases).

Existing practical encrypted data processing systems generally use partially homomorphic encryption schemes that already support the single-source privacy transformations required in our system [33, 39, 54, 69, 70, 80]. However, homomorphic evaluation alone is insufficient to support aggregations across data from different users. Supporting these functions is typically achieved via multi-party computation protocols that are optimized for aggregation operations [16, 39, 63, 73]. These protocols ensure that user inputs remain private and only the aggregation result is revealed to the server. However, these protocols are either limited to specific functions (e.g., updating sketches) or require the data producers to take an active part in the computation.

We address these challenges in Zeph using two ideas: *(i)* a new approach for encryption that decouples data encryption from privacy transformations. This logical separation of the data and privacy plane allows us to remain compatible with data flows in existing systems. Data producers remain oblivious to the transformations and do not need to encrypt data towards a fixed privacy policy. *(ii)* we introduce the concept of cryptographic *privacy transformation tokens* to realize flexible data transformations. These tokens are, in essence, the necessary cryptographic keying material that enables the respective transformation on encrypted data. Zeph creates these tokens via a hybrid construction of secure multi-party computation (MPC) and a partially homomorphic encryption scheme. Outputs of privacy transformations over encrypted data at the server-side are then released by combining the encrypted data with corresponding cryptographic transformation tokens.

We have built a prototype of Zeph[1] that is interfaced with Apache *Kafka* [21]. Our evaluation results show that Zeph can serve real-time privately transformed streams in different applications with a 2x to 5x latency overhead compared to plaintext. We optimize the interactive part of the underlying MPC protocol with ideas from graph theory to achieve the scalability requirements of Zeph. Our optimization improves performance up to 55x compared to the baseline.

2 Overview

In this section, we discuss end-to-end privacy and its requirements, give an overview of Zeph, and describe our security and privacy model.

2.1 End-to-End Privacy

In this work, we investigate a new cohesive end-to-end design for data privacy. Despite being heavily intertwined with users' data, data systems have evolved with design objectives centered around availability, performance, and scalability, while privacy is essentially overlooked. As privacy becomes a more urgent concern, we need system designs that retrofit privacy into existing established data framework designs. Embedding

[1] Zeph's code available at: https://github.com/pps-lab/zeph-artifact

Name	Zeph	Description
		DATA MASKING
Field Redaction [7, 9, 11]	●	Reveal some attributes and hide others
Predicate Redaction [7]	◐	Only reveal data that satisfy a predicate
Det. Pseudonym. [12]	○	Replace value with a deterministic pseudonym
Rand. Pseudonym. [12]	●	Replace value with a random pseudonym
Shifting [4]	●	Shift actual values by a fixed offset
Perturbation [41]	●	Perturb data by adding random noise i.e., additive differential privacy mechanism
		DATA GENERALIZATION
Bucketing [4, 11]	◐	Map values to a coarse space
Time Resolution [33]	●	Aggregate data across time
Population [28, 37, 39, 73]	●	Aggregate data across a population

Table 1: Overview of existing privacy transformations: Data Masking techniques (top), Data Generalization techniques (bottom). We use ● (full support), ◐ (partial support), and ○ (no support) to indicate which of these techniques are currently supported in Zeph.

privacy in the current complex, data-rich systems while ensuring the desired level of utility is, however, challenging. What is considered an appropriate privacy/utility balance in one context might not be a proper trade-off for another context. Therefore, end-to-end system designs for privacy need to account for various privacy solutions and accommodate heterogeneous privacy preferences. Next, we discuss some key design aspects for realizing end-to-end data privacy.

User-Centric Privacy. Users' perception of privacy varies widely across individuals, cultures, and contexts. Therefore, the system needs to provide the means for users to set their privacy preferences and define how their data can be accessed, processed, and shared. In practice, user preferences can also vary with respect to the trade-offs between increased privacy and utility. Their preferences can vary based on the data involved and the target consumer. While we want to offer users the option of strong privacy guarantees, we also need to provide options for more relaxed privacy guarantees when incentives to do so exist. For example, users might voluntarily share their off-platform shopping activities with a service provider in return for financial incentives [71]. Therefore, a practical system needs to support a range of privacy preferences and be able to build privacy-compliant views across data covered by heterogeneous policies.

Retrofit into Existing Data Pipelines. A practical privacy solution should augment existing data pipelines while ensuring the privacy of the underlying data. Additionally, privacy transformations need to respect/adhere to traditional data protection mechanisms already in place (i.e., end-to-end encryption). Therefore, the design needs to offer composability to support a variety of privacy solutions and ensure that privacy solutions can work with encrypted data. We want to leave the flow of data in end-to-end encrypted systems intact.

Privacy Transformations. Privacy solutions for data analytics focus on allowing the use of data or computation on data subject to privacy restrictions specified by users (e.g., restrict

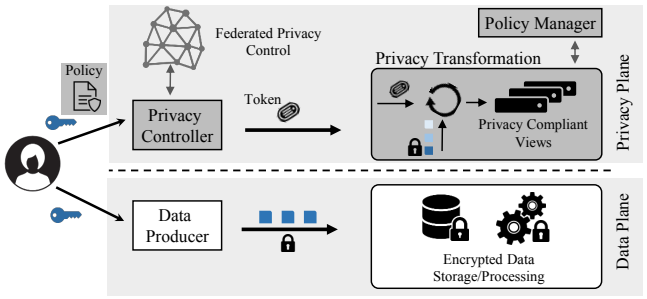


Figure 1: Overview of Zeph's end-to-end approach to privacy.

what can be inferred from the data). They are designed to enable extracting the utility from data while preserving individual's privacy preferences. This is often achieved through a range of data modifications that we refer to as privacy transformations, i.e., functions applied to the data to limit and control the extent of sensitive information revealed to authorized parties. Solutions in this space can be grouped into three broad classes (Table 1): *(i)* data masking techniques that obfuscate sensitive parts of the data, *(ii)* generalization techniques that reduce data fidelity, e.g., by aggregating data, *(iii)* combinations of *(i)* and *(ii)* which can realize complex transformations by chaining masking and generalization techniques. Privacy transformations are the primary tools to safely release data, achieving either a range of privacy guarantees common in practice (e.g., as in aggregate statistics) or formal privacy definitions such as k-anonymity [78] or differential privacy [40]. A useful end-to-end system design for privacy therefore needs to support a broad set of existing privacy transformations.

2.2 Zeph in a Nutshell

Zeph is a privacy platform that augments encrypted stream processing pipelines with the means to enforce privacy controls cryptographically. Figure 1 shows an overview of Zeph's design. We aim to enable authorized third-party services to access and process data and to gain insights from it without violating the privacy preferences of the data owners. We design Zeph to encapsulate state-of-the-art privacy solutions (e.g., generalization, differential privacy) while preserving the data flow in existing streaming pipelines.

Privacy Plane. To illustrate a deployment of Zeph, we consider a health monitoring provider that stores health-related data from wearable devices such as heart rate and other metrics. We assume that the data streams are already end-to-end encrypted, i.e., the wearables encrypt data before uploading, while applications (e.g., health dashboard) query encrypted data and locally decrypt the result [33]. We refer to this data flow through a streaming platform as the *data plane*. The privacy logic resides and is executed outside of the data plane, allowing *data sources* to continue writing encrypted data streams to a remote stream processing pipeline as before.

Zeph exposes an API for *data owners* to set their privacy preferences, which forms the base for users' privacy policies. A Zeph deployment augments the data plane with a *privacy plane* that enables the provider to extract information from the encrypted data streams through privacy transformers. Hence, the service gets access to privacy-compliant transformed views of the underlying raw data streams. For example, the service might collect the average heart-rate per day for different age-groups (i.e., population aggregate transformations). To collect these statistics, Zeph allows the service to express privacy options for data stream attributes through a modified data schema (§4.1), which describes a set of possible privacy transformations for attributes. Upon registering with the services, *data owners* set their privacy preferences for each stream based on these options, forming the base for users' privacy policies. For example, they can indicate if the service is allowed to include their heart rate stream in the specified aggregate transformations. The additional logic for handling privacy options and coordinating transformations is handled by an additional server component, the *policy manager*. The *policy manager* offers an API to handle privacy options per data stream and coordinates privacy transformations as *stream processors* in the streaming pipeline.

Privacy Controller. Policy enforcement in Zeph is handled by the *privacy controller*. The privacy controller is responsible for supplying the cryptographic privacy transformation tokens that enable privacy transformations at the server. As some privacy policies require data to be aggregated across different users before being made available, generating tokens specific to these types of transformations require interaction between several privacy controllers in what we refer to as *federated privacy control*. While the tokens generated by the privacy controllers cryptographically enforce the data owners' privacy policies, the server is responsible for composing and executing transformations efficiently. The *privacy controller* does not require access to the data and can be hosted in a location with higher availability guarantees. Zeph allows users to choose a variety of deployment scenarios for privacy controllers. Privacy controllers could be self-hosted, hosted on-premise for corporations, or outsourced to a trusted provider (e.g., OpenID identity providers).

Data Consumers. We distinguish between two types of *data consumers*: (a) services that access the data to provide utility to the user (i.e., personalization), and (b) third-party services, e.g., to provide a utility that is beneficial to the public or the service itself, but not directly to the user (e.g., allow your health data records to contribute to a medical study). Enabling direct access to the data for the first type of data consumers is handled by cryptographic access control and is supported in our design, but it is not the focus of this work. In Zeph, we focus instead on the latter with the goal to continue enabling the benefits of these services while respecting users' privacy.

2.3 Threat Model

Zeph enforces users' privacy preferences cryptographically, i.e., users are guaranteed that the data is transformed with the privacy transformation corresponding to their privacy preference before it is released to applications. Meanwhile, their original data remains end-to-end encrypted.

Setting. We assume an *honest-but-curious* [70] server, i.e., the server performs the computations correctly but will analyze all observed data to gain as much information as possible. We also assume the existence of a public-key infrastructure (PKI) for authentication of privacy controllers/data producers. In this setting, Zeph ensures *data confidentiality*, more specifically *input privacy*, guaranteeing that the adversary learns nothing about the raw data streams except what can be learned from the output of the transformation F (i.e., $\hat{F}$*-privacy* [39]) with some modest leakage function due to encodings (§3). Zeph also ensures that an adversary controlling the server and at most a fraction α of privacy controllers is unable to violate the privacy policies of other data owners.

Data Plane. In Zeph, data streams are encrypted at the source with a semantically secure encryption scheme, while the metadata (e.g., timestamps) is sent in plaintext. Decryption keys are never disclosed to the server; therefore, raw data confidentiality is guaranteed even in the case of a server compromise. If an adversary gains control over a data producer or the responsible privacy controller, only the data associated with that producer/controller is revealed.

Privacy Plane. Zeph ensures input privacy for honest data owners even if the stream processor executing a privacy transformation or the policy manager coordinating it is compromised by an adversary. For the case where F is an aggregation function involving data from different privacy controllers (i.e., federated privacy control), we assume that at most a fraction of α of the entities in the aggregation transformation are controlled by the adversary. Note that this can also include the server. The choice of α depends on the deployment scenario. In (§3.4), we show how this choice affects performance. For our evaluation, we use a pessimistic value of $\alpha = 0.5$, but real-world deployments might use significantly lower values.

Robustness. While Zeph can handle various failures in practice, formal robustness against misconfigured or malicious privacy controllers or data producers is out of scope for this design. A privacy controller sending corrupted tokens cannot compromise privacy but could alter the output of a transformation or prevent a transformation from completing.

3 Encryption for Privacy Transformations

In this section, we describe our approach to enable privacy transformations in end-to-end encrypted systems. Our design serves privacy-compliant transformed views of data without affecting the data flow of an end- to-end encrypted stream-processing system. To meet this goal, our design

logically decouples privacy transformations and policy enforcement from the generation and storage of data. Data producers remain oblivious to the transformations and do not need to encrypt data towards a fixed privacy policy. The modifications needed for privacy transformations are instead executed outside the data plane (i.e., conventional data flow), working exclusively on encryption keys to generate what we call cryptographic *transformation tokens*. These tokens are, in essence, the necessary cryptographic keying material that enables the respective transformation on encrypted data. Outputs of privacy transformations over encrypted data at the server-side are then released by combining the encrypted data with corresponding cryptographic transformation tokens. Introducing a logical separation between the data plane and privacy plane allows for heterogeneous policies atop the same data and leaves the conventional data flow unaffected. In this realization of Zeph we focus on streaming data. Hence, we focus on cryptographic building blocks that optimize efficiency for this type of data. The design concepts underpinning Zeph are generic and could be adapted to other systems using other cryptographic building blocks. These, however, can introduce their own trade-offs between computation expressiveness and performance.

3.1 Decoupling Encryption from Privacy Transformations

This design requires an encryption approach that supports *homomorphic evaluation* in a variety of settings. Namely, it needs to support: *(i)* evaluation on encrypted data, i.e., encrypted data processing, *(ii)* construction of cryptographic *transformation tokens*, and *(iii)* combining the encrypted data with the matching cryptographic transformation tokens for selective release of privacy-compliant transformed data views.

Encrypted Data Processing. Existing encrypted data processing systems utilize homomorphic encryption schemes to enable server-side computation on encrypted data [33, 69, 70, 80]. To meet applications' stringent performance requirements, systems typically combine efficient partially homomorphic encryption schemes [27, 33, 39, 54, 69] with specialized client-side encodings to support a wider set of queries. However, standard homomorphic encryption schemes do not lend themselves to support selective release of data (i.e., handing out the decryption key allows to decrypt all data) or support privacy transformations that require data evaluation across different users (i.e., different trust domains). Supporting functions across populations in the multi-client/single-server setting is typically achieved via specialized multi-party computation protocols [16, 39, 63, 73]. These existing protocols ensure that the user inputs remain private and only the output of the function evaluation is revealed to the server. However, they require active participation by the data producers and are often limited to specific functions. We want to remove the need for – potentially resource-limited – data producers to take part in or even be aware of privacy transformations.

Homomorphic Secret Sharing. To decouple encryption from privacy transformations, we draw on ideas from the Homomorphic Secret Sharing (HSS) [31, 32] literature. In essence, HSS allows computing a function F on secret shared messages by combining the outputs of a function $\hat{F}$ applied on the individual secret shares. HSS could be used to split stream events into two shares: one for the data plane (server) and one for the privacy plane. The privacy plane could authorize a transformation F by computing the same function $\hat{F}$ as the server on their local input shares, and releasing the output. Here, $\hat{F}$ supports all of the required core functions, as secret shares can also be aggregated across different data owners. Applying standard HSS in our setting raises two issues: *(i)* general-purpose HSS incurs non-negligible overhead [31], and *(ii)* with this approach privacy controllers remain dependent on data producers as they continue to receive a secret share for each new stream event.

To address the first issue, we employ *additively* homomorphic secret sharing. This is considerably more efficient than general-purpose HSS, allowing our system to sustain the high throughput needed for streaming data workloads. Used naively, additively homomorphic secret sharing can significantly limit expressiveness. However, as we show in the next section, using carefully selected data encodings allows us to support a wide set of privacy transformations.

To break the dependency between privacy controllers and data producers, we enable the privacy controller to independently derive the tokens (i.e., its shares) based only on metadata about the stream. Given a shared common master secret, the data producer and privacy controller never have to communicate or even be online at the same time. We introduce our scheme in more detail in §3.3. Our tailored scheme offers both the required efficiency and the flexibility necessary to decouple the data plane from the privacy plane. The data producer and the responsible privacy controller need to only agree on a shared master secret. Then, the privacy plane can authorize a transformation F by deriving the involved shares and executing $\hat{F}$ on them, which results in a *transformation token*. This token allows the server to compute and reveal the output of F by performing $\hat{F}$ on the ciphertexts and combining the result with the *transformation token*. If the transformation F spans multiple trust domains, i.e., the privacy plane consists of multiple privacy controllers, the privacy controllers run an MPC protocol to compute the final transformation token. Note that this does not require the data producers to participate or even be online. Next, we show how we can support a broad set of privacy transformations with this scheme.

3.2 Privacy Transformation Functions

Broadly, privacy transformations (§2.1) generally involve computation/perturbation of individual user's data, computations across different users' data, or combinations of the two.

Based on this insight, Zeph exposes three core functions for developers that allow for privacy transformations in the encrypted setting: *(i)* Σ_S, which enable ciphertext aggregation operations within the same user's data streams. *(ii)* Σ_M, which enables ciphertext aggregation across streams from a population of users, *(iii)* Σ_{DP}, which supports perturbation via noise addition to streams aggregated across multiple users.

Privacy Transformations in Zeph. A privacy transformation F in Zeph is realized by combining a chain of core functions and/or withholding certain shares when creating a token. *(i) Data Masking.* Data masking techniques such as field-redaction and randomized pseudonymization are directly supported by the secrecy properties of our scheme. The privacy controller redacts or pseudonymizes a field by withholding the corresponding shares from the transformation token. Shifting and perturbation are realized with Σ_S by adding a constant or calibrated random offset to the transformation token. Zeph supports a subset of predicate redactions using client-side encodings that represent a value as a vector of elements. A privacy controller can then construct a transformation token that only reveals a subset of elements in the vector or a certain sum of the elements (Σ_S). For example, to enable predicate redaction based on a threshold, the client would encode the value as a vector of two elements. If the value is above the threshold, the client stores the value as the first element in the vector or else as the second element. To only reveal the values above the threshold, the privacy controller can disclose the first elements of the vectors with the tokens.

(ii) Data Generalization. Bucketing similarly builds on client-side encodings that map a value to a one-hot vector representing the whole domain. Instead of releasing a token for all elements, the privacy controller uses Σ_S to release a sum of shares for elements mapping to the same bucket. For values with a large domain, we can approximate the frequency count with a histogram using a larger bin width. Zeph supports data generalization over time with Σ_S and population with Σ_M. Moreover, Zeph provides Σ_{DP} to release a differentially private aggregate across a population. To extend the supported aggregation functions, we leverage existing encoding techniques [27, 33, 39, 54, 69, 83]. In essence, these encodings map a value to a vector with different statistics that allows the computing platform to execute functions by performing element-wise addition. The aggregate functions sum and count are inherently additions. With a vector of sum and count, a party can obtain the mean by dividing the sum by the count. By adding the square of a value to the encoding vector, a party can calculate the variance using that $Var(x) = \mathbb{E}(x^2) - \mathbb{E}(x)^2$. Moreover, with the one-hot encoding, constructing a histogram corresponds to the element-wise sum of a set of one-hot vectors. Given a histogram, a party can compute the median or other percentiles, min, max, mode, range, or topk. Prior work [39] presents further encoding techniques for other functions that we support in Zeph.

3.3 Transformation Tokens

In Zeph, we build upon a symmetric homomorphic encryption scheme [33] explicitly designed for streaming workloads. We use this scheme to realize efficient additively homomorphic secret sharing for our setting. The scheme efficiently derives a unique sub-key for each message from a master secret and encryption is performed via modular addition of the key and the message. Here, the encrypted message and the (message-specific) sub-key can be seen as additive shares of the message. Since encryption and decryption are linear operations, the scheme supports linear aggregation by computing the function on both the sub-keys and the encrypted messages independently. *Transformation tokens*, which authorize the release of privacy transformation results, are derived from the sub-keys via aggregations. We now describe how these are constructed in our system. We start with a description of a simplified version of Zeph that assumes a single privacy controller and extend our description to consider multiple privacy controllers in §3.4.

Symmetric Homomorphic Stream Encryption. First, we give a brief summary of the symmetric homomorphic stream encryption scheme [33] we build upon. Let a data stream be a continuous stream of events $\{e_0, e_1, ..., e_i, e_{i+1}, ...\}$ for events $e_i := (t_i, m_i)$ consisting of a message and a timestamp. Each message m_i is an integer modulo M and is annotated with a discrete timestamp $t_i \in I$. We assume events are ordered by their timestamps and created in-order.

In the setup phase, a master secret k is generated, the group size M is defined (e.g., 2^{64}), and a keyed pseudo-random function (PRF) $f_k : I \rightarrow [0, M-1]$ that outputs a fresh pseudo-random key k_i for timestamp t_i is selected. To encrypt an event e_i, the data producer uses the event timestamp from the last encrypted message t_{i-1} and computes:

$$Enc(k, t_{i-1}, e_i) = (t_i, t_{i-1}, m_i + k_i - k_{i-1} \mod M) \quad (1)$$

where $k_i = f_k(t_i)$, $k_{i-1} = f_k(t_{i-1})$. This scheme is additively homomorphic: ciphertexts can be aggregated via modular additions. Keys can be aggregated the same way, but for a time-window $[t_i, t_j]$, the client can decrypt more efficiently by deriving only the two outer keys $k_i = f_k(t_i)$ and $k_j = f_k(t_j)$ because the inner keys cancel out. This encryption scheme hides the inputs from the server and allows the server to perform aggregations among the streams without accessing the plaintext data.

Authorizing Transformations. The intuition in Zeph is that the encryption scheme essentially splits a message m_i into two additive secret shares: the key $-k_i + k_{i-1}$ and the ciphertext c_i, with $m_i = c_i + (-k_i + k_{i-1}) \mod M$. Therefore, any transformation F consisting of the three core aggregate operations can be performed independently on both the ciphertexts and the keys using modular additions. The latter produces a *transformation token* τ_F, which the server can use to reveal the output o_F of a transformation F by computing $o_F + \tau_F \mod M$.

Hence, a privacy controller that is in possession of the master keys of the streams can authorize a transformation F by deriving the necessary keys and performing the transformation on top of them to produce a matching *transformation token* τ_F. In the following, we assume that all additions are performed modulo the parameter M.

Single-Stream Transformation Tokens. We now describe how a privacy controller can create transformation tokens for Σ_S transformations, e.g., only reveal the approximate locations aggregated over a month. We start with a window aggregation to reduce the time resolution. The server adds values within the specified time window t_i to t_{i+w} , where w is the window size. As long as data producers submit a value on each window border, the resulting ciphertext of the window aggregation on the server shares has the form $c_w = m_{aggr} + k_{i+w} - k_{i-1}$. The privacy controller can compute the transformation token for this window $\tau = -k_{i+w} + k_{i-1}$ by deriving only the two outer keys $k_i = f_k(t_i)$ and $k_j = f_k(t_j)$ as the inner-keys cancel each other out [33, 69]. With this token, the server can decrypt the window aggregation if and only if the correct windows were aggregated, as the keys directly encode the window range. For aggregations within events, the privacy controller uses modular addition to add the respective sub-keys to create the transformation token. The privacy controller can construct transformation tokens for values with encodings (§3.2) by selectively releasing the sub-keys of certain elements in the encoding vector or by aggregating sub-keys of elements in the vector.

Multi-Stream Transformation Tokens. Multi-stream transformation tokens reveal the output of Σ_M transformations, which aggregate data over multiple streams, e.g., only reveal the approximate location aggregated among multiple users. In multi-stream aggregation, the server sums a fixed window t_i to t_{i+w} across different streams. Let S be the set of streams in the aggregation. For each stream $j \in S$ we have a window aggregated share $c_w^{(j)} = m_{aggr}^{(j)} + k_{aggr}^{(j)}$ where $k_{aggr}^{(j)} = k_{i+w}^{(j)} - k_{i-1}^{(j)}$. The aggregation over all streams in S results in the sum of all window aggregates and the sum of all window share keys. Hence, a privacy controller can compute the transformation token by aggregating the window keys $\tau^{(j)} = -\sum_{j \in S} k_{aggr}^{(j)}$.

Differentially-Private Transformations. Differential Privacy [40] provides formal bounds on the leakage of an individual's private information in aggregate statistics. The most common technique to achieve a differentially private release of information is to add carefully calibrated noise. Zeph supports *noisy* transformations (i.e., Σ_{DP}) on multi-stream window transformations, but could be extended to the single-stream setting. The privacy controllers add carefully calibrated noise to the keys (i.e., submit noisy keys): $\tilde{\tau}_j = \tau_j + \eta_j$ where η_j is the noise. Zeph therefore supports all additive noise mechanisms from the Differential Privacy literature [41] with noise drawn from a divisible distribution. However, mechanisms like the *Sparse Vector Technique* [42] that require access to the underlying data cannot be applied this way. In previous work, noise is added to plaintexts prior to encryption, whereas in Zeph noise is added to the decryption keys. The two approaches are cryptographically equivalent. However, previous work requires deciding on the noise to add at encryption time. Our approach has the advantage of allowing noise to be added to data that was previously encrypted without consideration for noise. This also means, that the same data is reusable for encrypted storage and to facilitate one or multiple differentially private privacy transformations.

3.4 Transformations Across Different Trust Domains

Until now we assumed a single privacy controller that is in control of all streams. We now discuss how Zeph enables multiple privacy controllers that are each responsible for a distinct subset of streams. While we assume that data owners trust their own privacy controller, different data owners might not want to trust the *same* controller. In such multi-trust setting, the server needs to interact with all privacy controllers involved in a transformation. Hence, when aggregating across streams the server needs to request a transformation token from each privacy controller. In a naïve approach, the privacy controllers might simply send a combined token for the aggregation of the streams under their control. However, this leaks the intermediate result from each controller to the server. Instead, we need the individual tokens to reveal no additional information while still enabling correct decryption of the transformation output. We enable this in Zeph using secure aggregation [16, 28], a specialized secure MPC protocol. For our system, we require a secure aggregation protocol that is *(i)* lightweight in terms of computation for privacy controllers and *(ii)* can be efficiently executed multiple times with similar participants. Based on these requirements, Zeph builds on the secure aggregation protocol from Ács et al. [16] to create transformation tokens over multiple parties. The protocol goes hand in hand with the design of the transformation tokens, as it also relies on additive masking. In the following, we outline the core protocol and then describe our optimizations that reduce the computation cost for privacy controllers.

Core Protocol. We consider a set $\mathcal{P}$ consisting of N privacy controllers and a server that aggregates the inputs. Each privacy controller $p \in \mathcal{P}$ owns a token τ_p that is constructed by aggregating the tokens for the corresponding Σ_S transformation for each stream under their control. The goal of the protocol is to compute $\tau = -\sum_{p \in \mathcal{P}} \tau_p$ without revealing the individual inputs τ_p to the server or to the other privacy controllers. Each privacy controller masks its input τ_p with a nonce k_p, i.e., it computes $\tau_p + k_p \mod M$. The nonces are constructed such that the sum over all nonces results in $\sum_{p \in \mathcal{P}} k_p = 0$. As a consequence, the

sum over all encrypted inputs results in the sum of inputs:

$$\sum_{p\in\mathcal{P}} \tau_p + \sum_{p\in\mathcal{P}} k_p \mod M = \sum_{p\in\mathcal{P}} \tau_p \mod M \quad (2)$$

To construct the canceling nonce, each privacy controller establishes $N-1$ pairwise shared secrets $k'_{p,q}$ with all other privacy controllers which are aggregated to form the nonce k_p. In particular, if $p > q$, then the controller p adds $-k'_{p,q}$ else $k'_{p,q}$.

$$k_p = \sum_{p>q} -k'_{p,q} + \sum_{p<q} k'_{p,q} \mod M \quad (3)$$

Hence, the pairwise secrets cancel each other out when the masks are combined in the aggregation. For conciseness, we refer to Ács et al. [16] for a description of dropout handling.

Constructing Canceling Nonces. In Zeph, the secure aggregation protocol is run repeatedly for multiple rounds due to the continuous nature of streaming queries. Thus, privacy controllers require an efficient method to establish many pairwise shared secrets. The standard protocol achieves this with a setup phase where the parties create pairwise shared secrets $k_{p,q}$ using a Diffie-Hellman key exchange. These pairwise secrets then serve as seeds (or keys) for a PRF to establish nonces for each round r: $k^r_{p,q} = PRF(k_{p,q}, r)$. Even though PRF computations are significantly more efficient than a Diffie-Hellman key exchange, this protocol still requires each privacy controller to evaluate $O(N)$ PRF's and additions to create the blinding nonce k_p for a single transformation token, which can be expensive for large N.

To improve this theoretical overhead, we view the complexity of creating a shared blinding nonce as a graph $G = (V, E)$ with the set of vertices V representing the involved parties ($|V| = N$), and the set of edges E denoting the pairwise canceling masks $k'_{p,q}$. In the standard form described above, the graph G forms a Clique because every privacy controller includes a pairwise mask $k'_{p,q}$ with every other privacy controller. To reduce the number of PRF evaluations in the online phase for a privacy controller (i.e., reduce the number of edges in the graph G), we propose an optimization that leverages the fact that the protocol is repeated over a long period of time with similar participants, i.e., the long-running nature of streaming queries.

Online Phase Optimization. We reduce the communication overhead during the online phase by choosing privacy controller's nonce as to only include a small random subset of the pairwise-secrets in each round. In graph terms, this corresponds to a small expected degree of each vertex. As long as the graph remains connected[2], confidentiality is guaranteed. We divide the online phase into epochs consisting of t rounds. At the beginning of each epoch, we use $N-1$ evaluations of the PRF to bootstrap the secure aggregation graphs for the epoch. A privacy controller assigns each edge to a small number of rounds, based on the output of a PRF evaluated on the shared secrets. More specifically, we divide the output of $PRF(k_{p,q}, r)$, where r is a public epoch-identifier, into b-bit segments. Each segment assigns the edge $e_{p,q}$ to one of 2^b graphs using the number encoded in the b-bit segment.

Assuming a 128-bit output size of a PRF (e.g., AES), an epoch consists of $t = \lfloor 128/b \rfloor \cdot 2^b$ rounds. In comparison, the protocol of Ács et al. [16] uses the same $N-1$ PRF evaluations to create only a single secure aggregation graph (i.e., epoch size of one). Ideally, we want to create as many graphs as possible, i.e., select a large b, since with increasing b, an epoch consists of more rounds. However, with increasing b, each of the associated graphs has fewer edges, which increases the risk of a graph being disconnected. In the extended version of this paper [34], we show how to select b so that the probability of any honest subset of nodes being isolated in any of the t generated graphs is bounded by δ, assuming a fraction of at most α parties collude.

For example, for 10k privacy controllers, assuming that up to half are colluding ($\alpha = 0.5$), and bounding the failure probability by $\delta = 1 \times 10^{-9}$, allows for $b = 7$, which results in an epoch consisting of 2304 rounds where each vertex has a expected degree of 78. As a consequence, our optimization requires 190k PRF evaluations and 180k additions for constructing all 2304 blinding nonces of an epoch. In comparison, the basic protocol requires 23 million PRF evaluations and additions while the protocol from Ács et al. [16] requires 23.2 million PRF evaluations and 180k additions[3].

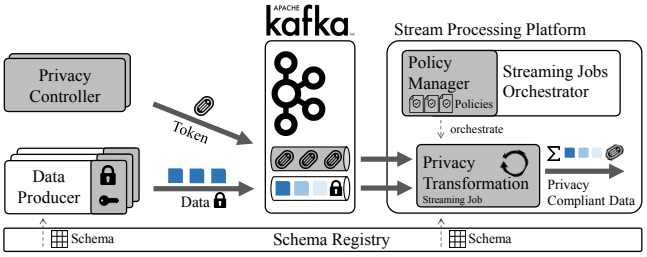


Figure 2: Overview of Zeph's architecture and integration into existing data streaming pipelines. Zeph's components are highlighted in gray.

4 Zeph System Design

Zeph is a privacy platform that cryptographically enforces user-defined privacy preferences in streaming platforms by sharing only transformed privacy-compliant views of the underlying encrypted data. So far, we have described the cryptographic building blocks that enable privacy transformations in Zeph. Here, we describe how we overcome the system challenges that need to be addressed to allow practical deployment.

[2] More specifically, the subgraph of *honest* nodes must remain connected.

[3] All results assume that τ_p is at most 128-bit long and hence a single evaluation of AES is sufficient for encryption.

Zeph augments existing stream processing pipelines, similar to existing frameworks operating on data in-the-clear [64]: *(i)* On data producers, Zeph adds a proxy module for encoding and encryption. *(ii)* On the server, Zeph adds a microservice running in the existing stream processing platform. This microservice transforms the incoming encrypted streams into privacy-compliant output streams (Figure 2), which can then be consumed by existing stream processing queries for arbitrary post-processing.

4.1 User API and Privacy Policies

Before introducing the Zeph components in detail, we discuss aspects related to users' interaction with Zeph.

Privacy Preferences. Zeph provides the capabilities for users to set their privacy preferences (i.e., user-centric privacy) and the means to cryptographically enforce various privacy policies in a unified system. In this paper, we do not consider the question of what this set of privacy preferences should be. Nevertheless, we suggest and implement a sensible set of options to demonstrate how Zeph can be used in practice. In the current design, data owners can set their preferences as follows: *(i)* do not share my data, *(ii)* share my data without restrictions, *(iii)* share my data only when aggregated with other users, and *(iv)* share only generalized views of my data and/or mask sensitive data, i.e., share but limit inference of sensitive information from my data. The realization of these preferences in practice is application- and data-dependent (i.e., generalization and data minimization techniques can differ depending on the data type, e.g., image, location, heart rate).

Data Stream Schema. In Zeph, developers can translate user preferences to an application-specific set of transformations by mapping them in a schema language. Zeph's schema language builds on the *Avro* [20] schema language (Figure 3). Using our extended schema language, developers can translate users' privacy options to configurations, encodings, and transformations for their application. In addition, the schema contains meta-information about the stream and the contents of events within a stream. This enables seamless integration into existing streaming services employing schema registries to store structural information about the events flowing through the system. A Zeph stream schema contains: *(i) Metadata attributes* describing static fields that remain constant for an extended period of time and are public information. Zeph's microservice uses these metadata tags to group and filter streams for transformations over different populations (§4.3). For example, the region where a data stream originates from (Figure 3). *(ii) Stream attributes* describe the private contents of an event message and are annotated with all possible supported queries. These explicit annotations are required to derive the necessary encodings to execute queries using the three core functions (§3). For example, a heart rate sensor might have two stream attributes such as heart-rate and heart-rate variability (Figure 3). The heart-rate is annotated to support aggregates with variance statistics. *(iii)* The *privacy options* for stream attributes. A privacy option describes the set of transformations that the service can perform to reveal an output. The options *stream-aggregate* (Σ_S), *aggregate* (Σ_M), and *dp-aggregate* (Σ_{DP}) directly correspond to the three core functions defined in §3. In addition, *private* does not allow any transformations on the stream while *public* allows access to the raw data. For each transformation set, one can add further constraints, e.g., defining a minimum population size, specifying a lower temporal resolution by aggregating over time or providing a privacy budget for the transformation.

```
name: MedicalSensor
metadataAttributes:
  - name: ageGroup
    type: [enum, optional]
    symbols: [young, middle-
              aged, senior]
  - name: region
    type: string
streamAttributes:
  - name: heart-rate
    type: integer
    aggregations: [var]
  - name: hrv
    type: integer
streamPolicyOptions:
  - name: aggr
    option: aggregate
    clients: [medium, large]
    window: [1hr]
  - name: priv
    option: private
```

```
id: 235632224234
ownerID: 2474b75564b
serviceID: app.com
validFrom: 2020-04-20
validTo: 2021-04-20
stream:
  type: MedicalSensor
  metadataAttributes:
    ageGroup: middle-aged
    region : California
  privacyPolicy:
  - heartrate:
      option: aggr
      clients: medium
      window: 1hr
  - hrv:
      option: priv
```

Figure 3: An example privacy policy schema of a medical sensor (*left*) and a stream annotation for this schema (*right*). (YAML format for display)

Annotating Streams. The Zeph schema for a particular application can be accessed by all privacy controllers. A user's privacy selection in the application triggers the responsible privacy controller to create a matching stream annotation and share it with the server. A stream annotation contains the selected privacy option along with values of the metadata attributes and additional information about the stream (Figure 3). This information later allows Zeph's server to identify suitable streams to include in privacy transformations. Stream annotations contain an identifier of the data owner (e.g., the hash of their public key) that maps to the data owner's public key in the PKI.

4.2 Writing Encrypted Data Streams

Data producers submit streams of events to the pipeline where each event conforms to a data schema in the schema registry, as in standard streaming pipelines. However, Zeph augments data producers with a proxy module to handle encoding and encryption.

Setup. To initialize a new data stream matching a Zeph schema, the data producer generates a master secret and shares both the schema and the master secret with the associated privacy controller. After the initial setup phase, the data producer can start sending encrypted data to the server without any further coordination with the privacy controller.

Encrypting Data Streams. The proxy module encrypts each record with a symmetric homomorphic encryption scheme (§3), using the master secret from the setup phase. In order to allow the privacy controller to derive a transformation token without observing the data (§3), the data producer sends a neutral value at regular intervals (e.g., every minute) to terminate the window. This does not affect the result of computations but is required for efficient Σ_S transformations across time. Additionally, these messages allow Zeph's microservice to detect and handle dropout of data producers (e.g., due to network interruptions).

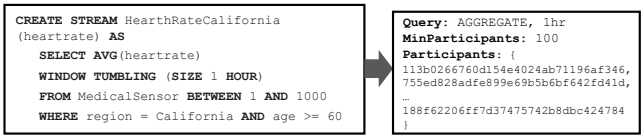
4.3 Matching Queries with Privacy Policies

Zeph's microservice contains a policy manager that maintains a global view of the system and coordinates active streams, privacy controllers, and transformations in the streaming pipeline. It provides a query interface for launching new transformations and matches queries with available streams by considering their chosen privacy options. Privacy transformations are constructed from chains of the core operations (§3.2) and are executed as stream processing queries running continuously on a set of encrypted streams.

Zeph's policy manager includes a query planner that leverages the fact that privacy transformation queries follow the same structure, which we discuss in more detail below. The policy manager needs to ensure that queries comply with all the stream's selected policy options. Otherwise, it will not receive the required transformation tokens from the privacy controllers.

Query Language. The query language of Zeph builds on *ksql* [51], an SQL-like query language for expressing continuous queries on data streams. Any authorized service can express privacy transformations that follow the pattern explained above. Figure 4 shows an example query, which creates a transformed stream for the hourly average heart rate of seniors in California, including at most 1k streams.

Query Planner. The query planner executes queries from authorized services in three steps: *(i)* streams are filtered by their metadata attributes (e.g., all medical sensor streams in California). *(ii)* an Σ_S operation using a time-window is performed on certain attributes of each selected stream (e.g., average heart rate over 1 hour). The query planner checks for each selected stream that the transformation complies with the annotated privacy options for the attributes used, else the stream is excluded. *(iii)* If more than one stream is selected, an Σ_M or Σ_{DP} operation is performed on the results of the previous step. The query planner checks for each remaining stream that the transformation complies with the privacy option and checks that the population constraints are met (e.g., minimum population size), or otherwise excludes the stream. These compliance checks are necessary, as privacy controllers would not provide the required tokens for a stream where the privacy options do not allow the query. To prevent an attacker from combining outputs of different transformations to violate privacy policies, any stream attribute can be matched to only one transformation, and is removed from the set of queriable streams for this attribute as long as the stream is part of the running transformation. The privacy controller generally only supplies a single transformation token for each window in a given stream, preventing differencing or re-use attacks. For DP aggregations, a stream value can contribute to multiple transformations if allowed by its current privacy budget. The privacy controller maintains the privacy budget and suppresses transformation tokens if the privacy budget is used up. After processing the query, the query planner outputs a *transformation plan* that encodes the list of streams in the transformation, fault tolerance details (i.e., number of participants dropout the system can handle), and the sequence of operations the system need to perform (Figure 4).



privacy transformation *transformation plan*

Figure 4: The query planner converts privacy transformations into transformation plans with complying data streams.

4.4 Coordinating Privacy Transformations

Once the query planner outputs a transformation plan, Zeph executes the privacy transformation in the streaming pipeline. Zeph provides a customized stream processor that handles the required coordination between the transformation job running in the streaming pipeline and the privacy controllers. In addition to handling data, it consumes event messages (i.e., tokens) from privacy controllers and writes events about the state of the transformation back to the privacy controllers.

Transformation Setup. Zeph introduces a coordinator component that initiates the setup based on transformation plans provided by the query planner. In order to initialize a new job, the coordinator first determines the involved privacy controllers and distributes the transformation plan to them. This step enables the privacy controllers to verify the compliance of the transformation against the user-defined privacy option. The verification involves checking the privacy policy based on the included attributes, window size, aggregation size, and/or noise configurations. If the transformation plan includes multiple data owners, each privacy controller needs

to verify the identities involved in the transformation plan by fetching their certificates from the PKI. Afterwards, each privacy controller initiates the setup phase of the secure aggregation protocol (§3.4) among the involved privacy controllers. Once all privacy controllers agree, the coordinator initiates the transformation job in the streaming pipeline.

Transformation Execution. The stream processor continuously aggregates incoming encrypted events into windows and applies the transformation tokens received from the privacy controllers. Zeph runs an interactive protocol with the privacy controllers once per window, to robustly adjust to failures of both data producers and privacy controllers. At the end of each window, the stream requests a heartbeat from all privacy controllers in the transformation. Note that data producer dropouts can be detected by the absence of their events. After a specified timeout, the data transformer computes the intersection of available data producers and privacy controllers and broadcasts a membership delta in comparison to the previous window to all involved privacy controllers.

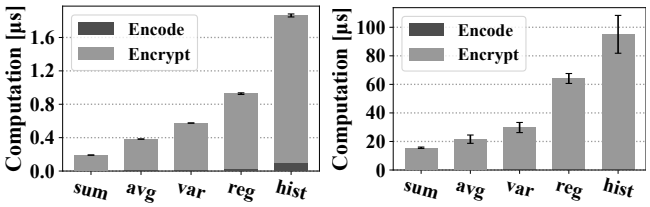
After receiving an update, the privacy controllers verify that the transformation still complies with the selected privacy options and update the tokens they send to match the new transformation. Upon the arrival of all transformation tokens, Zeph can complete the transformation and output the result.

5 Implementation

Our prototype of Zeph is implemented on top of Apache *Kafka* [21], consisting of roughly 4500 SLOC for Zeph and and additional 5500 SLOC for benchmarks. We provide a data producer proxy library written in Java that relies on the Bouncy Castle library [58] for cryptographic operations, and *Avro* [20] for serialization. The privacy controller is implemented in Java but, via the Java native interface (JNI), calls native code in Rust for the secure aggregation protocol. For the PRF, we rely on CPU-based AES-NI using the `AES` Rust crate [74], and for the ECDH key exchanges we use the `secp256r1` elliptic curve from Bouncy Castle [58]. We use the Apache *Kafka* Streams [22] to implement the stream processor for the privacy transformations. We emulate the policy manager with a configurable Ansible [19] playbook.

6 Evaluation

Meeting the performance requirements of data stream processing is a key goal of Zeph's design. Therefore our experimental evaluation is designed to validate this and more concretely answer the following two questions: *(i)* what is the cost of enforcing privacy policies with encryption in Zeph?, and *(ii)* can Zeph provide the means to support practical privacy for various applications in an acceptable overhead?



(a) EC2 instance (b) Raspberry Pi

Figure 5: The computation cost at the data producer for encryption and different stream encodings: sum, average, variance, linear regression, histogramm. The encoding for the histogramm has ten buckets.

6.1 Experimental Setup

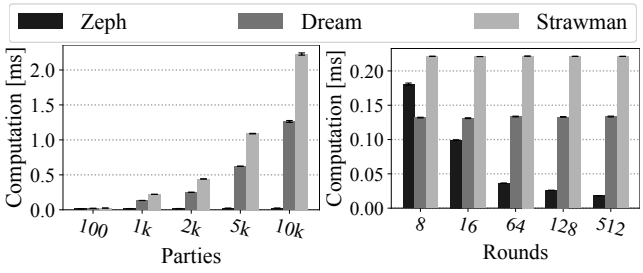
The experimental evaluation consists of two parts. First, we quantify the overhead of Zeph components with microbenchmarks. We start by quantifying the performance of our proposed secure aggregation optimization compared to a *Strawman* with no optimizations (§3.4) and the optimized protocol by Ács et al. *Dream* [16]. The second part of the evaluation aims to quantify the end-to-end performance of Zeph as we integrate it into three applications with different privacy options. Moreover, we show how various data-dependent privacy logic can be realized in Zeph. In these experiments, we consider a setting where each data producer has a separate privacy controller; this represents the worst-case scenario – the number of privacy controller involved in the MPC protocol is equal to the number of data streams.

Compute. We run the microbenchmarks on Amazon EC2 machines (m5.xlarge, 4 vCPU, 16 GiB, Ubuntu Server 18.04 LTS). Additionally, we run the data producer microbenchmarks on a Raspberry Pi 3 B to analyze the performance on more resource-constrained edge devices. For the end-to-end evaluation, we employ Amazon MSK [18], which provides a *Kafka* cluster as a fully managed service. The *Kafka* cluster contains two broker nodes (m5.xlarge) spread over two availability zones in Frankfurt. The stream processor application spreads over a set of two EC2 machines (m5.2xlarge) using *Kafka* streams. Data producers and privacy controllers are grouped into partitions of up to 100 entities. A single producer- or controller-partition runs on one EC2 machine (m5.large). We run the partitions in three different regions London, Paris, and Stockholm, to simulate federation.

Configuration. In the microbenchmarks, Zeph uses an event with a single stream attribute x encoded as $\vec{x} = [x, x^2, 1]$ while for the end-to-end setup, we use application-specific encodings. Throughout the evaluation, Zeph's optimized secure aggregation assumes that up to half the participants are colluding (i.e., $\alpha = 0.5$), and that the failure probability is below $\delta = 1e-7$. For the end-to-end evaluation the data producer uses a Poisson process with a mean of 0.5 to time inserts (i.e., an average of 2 inserts/s).

Privacy Controllers	100	1k	10k	100k
Bandwidth	9.0 KB	91 KB	910 KB	9.1 MB
Bandwidth Total	901 KB	91 MB	9.1 GB	910 GB
Shared Keys	3.2 KB	32 KB	0.3 MB	3.2 MB
ECDH	25 ms	249 ms	2.5 sec	25 sec
ECDH Total	2.5 sec	4 min	7 h	693 h

Table 2: The computation and bandwidth costs for the privacy controller in the *setup phase* of a multi-stream transformation. The total amount consists of the sum of all cost involved over all privacy controllers of the transformation, versus the costs for a single privacy controller. The Elliptic-curve Diffie–Hellman (ECDH) key exchange dominates the computation and bandwidth costs.



(a) Average per round (b) Varied rounds for 1k parties

Figure 6: Computation costs for privacy controllers in the *privacy transformation phase* to execute multi-stream queries. A round corresponds to a transformation of a single time window.

6.2 Data Producer

We now discuss Zeph's overhead at the data producers.

Computation. The encryption cost for a single record with *Enc* is 0.19μs on EC2 and 16μs on a Raspberry Pi, the cost is low because the encryption scheme relies on symmetric primitives (i.e., efficient AES). Figure 5 shows the encryption latency for different encodings. A data producer can encrypt events at a rate in the range of 5.3m to 524k records per second (rps), depending on the encoding. Even on a Raspberry Pi, the computation cost is moderate, and a throughput of 7.7k to 76.6k rps can be observed. To accommodate for window borders, the data producer has to additionally submit a ciphertext per-window, which increases the cost at a fixed rate.

Bandwidth. Compared to plaintext, Zeph's aggregation-based encodings and timestamp introduce a ciphertext expansion which manifests itself in increased bandwidth requirements. The expansion varies from 24 bytes (1.5x) with one encoding to 96 bytes (6x) with 10 encodings, i.e., grows by 8 bytes per encoding. Besides this, the window border ciphertexts increase bandwidth with an additional constant factor.

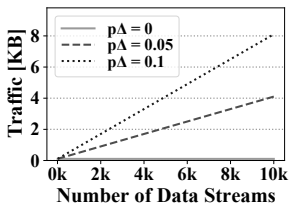
6.3 Privacy Controllers

The cost of the privacy controller depends on the executed transformations on the service side. Single-stream window transformations are efficient both in computation and bandwidth because no MPC is involved. The privacy controller computes the transformation tokens on a per-window basis from the master secret with a computation cost of around 0.2μs and bandwidth cost of 8 bytes per token.

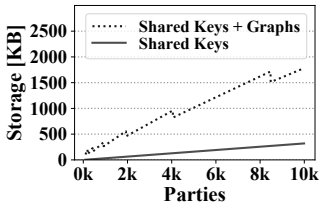
For the multi-stream case, the privacy controller engages in the secure aggregation protocol (§3.4). We quantify the overhead by running the secure aggregation protocol for different numbers of privacy controllers and compare it against the strawman approach. As a first step, all these protocols require a *setup phase* to establish pairwise shared secrets with all involved parties. Afterward, the *privacy transformation phase* starts, during which the privacy controllers create the required transformation tokens at the end of each window.

Setup Phase. The *setup phase* overhead increases quadratically with the number of privacy controllers, i.e., $O(N^2)$. However, we assume that realistic deployments will feature at most a few thousand privacy controllers in a single aggregation. Beyond this point, further scalability should be realized through hierarchical transformations. In our evaluation, we explore aggregations with up to 10k privacy controllers, which is the current limit of feasibility without resorting to hierarchies. Table 2 shows a quadratic increase of the bandwidth and computation costs for running the setup phase with the ECDH key exchanges. However, the overall amount is reasonable even for 10k participants, setting with 910 KB bandwidth and 2.5 sec computation cost per privacy controller. Note that the setup phase has to be performed only when a new transformation query is created. In terms of memory, the privacy controllers need to store their private-key (i.e., 150 bytes) and the established shared secrets of the current privacy transformation. Each shared key requires 32 bytes, e.g., 3.2MB for 100k shared keys.

Privacy Transformation Phase (Optimization). Zeph optimizes the cost of the secure aggregation protocol per round by computing the shares in random sub-groups (§3.4). In the initial phase, the controllers have to invest more resources to compute the random subgroup for the upcoming rounds (i.e., epoch). After a few rounds, the additional work performed at the beginning of an epoch is amortized and, therefore, the overall cost of the computation reduces significantly in the long run, as depicted in Figure 6. With 1k participants, the computation costs for the first window is 1 ms for a privacy controller, while in the following windows Zeph reduces the computation cost by 2.6x. Already for 8 and 16 windows for 10k and 1k participants, respectively, the Zeph optimization is more efficient on average and the amortized performance improvement increase linearly with the number of rounds the transformation runs, as shown in Figure 6a. For 10k privacy controllers, an individual participant requires less than 2 MB



(a) Bandwidth for transformation phase depending on delta probability p_Δ.



(b) Memory costs for a privacy controller during the privacy transformation phase.

Figure 7: Bandwidth and memory costs for privacy controllers in the *privacy transformation phase*.

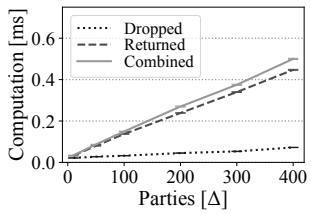


Figure 8: Computation cost for a privacy controller to adapt to Δ dropping or joining parties. In the combined case, Δ members dropped out and Δ other members returned.

to store the shared keys and the secure aggregation graphs of the epoch (Figure 7b). As a result, even though the overhead increases in the number of privacy controllers, the total memory remains acceptable. In case memory is scarce (e.g., because a privacy controller is in charge of large number of data streams), a privacy controller can resort to storing a fraction of the secure aggregation graphs and recalculate the next batch of graphs at the required time.

Dropout. In Zeph, privacy controllers can dynamically join or leave in the transformation phase, which increases both the computational cost and the required bandwidth due to the additional communication, as depicted in Figure 8. The computation and bandwidth costs for adapting the transformation token are linear in the number of returning participants as well as dropout participants. These costs are modest, even for the extreme fraction of dropping and joining users (i.e., 400 each), the induced cost remains below 0.5 ms. In terms of bandwidth, a privacy controller observes less than 10KB bandwidth, even under the assumption of a 10% fluctuation of dropout participants (Figure 7a).

6.4 End-to-End Application Scenarios

This section evaluates the end-to-end overhead of Zeph and its effectiveness in supporting a variety of privacy policies relevant to real-world applications. We develop three applications with Zeph that represent different complexities of privacy transformations. We evaluate each application with 300 and 1200 active data producers, each producing two events per second with a window size of 10 seconds. Each data producer has its own privacy controller and we set $\alpha = 0.5$ as usual.

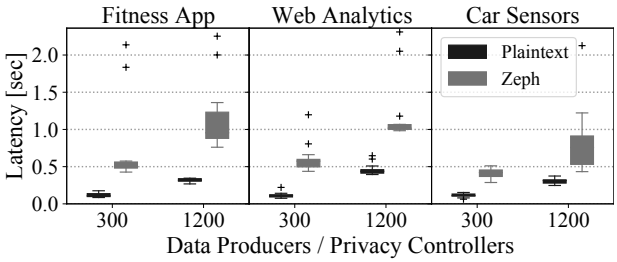


Figure 9: Computation cost for Plaintext (no encryption) and Zeph for different Applications. The latency measures the time after the grace period (5s) of a window is over until the result of the transformation is available.

Fitness Application. We consider the Polar App [10] which collects data during users' sports activities. Recorded data includes heart-rate, altitude and weather information, among others. We consider a privacy policy that limits the resolutions of sensor data temporally and/or spatially. In our evaluation, we gather statistics about the average heart-rate of a population organized into per-altitude buckets with a maximum resolution of 5 meters. Each exercise event consists of 18 attributes that are encoded in 683 values in Zeph.

Web Analytics. We implement Zeph on a subset of statistics from the Matamo [8] web analytics platform for gathering website statistics such as page views, user flows, and click maps. Here we evaluate aggregation queries using a privacy policy that translates to only differentially private (i.e., noised) information aggregated over all users being made available to a third-party service. To enable this functionality in Zeph, we encode the 24 attributes into 956 values.

Car Predictive Maintenance. We consider a car metric data platform that contains a predictive maintenance service [1]. We consider a setting where users allow a third-party service to observe sensor readings only if they are out of the ordinary or differ too much from long-term aggregates across different cars. Therefore we compute both the long-term aggregates across many users and individual histograms for each user. The application records 23 different attributes from car sensors and encodes them into 169 values.

Performance. Figure 9 shows the observed stream transformation latencies for the different applications compared against plaintext. The latency overhead varies between 2x and 5x for the different applications. Zeph completes processing the current window before the next one needs to be processed. With this, we show that Zeph is capable of performing real-time privacy transformations atop encrypted streams for a variety of application scenarios.

7 Related Work

Privacy Policy Enforcement. Enforcing privacy policies automatically in real-world data processing systems is often

achieved by resorting to Information Flow Control (IFC) to check and constrain how information flows through the system [26, 43, 44, 61, 76, 82]. These systems feature different variations on how IFC rules can be expressed and who enforces these rules in application code. In contrast to Zeph, these approaches rely on a trusted service or trusted hardware for privacy enforcement. Riverbed [82] is a practical IFC system that enforces user-defined privacy policies with information flow techniques by grouping users with similar policies into separately running containers (i.e., universes). Ancile [23] introduces a trusted data processing library that automatically enforces user-defined privacy preferences on passively generated data by only releasing policy complying transformations of data to applications. In a *multiverse database* [60, 75], global privacy policies are enforced by only exposing materialized views of the database to each user in an application. A multiverse database is fully trusted to enforce privacy policies correctly. Qapla [62] allows a policy compliance team to associate a set of policies to database schemas, which a trusted reference monitor then enforces.

Private Aggregate Statistics. Secure aggregation protocols have been used in a variety of private system designs, largely to enable services to collect statistics over users' data without accessing individual data [16, 28, 36, 39, 56, 63, 73, 77]. Compared to Zeph, these systems require data producers to actively participate in the aggregation protocol, keep data local, and do not support a wide range of privacy transformations. While we utilize a secure aggregation protocol [16, 28] to construct privacy transformation tokens that require inputs from multiple trust domains, this does not impact data producers in Zeph design. Several systems [16, 63, 73, 77] combine differential privacy techniques [40, 42] (i.e., by adding noise to inputs) with secure aggregation in a way that minimizes the amount of added noise. This line of work is orthogonal to this work, and some can be integrated with Zeph.

Private Outsourced Computation. A different line of work investigates how to protect the confidentiality of data while allowing a server to compute on encrypted data either with homomorphic encryption [33, 69, 70, 80] or secret sharing [52, 53]. This line of work is orthogonal to Zeph, and the goal of Zeph is to augment these systems with the capability to selectively release encrypted data following an evaluation of a privacy transformation. Encrypted processing systems can be adapted to perform privacy transformations but then require clients first to decrypt the outputs. Zeph supports both direct release of privacy-compliant views of data and privacy transformations to a targeted authorized party.

Functional Encryption. Another closely related line of work is functional encryption [29, 30, 46] (FE). Functional encryption allows a data owner to issue restricted secret keys that enable the key holder to learn only the output of a specific function. Existing constructions are currently not yet efficient enough for practical systems. Additionally, some of the privacy transformations in Zeph require functions on multiple inputs from multiple trust domains, which requires techniques that are even more complex than standard FE [65].

8 Conclusion

The practice of massive data collection is not likely to diminish anytime soon. Corporations across all sectors consider data as a valuable asset that has enormous value to their business. However, as we accumulate more and more sensitive data, protecting individuals' privacy is gaining critical urgency. Today's privacy landscape presents a unique set of challenges and opportunities that make this an auspicious time to reshape our data ecosystems for privacy. Adequately addressing privacy in the current complex computing landscape is an acute challenge and is vital to avoid the pitfalls of big data. The path for achieving this necessitates developing privacy tools that can easily be implemented in existing data pipelines. In this paper, we propose a new end-to-end design for privacy. A design that empowers users with more control with a user-centric model to privacy and that ensures strong data protection and compliance assurance with a cryptographic enforcement approach to privacy policies.

Acknowledgments

We thank our shepherd Amit Levy, the anonymous reviewers, Hidde Lycklama, and Emanuel Opel for their valuable feedback. This work was supported in part by the SNSF Ambizione Grant No. 186050 and an ETH Grant.

References

[1] Bosch Predictive Maintenance. Online: https://www.bosch-mobility-solutions.com/en/products-and-services/mobility-services/predictive-diagnostics/. Accessed: 09-12-2020.

[2] California Consumer Privacy Act (CCPA). CCPA, Online: https://oag.ca.gov/privacy/ccpa. Accessed: 09-12-2020.

[3] General Data Protection Regulation: GDPR. GDPR, Online: https://gdpr-info.eu/. Accessed: 09-12-2020.

[4] Google Cloud De-identification. Online: https://cloud.google.com/dlp/docs/classification-redaction. Accessed: 09-12-2020.

[5] Immuta Platform. Online: https://www.immuta.com/. Accessed: 09-12-2020.

[6] Instagram Data Policy. Online: https://help.instagram.com/519522125107875. Accessed: 09-12-2020.

[7] IRI Total Data Management Redaction. Online: https://www.iri.com/blog/data-protection/redaction-options-for-data-privacy/. Accessed: 09-12-2020.

[8] Matomo Web Analytics. Online: https://matomo.org/. Accessed: 09-12-2020.

[9] Oracle Responsys Data Redaction. Online: https://docs.oracle.com/en/cloud/saas/marketing/responsys-user/DataRedaction.htm. Accessed: 09-12-2020.

[10] Polar Platform. Online: https://www.polar.com/accesslink-api/#detailed-sport-info-values-in-exercise-entity. Accessed: 09-12-2020.

[11] Privitar Privacy Platform. Online: https://www.privitar.com/. Accessed: 09-12-2020.

[12] Pseudonymisation techniques and best practices. Online: https://www.enisa.europa.eu/publications/pseudonymisation-techniques-and-best-practices/. Accessed: 09-12-2020.

[13] Twitter Privacy Policy. Online: https://twitter.com/en/privacy. Accessed: 09-12-2020.

[14] Gartner Says Just Four in 10 Privacy Executives Are Confident About Adapting to New Regulations. Gartner, Online: https://www.gartner.com/en/newsroom/press-releases/2019-04-23-gartner-says-just-four-in-10-privacy-executives-are-confident-about-adapting-to-new-regulations, April 2019.

[15] John M. Abowd. The U.S. Census Bureau Adopts Differential Privacy. In *ACM SIGKDD*, 2018.

[16] Gergely Ács and Claude Castelluccia. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *International Workshop on Information Hiding*. Springer, 2011.

[17] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *VLDB*, 6(11):1033–1044, 2013.

[18] Amazone MSK. Online: https://aws.amazon.com/de/msk/. Accessed: 09-12-2020.

[19] Ansible. Online: https://www.ansible.com/. Accessed: 09-12-2020.

[20] Apache Avro. Online: https://avro.apache.org/. Accessed: 09-12-2020.

[21] Apache Kafka. Online: https://kafka.apache.org/. Accessed: 09-12-2020.

[22] Apache Kafka Streams. Online: https://kafka.apache.org/documentation/streams/. Accessed: 09-12-2020.

[23] Eugene Bagdasaryan, Griffin Berlstein, Jason Waterman, Eleanor Birrell, Nate Foster, Fred B. Schneider, and Deborah Estrin. Ancile: Enhancing Privacy for Ubiquitous Computing with Use-Based Privacy. In *ACM WPES*, 2019.

[24] E. Balsa, C. Troncoso, and C. Diaz. OB-PWS: Obfuscation-Based Private Web Search. In *IEEE Symposium on Security and Privacy*, 2012.

[25] John Biggs. It's time to build our own Equifax with blackjack and crypto. Online. http://tcrn.ch/2wNCgXu, September 2017.

[26] Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B. Schneider. SGX Enforcement of Use-Based Privacy. In *ACM WPES*, 2018.

[27] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences*, 117(21):11608–11613, 2020.

[28] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM CCS*, 2017.

[29] Dan Boneh, Amit Sahai, and Brent Waters. Functional Encryption: Definitions and Challenges. In *TCC*. Springer, 2011.

[30] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC*. Springer, 2014.

[31] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: optimizations and applications. In *ACM CCS*, 2017.

[32] Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. Foundations of homomorphic secret sharing. In *ITCS*, 2018.

[33] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. In *USENIX NSDI*, 2020.

[34] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. [Extended Version] Zeph: Cryptographic Enforcement of End-to-End Data Privacy. In *arXiv*, 2021.

[35] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering*, 36(4), 2015.

[36] C. Castelluccia, E. Mykletun, and G. Tsudik. Efficient Aggregation of Encrypted Data in Wireless Sensor Networks. In *ACM MobiQuitous*, July 2005.

[37] Claude Castelluccia, Aldar CF Chan, Einar Mykletun, and Gene Tsudik. Efficient and Provably Secure Aggregation of Encrypted Data in Wireless Sensor Networks. *ACM TOSN*, 2009.

[38] Long Cheng, Fang Liu, and Danfeng Daphne Yao. Enterprise Data Breach: Causes, Challenges, Prevention, and future Directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5), 2017.

[39] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *USENIX NSDI*, 2017.

[40] Cynthia Dwork. Differential Privacy. In *ICALP*, Lecture Notes in Computer Science, 2006.

[41] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*. Springer, 2006.

[42] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.*, 2014.

[43] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *USENIX Security*, 2016.

[44] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI*, 2010.

[45] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *ACM CCS*, 2014.

[46] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *IEEE FOCS*, 2013.

[47] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *USENIX OSDI*, 2018.

[48] Eloise Gratton. Beyond Consent-based Privacy Protection. Online: https://www.eloisegratton.com/files/sites/4/2016/07/Gratton_Beyond-Consent-based-Privacy-Protection_-July2016.pdf, July 2016.

[49] Stephanie Hare. These new rules were meant to protect our privacy. They don?t work. The Guardian, Online: https://www.theguardian.com/commentisfree/2019/nov/10/these-new-rules-were-meant-to-protect-our-privacy-they-dont-work, November 2019.

[50] Amnesty International. The google-fitbit merger must include human rights risks. Online. https://www.amnesty.eu/wp-content/uploads/2020/11/Google-Fitbit-merger-complaint-to-the-EU-Commission-FINAL.pdf, November 2020.

[51] Hojjat Jafarpour and Rohan Desai. KSQL: Streaming SQL Engine for Apache Kafka. In *EDBT*, 2019.

[52] Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A Framework for Outsourcing of Secure Computation. In *ACM CCSW*, 2014.

[53] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing Multi-Party Computation. *IACR Cryptol. ePrint Arch. Report 2011/272*, 2011.

[54] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics*, 14(2):263–279, 2005.

[55] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *ACM SIGMOD*, 2015.

[56] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-Friendly Aggregation for the Smart-Grid. In *PoPETS*, 2011.

[57] Crystal Lee and Jonathan Zong. Consent Is Not an Ethical Rubber Stamp. Online: https://slate.com/technology/2019/08/consent-facial-recognition-data-privacy-technology.html, August 2019.

[58] Java BouncyCastle Cryptograpy Library. Online: https://www.bouncycastle.org/. Accessed: 28-04-2020.

[59] Kevin Litman-Navarro. We Read 150 Privacy Policies. They Were an Incomprehensible Disaster. nytimes, Online: https://www.nytimes.com/interactive/2019/06/12/opinion/facebook-google-privacy-policies.html, June 2019.

[60] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. Towards Multiverse Databases. In *ACM HotOS*, 2019.

[61] Miti Mazmudar and Ian Goldberg. Mitigator: Privacy Policy Compliance using Trusted Hardware. In *PoPETS*, 2020.

[62] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy Compliance for Database-Backed Systems. In *USENIX Security*, 2017.

[63] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *NDSS*, 2016.

[64] David Millman. Blog: Data Privacy, Security, and Compliance for Apache Kafka. Online: https://www.confluent.io/blog/kafka-data-privacy-security-and-compliance/. Accessed: 09-12-2020.

[65] Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl Gunter. Controlled Functional Encryption. In *ACM CCS*, 2014.

[66] Lily Hay Newman. The Alleged Capital One Hacker Didn't Cover Her Tracks. WIRED, Online: https://www.wired.com/story/capital-one-hack-credit-card-application-data/, July 2019.

[67] Cristina Onose. 10 privacy issues and trends. Online: https://www.pwc.com/ca/en/services/consulting/privacy/privacy-canadian-business-hub/2020-and-beyond-10-privacy-issues-and-trends-part-1.html, January 2020.

[68] Oracle. Innovation in Retail: Using Machine Learning to Optimize Retail Performance. Online: http://www.oracle.com/us/industries/retail/data-analytics-retail-perform-info-4124126.pdf. Accessed: 09-12-2020.

[69] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX OSDI*, 2016.

[70] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP*, 2011.

[71] PYMNTS. Amazon to pay consumers for their shopping data. https://www.pymnts.com/amazon/2020/amazon-to-pay-consumers-for-their-shopping-data/, 21 October 2020.

[72] J. L. Raisaro, J. R. Troncoso-Pastoriza, M. Misbach, J. S. Sousa, S. Pradervand, E. Missiaglia, O. Michielin, B. Ford, and J. P. Hubaux. MedCo: Enabling Secure and Privacy-Preserving Exploration of Distributed Clinical and Genomic Data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(4):1328–1341, 2019.

[73] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: Large-Scale Differentially Private Aggregation without a Trusted Core. In *ACM SOSP*, 2019.

[74] Rust AES Crate. Online: https://docs.rs/aes/0.3.2/aes/. Accessed: 09-12-2020.

[75] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. Position: GDPR Compliance by Construction. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops*, pages 39–53, 2019.

[76] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *IEEE Symposium on Security and Privacy*, 2014.

[77] Elaine Shi, Richard Chow, T-H. Hubert Chan, Dawn Song, and Eleanor Rieffel. Privacy-preserving Aggregation of Time-series Data. In *NDSS*, 2011.

[78] Latanya Sweeney. Achieving k-Anonymity Privacy Protection Using Generalization and Suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):571–588, 2002.

[79] Jeroen Tas. Going virtual to combat COVID-19. Online: https://www.philips.com/a-w/about/news/archive/blogs/innovation-matters/2020/20200403-going-virtual-to-combat-covid-19.html, April 2020.

[80] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing Analytical Queries over Encrypted Data. *VLDB*, 6(5):289–300, 2013.

[81] Iowa State University. Iowa State University scientists propose a new strategy to accelerate plant breeding by turbocharging gene banks. Online: https://www.news.iastate.edu/news/2016/10/03/sorghumgenebanks, October 2016.

[82] Frank Wang, Ronny Ko, and James Mickens. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *USENIX NSDI*, 2019.

[83] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical Private Queries on Public Data. In *USENIX NSDI*, 2017.

[84] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-Scale Secure Multiparty Computation. In *ACM CCS*, 2017.

[85] Shoshana Zuboff. *The Age of Surveillance Capitalism*. Profile Books, 2019.

DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols

Jianan Yao Runzhou Tao Ronghui Gu Jason Nieh Suman Jana Gabriel Ryan
Columbia University

Abstract

Distributed systems are notoriously hard to implement correctly due to non-determinism. Finding the inductive invariant of the distributed protocol is a critical step in verifying the correctness of distributed systems, but takes a long time to do even for simple protocols. We present DistAI, a data-driven automated system for learning inductive invariants for distributed protocols. DistAI generates data by simulating the distributed protocol at different instance sizes and recording states as samples. Based on the observation that invariants are often concise in practice, DistAI starts with small invariant formulas and enumerates all strongest possible invariants that hold for all samples. It then feeds those invariants and the desired safety properties to an SMT solver to check if the conjunction of the invariants and the safety properties is inductive. Starting with small invariant formulas and strongest possible invariants avoids large SMT queries, improving SMT solver performance. Because DistAI starts with the strongest possible invariants, if the SMT solver fails, DistAI does not need to discard failed invariants, but knows to monotonically weaken them and try again with the solver, repeating the process until it eventually succeeds. We prove that DistAI is guaranteed to find the $\exists$-free inductive invariant that proves the desired safety properties in finite time, if one exists. Our evaluation shows that DistAI successfully verifies 13 common distributed protocols automatically and outperforms alternative methods both in the number of protocols it verifies and the speed at which it does so, in some cases by more than two orders of magnitude.

1 Introduction

Distributed systems are hard to design and implement correctly. This is due to the intrinsic non-determinism from asynchronous node communications and various failure scenarios. Formal verification techniques offer a solution by proving that a distributed system is correct under all circumstances [10, 16, 32]. The verification of distributed systems consists of two components: i) proving that the desired safety properties hold for the distributed protocol itself, and ii) proving that the protocol implementation is correct.

While much work has focused on proving a system correctly implements a protocol [10, 16, 31–33], we focus on proving the protocol itself has the desired safety properties. A safety property is an invariant that should hold true at any point in a system's execution. It ensures the protocol does not reach invalid or dangerous states. For example, the safety property for a distributed lock protocol [10] is that no two nodes in the system hold a lock at the same time. The typical proof strategy is to prove that an invariant that implies the safety property is inductive, meaning that if the system starts from a state that satisfies the invariant, the invariant will still hold for any state that is reachable via a valid transition from the previous state. If the safety property itself is inductive, the proof is done. However, this is not true for almost all nontrivial distributed protocols, so that the proof requires finding an invariant that implies the safety property, then proving that it is inductive.

Finding the inductive invariant for distributed protocols is difficult, taking a long time for even simple protocols [18]. IVy [24] provides an interactive tool to make this easier. A developer provides a set of invariants and protocol specification that defines its safety property, which IVy automatically checks using an SMT solver. Each invariant can be expressed as a logical formula, which consists of a prefix with quantifiers ($\forall$ or $\exists$) and a certain number of variables, and a set of logical relations among the variables. IVy checks if adding the invariants to the safety property makes it inductive, meaning that the conjunction of all invariants with the safety property is inductive. Conjunction requires each invariant to hold, so IVy reports whether any invariant fails, at which point the developer can try again with a different set of invariants. This requires substantial manual effort by the developer.

Recent work has focused on automating invariant generation for distributed protocols, but with various limitations. I4 [18] observes that invariants for some distributed protocols do not depend on the size of the system, so I4 uses a specialized model checker to generate invariants for a small size system, then generalizes them and uses IVy to check if the conjunction of the

invariants with the safety property is inductive for the protocol specification. If not, IVy indicates which invariants failed. I4 removes them and tries again, and if that fails, tries using a larger size system to generate invariants. However, I4 provides no guarantee that it can find the inductive invariant, as it may not be possible to verify a protocol based on invariants derived from a single instantiation of the protocol. For example, if the protocol involves the parity of nodes, then no single instance can capture all behaviors of the protocol. I4 still requires manual effort, as a developer must inspect a protocol to add additional constraints that reduce the state space to make model checking feasible.

FOL-IC3 [11] infers invariants by searching for logical separators between reachable and invalid states in the protocol. It searches for separators by checking if a separator exists for a fixed number of variables and logical constraints, iteratively increasing the number of possible variables and constraints it considers if it fails. FOL-IC3 provides a strong theoretical guarantee that it can always find the inductive invariant, but does not scale to more complex protocols due to the large space of possible separators that it enumerates and its heavy and repeated use of expensive SMT queries.

We present DistAI (DISTributed protocol Automated learning of Invariants), a fully-automated system for learning inductive invariants for distributed protocols. Like I4, DistAI uses IVy to check invariants, but takes a completely different approach to generating invariants and retrying when invariants fail. We observe that even though a distributed protocol may be used in very large systems, its invariants are likely to be concise, as protocols need to be designed and understood by humans to be correct. For example, the two-phase commit protocol has an invariant that one node can commit only if all nodes have voted yes, which can be expressed as the following formula:

$$\forall N_1\, N_2.\ go\_commit(N_1) \Rightarrow vote\_yes(N_2). \qquad (1)$$

The formula for this invariant only requires two variables, N_1 and N_2, and two relations, $go\_commit(N_1)$ and $vote\_yes(N_2)$, to represent the constraint, but applies to all possible pairs of nodes in an implementation of the protocol regardless of the number of nodes in the implementation. Rather than picking a finite size system from which to generate invariants as in I4, DistAI operates in formula space and picks a finite formula size, with a maximum number of quantified variables (a variable and its quantifier $\forall$ or $\exists$) and literals (a relation such as $go\_commit(N_1)$ in the above example or its negation), for which it enumerates candidate invariants. It then combines the candidate invariants with the desired safety property and feeds them to IVy. If DistAI does not succeed for a given formula size, it increases the formula size and repeats the process until the inductive invariant is found.

Although formula space is finite, enumerating and checking all possible invariants with an SMT solver for even a modest size formula is prohibitively expensive. DistAI limits the set of candidate invariants it feeds IVy to check such that it can still provide a strong theoretical guarantee of finding the inductive invariant while delivering fast performance. DistAI provides this key feature by introducing a novel data-driven approach that uses data from protocol simulations to prune the invariants that are checked to only those that hold for the simulations.

DistAI's data-driven approach starts with the protocol specification used by IVy and automatically converts it into a form it can use to simulate the protocol for various size systems. Protocol simulation simply performs protocol actions by modifying the system state as described in the specification. This generates many raw data samples, where a sample is a snapshot of the system state after an action. Given a formula size, DistAI projects these data samples into subsamples that only involve at most the number of variables allowed by the formula size. For example, if DistAI simulates the two-phase commit protocol for a system with 100 nodes, each data sample would contain the system state for 100 nodes, but each subsample would contain the state of only two of the 100 nodes, assuming a formula size with two variables is being considered as shown in Equation (1).

Using this data, DistAI introduces a novel approach that enumerates only the strongest candidate invariants that hold for all subsamples. An invariant I is stronger than I' if I implies I'. DistAI decomposes the enumeration space of possible invariants based on the number of variables in a formula and starts enumerating smallest formulas first. Any weaker invariants already covered by an already enumerated candidate invariant are skipped. For example, if DistAI has found a candidate invariant $\forall X.p(X)$, it will not enumerate $\forall X.p(X) \vee q(X)$ since the latter is implied by the former. This approach results in fewer candidate invariants being generated, and the candidate invariants generated having smaller formula sizes, but still cover the enumeration space. Feeding these candidate invariants to IVy results in fewer and smaller SMT queries, improving performance. Currently, DistAI only enumerates universal ($\forall$ not $\exists$) invariants.

DistAI does not require sampling to be extensive or complete as the candidate invariants are checked by IVy. If adding the candidate invariants to the desired safety property is inductive, the proof is done; IVy checks if the conjunction of all candidate invariants with the desired safety property is inductive. Otherwise, IVy indicates the invariants that failed, which DistAI then refines. DistAI introduces a novel monotonic invariant refinement approach that we prove is guaranteed to find the right inductive invariant if it can be represented by a given formula size. We prove that the candidate invariants initially generated by DistAI are guaranteed to be stronger than the inductive invariant. As a result, for each candidate invariant that failed, DistAI does not need to discard it, but instead can refine it to a weaker invariant, simply by adding literals. It then tries again by feeding the updated candidate invariants back to IVy. This refinement procedure is strictly monotonic and will converge in a finite number of rounds. If the procedure still fails to find the inductive invariant, DistAI increases the formula size and repeats the whole process of sampling, enumeration, and refinement.

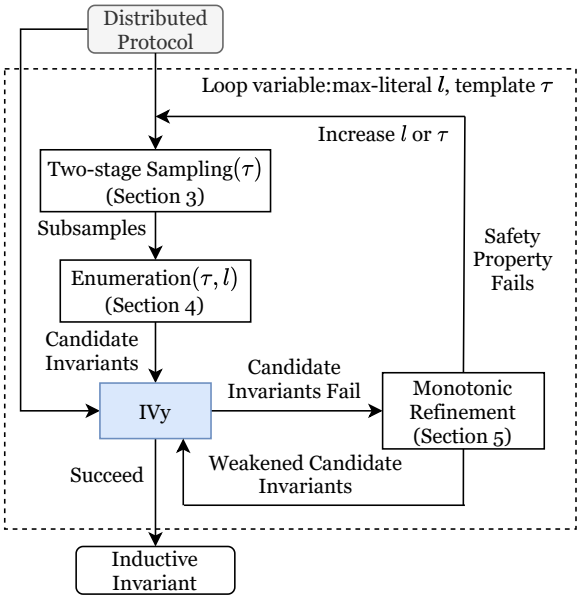


Figure 1: DistAI workflow.

We prove that if a protocol is verifiable with universal invariants, DistAI is guaranteed to verify it eventually. DistAI operates on formula space, and any invariant formula contains a finite number of variables and literals, so DistAI will converge eventually. Furthermore, DistAI is simple and self-contained, only relying on IVy. It has no other requirements for external components, such as a complex model checker. It also supports protocol abstraction, making it possible to verify protocols that use other protocols, without the need of an executable protocol implementation; only the protocol specification already required to use Ivy is needed.

We demonstrate the effectiveness of DistAI by evaluating it using 14 widely-used distributed protocols in a head-to-head comparison against I4 and FOL-IC3. DistAI outperforms I4 and FOL-IC3 in terms of both the number of protocols for which it finds the inductive invariant and the speed at which it does so. Most protocols take a few seconds and all solved protocols are proven in less than a minute. DistAI succeeds on almost 50% more protocols than either I4 or FOL-IC3. DistAI achieves these results up to an order of magnitude faster than I4 and two orders of magnitude faster than FOL-IC3, without requiring manual effort to add constraints or tune parameters.

2 Overview

Figure 1 illustrates how DistAI works. Starting with a distributed protocol specification for IVy, first, DistAI does two-stage sampling, as discussed in Section 3. It simulates the protocol on different sizes of systems, which we refer to as different size protocol instances, and records the system state as it changes as a sequence of data samples. It then projects the samples into subsamples based on the formula size currently being considered. We express formulas in prenex normal form (PNF), so that the prefix, which we refer to as *an invariant template*, has a maximum number of quantified variables and the matrix has a maximum number of literals. Second, DistAI does enumeration, as discussed in Section 4. It enumerates all strongest candidate invariants that satisfy the subsamples. Third, DistAI feeds the candidate invariants to IVy, which either succeeds with the conjunction of the invariants and the desired safety property as the inductive invariant, or fails and indicates each invariant that does not hold. Fourth, DistAI performs monotonic refinement, as discussed in Section 5. For each candidate invariant that does not hold, DistAI weakens the invariant by adding literals, then feeds the new set of candidate invariants to IVy, repeatedly weakening failed invariants until either it finds the inductive invariant or the safety property itself fails. In the latter case, DistAI increases the formula size by increasing either the maximum number of variables or the maximum number of literals allowed, and repeats the whole process of sampling, enumeration, and refinement.

We use the Ricart-Agrawala protocol [26] as an example of how DistAI works. Figure 2 shows the IVy specification of this classic distributed mutual exclusion protocol, which has five key pieces we use for learning invariants:

1. **Types.** (line 2) Types define different domains of the protocol (e.g., nodes, packets, epochs). The Ricart-Agrawala protocol only has one type, `node`.
2. **Relations.** (lines 4-6) Relations define the state of the protocol, with variables that represent types used as arguments. The Ricart-Agrawala protocol has three relations. For example, relation `holds(N)` has one variable `N` of type node, and indicates if `N` is in the critical section. If the current instance has three nodes N_1,N_2,N_3, then there are three concrete predicates $holds(N_1)$,$holds(N_2)$,$holds(N_3)$ associated with relation *holds*. Each predicate is either `true` or `false` at a certain system state.
3. **Initialization.** (lines 8-12) Initialization defines the initial state of the protocol in terms of its relations. For the Ricart-Agrawala protocol, all relations are initially false.
4. **Actions.** (lines 13-35) Actions define how the protocol may transition from one state to another, modifying the state by setting relations to `true` or `false`. Actions are defined with preconditions using the `require` keyword, which must be satisfied for the protocol to take the action.
5. **Safety Property.** (line 43) The target invariant, defined with logical constraints on the types and relations.

As shown in the specification, a node can send a request for the critical section to another node and can only enter the critical section after it has received replies from all other nodes. When receiving a request, a node delays its reply if it is currently holding the critical section, or if it has requested the critical section and already received a reply from the requester, which indicates an earlier timestamp and a higher priority. The node then sends the reply after it leaves the critical section. The safety property at line 43 asserts that at any time, there is no more than one node holding the critical section. For

```
1  #lang ivy1.7
2  type node
3
4  relation requested(N1:node, N2:node)
5  relation replied(N1:node, N2:node)
6  relation holds(N:node)
7
8  after init {
9      requested(N1, N2) := false;
10     replied(N1, N2) := false;
11     holds(N) := false;
12 }
13 action request(requester: node, responder: node) = {
14     require ~requested(requester, responder);
15     require requester ~= responder;
16     requested(requester, responder) := true;
17 }
18 action reply(requester: node, responder: node) = {
19     require ~replied(requester, responder);
20     require ~holds(responder);
21     require ~replied(responder, requester);
22     require requested(requester, responder);
23     require requester ~= responder;
24     requested(requester, responder) := false;
25     replied(requester, responder) := true;
26 }
27 action enter(requester: node) = {
28     require N ~= requester -> replied(requester, N);
29     holds(requester) := true;
30 }
31 action leave(requester: node) = {
32     require holds(requester);
33     holds(requester) := false;
34     replied(requester, N) := false;
35 }
36
37 export request
38 export reply
39 export enter
40 export leave
41
42 # safety property
43 invariant [safety] holds(N1) & holds(N2) -> N1 = N2
```

Figure 2: Ricart-Agrawala protocol written in IVy. "~" stands for negation. Capitalized variables are implicitly quantified. For example, line 9 means $\forall N_1\ N_2 \in node$, $requested(N_1,N_2) := false$.

simplicity, Figure 2 specifies the protocol without explicit timestamps and only shows one `requested` relation as opposed to separate `request_send` and `request_received` relations which would be part of the real protocol. The safety property of Ricart-Agrawala is not an inductive invariant itself. One needs to add the following two invariants to the safety property so that the resulting conjunction forms an inductive invariant:

$$\forall N_1\ N_2.\ \neg(replied(N_1,N_2) \wedge replied(N_2,N_1)) \quad (2)$$

$$\forall N_1\ N_2.\ holds(N_1) \wedge N_1 \neq N_2 \rightarrow replied(N_1,N_2). \quad (3)$$

The first invariant asserts the absence of bidirectional reply, meaning that any two nodes cannot both give the other one a higher priority. The second invariant says that any node holding the critical section must have received replies from all other nodes. DistAI automatically finds the inductive invariant by learning the additional invariants.

Two-stage sampling. To automatically learning the inductive invariant and prove the correctness of the Ricart-Agrawala protocol, DistAI first does two-stage sampling, as shown in Figure 1. It simulates the protocol at different instance sizes and records the system state as a sequence of data samples, each of which presents the values of all the relations. For example, a data sample for an instance size of five nodes (i.e., $n_1, n_2, \cdots, n_5$) using the Ricart-Agrawala protocol consists of 55 boolean values denoting if the following 55 predicates hold or not at the current state:

$$requested(n_1,n_1),\ requested(n_1,n_2),\cdots,\ requested(n_5,n_5)$$
$$replied(n_1,n_1),\ replied(n_1,n_2),\cdots,\ replied(n_5,n_5)$$
$$holds(n_1),\ holds(n_2),\cdots,\ holds(n_5).$$

DistAI chooses a maximum formula size for a candidate invariant, which defines the maximum number of quantified variables that can be used per domain and the maximum number of literals (a predicate or its negation) in the formula. DistAI projects data samples to subsamples, which only contain values of predicates that match the formula size. For example, given a formula with two variables $\{\forall N_1\ N_2\}$, indicating that candidate invariants start with $\forall N_1\ N_2 \cdots$, each subsample only contains the value of predicates related to two assigned nodes.

Enumeration. DistAI then enumerates all strongest candidate invariants that satisfy the subsamples, up to the maximum formula size. Invariants are expressed as formulas in first-order logic. For example, given a maximum formula size of at most two variables and two literals, the following three formulas could be enumerated, assuming they all satisfy the subsamples:

$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \quad (4)$$

$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \vee replied(N_2,N_1) \quad (5)$$

$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \vee \neg holds(N_1). \quad (6)$$

However, DistAI would only generate the first one as a candidate invariant because the first one implies the other two, so the latter two formulas can be skipped. The first formula is the strongest candidate invariant among the three formulas.

Monotonic refinement. DistAI feeds the candidate invariants and the protocol specification to IVy, which runs its SMT solver to check if the conjunction of the invariants with the safety property is an inductive invariant. If the solver passes, DistAI has succeeded. Succeeding means that if the conjunction of the invariants with the safety property holds before a protocol action is taken, each invariant still holds after the action is taken. Otherwise, IVy outputs which candidate invariants failed, and DistAI weakens each failed invariant and tries again with IVy with the candidate invariants, each failed invariant being replaced by weakened invariants with no more variables and literals than the maximum formula size. For the Ricart-Agrawala protocol, IVy shows that the invariant in Equation (4) is incorrect. The invariant is then weakened into Equations (5) and (6), among others, which will then be checked by IVy. Later, IVy will also invalidate Equation (5), and since it has reached the maximum formula size, it will be

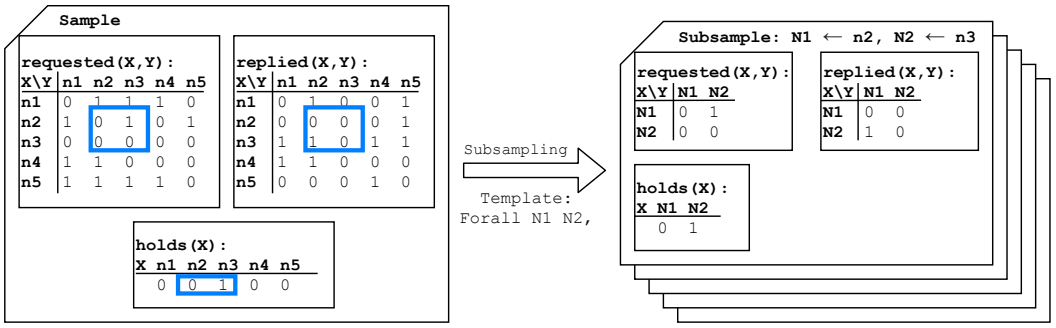


Figure 3: The subsampling process. The frame on the left shows a single sample state of a finite instance of the Ricart-Agrawala protocol with five nodes. A single subsample with two quantified variables $\{\forall N_1\ N_2\}$ is generated by mapping the quantified variables to concrete nodes in the finite instance, **n2** and **n3**, and extracting their associated values (shown in blue boxes in the sample frame). $0/1$ stand for false/true.

simply discarded. Equation (6) is never invalidated by IVy and will be part of the inductive invariant in the end. If IVy indicates that the safety property failed, it means that the formula size is not sufficient. DistAI will then increase the formula size by either increasing the maximum number of variables or the maximum number of literals, then re-run the process.

3 Two-Stage Sampling

Obtaining data samples for a distributed protocol requires simulating a finite instance of the protocol and recording the system state on each action. However, invariants are usually composed of quantified variables that impose constraints on all domains of the protocol, not just the specific domains of a finite instance. Therefore, DistAI projects the collected finite state samples into abstract subsamples on quantified variables that also apply to all domains of the protocol and represent potential predicates in the invariant. We refer to these two procedures as *sampling* and *subsampling* respectively, since many abstract subsamples can be generated from a single concrete data sample.

The two-stage sampling has four parameters: the absolute maximum number of instances to consider before terminating (*MI*), the maximum number of instances to consider before terminating if no further subsamples are generated (*MIS*), the number of actions to take when simulating a finite instance (*MA*), and the number of subsamples to generate from one data sample (*SD*). As we show in Section 6, DistAI's ability to find an inductive invariant is not sensitive to the specific values of these parameters, which are always set to their defaults of 1000, 20, 50, and 3, respectively.

Sampling. DistAI first translates the protocol from IVy into Python, with relations simulated by multidimensional arrays, and actions simulated by Python functions. This allows DistAI to efficiently simulate the protocol. The translation is not in the trusted computing base since learned invariants are eventually validated by IVy.

DistAI then simulates the protocol in Python from different valid initial states on randomly chosen finite instances of the protocol. DistAI randomly chooses an instance size from some range of sizes using a simple discrete uniform distribution. For each domain T (e.g., node), a protocol typically has some minimum instance size to function well, which we refer to as N^T_{min}. In practice, the minimum instance size N^T_{min} is determined as the maximum number of variables of type T in any relation; a protocol will not function well if its relations have variables that cannot be mapped to the instance size. For example, for a protocol with two relations $p(n_1:T_1,n_2:T_1,m_1:T_2)$ and $q(m_1:T_2)$, we have $N^{T_1}_{min}=\max(2,0)=2$ and $N^{T_2}_{min}=\max(1,1)=1$. The probability for choosing a given domain size N^T is then:

$$Pr[N^T]=1/w\ \ (N^T_{min}\leq N^T<N^T_{min}+w)$$

DistAI uses $w=4$ by default. This allows sampling from multiple instance sizes, but limits the instance sizes to within w of the minimum instance size for performance, as larger instances take more time to simulate.

For each valid initial state, DistAI simulates the protocol by performing *MA* number of actions. Since the distributed protocols are nondeterministic with regard to the next action taken (e.g., we do not know which node will send the next request or reply), multiple runs from the same initial state will result in different samples. Given an initial state s_0, DistAI uses the simple method formalized in Algorithm 1, to simulate a protocol, which randomly chooses an action from an action pool (line 6) with randomly chosen arguments from an argument pool (line 9) that satisfy the precondition (line 10). It then performs the action, records the new system state, and repeats the process (line 16-17). An action is removed from the action pool once its argument pool is exhausted, and the protocol terminates if the action pool is exhausted. Since some protocols may never terminate, *MA* defines an upper bound on the number of actions performed. Once the simulation completes, the set of reached states S is returned.

For example, for the Ricart-Agrawala protocol, during each iteration of the algorithm, DistAI first randomly selects one of the four possible actions: `request`, `reply`, `enter`, and `leave`. If `request` is selected, DistAI then randomly chooses the nodes for its two arguments, `requester` and `responder`. However, not every pair of nodes are valid arguments as the two nodes must satisfy lines 14-15, the two preconditions to legitimately trigger the `request` action under the protocol. If the current

Algorithm 1 Stochastic Sampling Algorithm.

Input: Protocol $\mathcal{P}$ with actions $\mathcal{A}$. Initial state s_0
Output: A simulation trace, represented by a set of states S

```
1:  S := {s_0}, s := s_0
2:  for iter := 1 to MA do
3:      action_pool := A
4:      action_found := false
5:      while ¬action_found ∧ |action_pool| > 0 do
6:          action := select_random(action_pool)
7:          args_pool := enum_arguments(s,a)
8:          while |args_pool| > 0 do
9:              args := select_random(args_pool)
10:             if precondition_holds(P,s,action,args) then
11:                 action_found := true
12:                 break
13:             args_pool := args_pool \ {args}
14:         action_pool := action_pool \ {action}
15:     if action_found then
16:         s:=execute_protocol(P,s,a,args)
17:         S:=S∪{s}
18:     else
19:         break
20: return S
```

⟨`requester`,`responder`⟩ pair violates the precondition, DistAI removes it from the argument pool and randomly selects another one. This repeats until a valid pair of arguments is found or the pool is exhausted, in which case DistAI removes `request` from the action pool and selects another action.

After each iteration, DistAI logs the current system state, represented by the value of all the relations. In Figure 2, that is the value of predicates $requested(N_1,N_2)$, $replied(N_1,N_2)$, and $holds(N_1)$ for all N_1 and N_2. Figure 3 shows a sample of the Ricart-Agrawala algorithm for an instance size of five nodes in the left frame. The *requested* and *replied* relations each take two nodes as arguments, so their samples record the relations for all possible pairs of nodes, resulting in 25 boolean values each. The *holds* relation only takes a single node as argument, so five boolean values are recorded, one for each of the five nodes in the protocol instantiation.

DistAI can also simulate protocols calling other protocols, even when the protocol being called is a blackbox. When a protocol calls another protocol through a blackbox interface, described by a specification without a concrete implementation, DistAI treats it as an action with nondeterministic behavior. If the action is selected with arguments that satisfy its preconditions, DistAI selects randomly updated states that satisfy its postconditions as the execution result.

Our simple stochastic sampling procedure, while very efficient, may not achieve high coverage and can leave corner cases uncovered. More sophisticated techniques [1, 25, 30] can be applied to improve coverage for complex protocols with sparse inputs and difficult to reach states. However, the correctness of learned invariants is guarded by the Z3 SMT solver used by IVy. If the samples are incomplete and the invariants fail the SMT check, DistAI will iteratively refine the invariants until they are correct, as discussed in Section 5.

Subsampling. The data samples from protocol simulation may be of all different lengths depending on the instance size used. We want to map the concrete samples from simulation to an invariant template, the small set of quantified variables that may appear in the invariant, denoted by τ. Given a set of data samples and an invariant template, DistAI applies a subsampling procedure translating the variable length data samples to fixed length vectors, which we call *subsamples*. Formally, a subsample corresponds to an assignment α of the variables in τ and contains the values of relations given the assignment to the template. Subsamples taken with an invariant template with variables $V_1,...,V_n$ can then be used to learn invariants (denoted I) on those variables:

$$\tau = \{\forall V_1 ... V_n\} \qquad \forall V_1 ... V_n.\ I(V_1,...,V_n).$$

For example, in the Ricart-Agrawala protocol, the relations *requested* and *replied* each operate on two nodes, so the initial template is $\tau = \{\forall N_1\ N_2\}$. Under this template, there are only 10 predicates that may appear in an invariant formula, namely:

$$requested(N_1,N_1),\ requested(N_1,N_2),\ requested(N_2,N_1),$$
$$requested(N_2,N_2),\ replied(N_1,N_1),\ replied(N_1,N_2),$$
$$replied(N_2,N_1),\ replied(N_2,N_2),\ holds(N_1),\ holds(N_2).$$

For this template, a 5-node data sample can induce $5*4=20$ subsamples, by first assigning one node X_1 for N_1 and then another node X_2 for N_2 , as illustrated in Figure 3. Since there are 10 predicates, each subsample has 10 possible boolean values, so one data sample results in $20\times 10=200$ boolean values.

Enumerating all valid subsamples from each sample can be computationally undesirable, especially for multi-domain protocols. If we add a new domain `msg`, and let the template be $\{\forall\ N_1\ N_2 \in \texttt{node}, M_1\ M_2 \in \texttt{msg}\}$, then a sample with five nodes and 10 messages will induce 1,800 subsamples. Therefore, DistAI randomly chooses *SD* valid subsamples from each sample. Two-stage sampling terminates when *MI* instances have been simulated, or no new subsample is found after simulating *MIS* consecutive instances of the protocol. The subsamples are then deduplicated and passed on to invariant enumeration.

Although DistAI's sampling has several parameters, they do not need to be manually tuned to find inductive invariants. We use the default values for all protocols. For example, when *MA* or *SD* become larger, each simulation round will take longer, but fewer rounds will be required to converge. Similarly, a small *MI*/*MIS* may stop two-stage sampling prematurely, but the missing states will be resolved later by monotonic refinement. The parameters do not affect whether DistAI finds

inductive invariants, only how fast it finds them. Sampling is useful simply as a performance optimization that reduces the number of SMT queries required during refinement.

4 Candidate Invariant Enumeration

Algorithm 2 shows DistAI's enumeration-based algorithm to generate candidate invariants from the subsamples obtained in Section 3. To reduce the number of candidate invariants required for covering the invariant space and reduce the maximum number of literals needed for finding the inductive invariant, we partition the invariant space into multiple regions, each represented by a constrained template called a *subtemplate*. We then enumerate all possible invariants in each region (i.e., under each subtemplate), and retain candidate invariants that hold for the collected subsamples.

Template decomposition. Before enumerating candidates invariants, we decompose templates into subtemplates that incorporate additional constraints (line 1). A template with N variables in the same domain will be split into N subtemplates which have from 1 to N variables. A subtemplate with more variables is said to be larger than a subtemplate with fewer variables. For example, a template $\tau = \{\forall N_1\ N_2\}$ will be split into two subtemplates, $\tau_1 = \{\forall N_1\ N_2.\ N_1 = N_2\} = \{\forall N_1\}$ and $\tau_2 = \{\forall N_1\ N_2.\ N_1 \neq N_2\}$, abbreviated as $\{\forall N_1 \neq N_2\}$, with τ_2 being larger than τ_1. All operations that use subtemplates in Algorithm 2 traverse them from smallest to largest (lines 2 & 5).

This subtemplate optimization reduces the cost of enumeration in two ways. First, subtemplates reduce the number of candidates that need to be enumerated due to symmetry. For example, for the Ricart-Agrawala protocol, when using template τ, both of the following invariants will be enumerated:

$$\forall N_1\ N_2.\ \neg replied(N_1,N_1) \qquad \forall N_1\ N_2.\ \neg replied(N_2,N_2).$$

On the other hand, when using the subtemplate τ_1, the equivalent enumeration would only result in one candidate:

$$\forall N_1.\ \neg replied(N_1,N_1). \tag{7}$$

Furthermore, DistAI will project Equation (7) using τ_2 to the following candidate invariants:

$$\forall N_1 \neq N_2.\ \neg replied(N_1,N_1) \quad \forall N_1 \neq N_2.\ \neg replied(N_2,N_2),$$

which are then marked as validated using τ_1, avoiding further redundant validations. We refer to this as invariant projection.

Second, subtemplates can reduce the maximum number of literals in the invariant formula. For example, one invariant of the Ricart-Agrawala protocol (Equation 3) can be rewritten as:

$$\forall N_1\ N_2.\ \neg(N_1 \neq N_2) \vee \neg holds(N_1) \vee replied(N_1,N_2) \tag{8}$$

This is a disjunction of three literals under the full template τ. However, using subtemplate $\tau_2 = \{\forall N_1 \neq N_2\}$, an equivalent form of this invariant can be learned using a formula size with a maximum of two literals:

$$\forall N_1 \neq N_2.\ \neg holds(N_1) \vee replied(N_1,N_2)$$

We can denote an invariant as $\tau : inv$, where τ is the subtemplate under which the invariant formula inv is found and inv is expressed as a disjunction of literals. In this example, we effectively can denote the same invariant using subtemplate τ_2 as $\tau_2 : inv'$, where one literal that was previously part of inv is no longer part of inv' because it is now a part of τ_2. Because DistAI operates in formula space and the time complexity of enumeration is exponential in the maximum number of literals, such a small reduction in the number of literals can have a significant impact on the overall cost of enumeration.

Subtemplates can also reduce the maximum number of literals by exploiting another form of symmetry. If there is a total order on a domain, such as with node identifiers, we will further assign an order on the variables in the subtemplate and strengthen $\{\forall N_1 \neq N_2\}$ into $\{\forall N_1 < N_2\}$ Because of symmetry, invariant formulas with $\{\forall N_1 < N_2\}$ are equivalent to those with $\{\forall N_2 < N_1\}$, so we do not need to enumerate the latter once we have done the former. This is useful since inductive invariants often contain comparisons between variables in a domain with a total order. For example, this optimization helps reduce the maximum number of literals required from six to four for the database chain replication protocol evaluated in Section 6.

Subtemplates work with multiple domains as well. Consider a protocol with two domains, T_1 and T_2, where T_1 defines a total order, and the template τ is $\{\forall X_1\ X_2\ X_3 \in T_1, Y_1\ Y_2 \in T_2\}$. After

Algorithm 2 Invariant Enumeration Algorithm.

Input: Template τ, subsample table ST, max-literal l
Output: A set of invariants I^*

```
1:  subtemplates := decompose_templates(τ)
2:  for τ' ∈ traverse(subtemplates) do
3:      proj_table[τ'] := ST_|τ'
4:      I[τ'] := ∅
5:  for τ' ∈ traverse(subtemplates) do
6:      predicates := proj_table[τ'].header
7:      P_τ := predicates ∪ {¬p | p ∈ predicates}
8:      for n := 1 to l do
9:          for inv ∈ combinations(P_τ, n) do
10:             if check_subset_exists(inv, I[τ']) then
11:                 continue
12:             if check_inv_holds(inv, proj_table[τ']) then
13:                 I[τ'] := I[τ'] ∪ {inv}
14:     for τ'_succ ∈ successors(τ') do
15:         for inv ∈ I[τ'] do
16:             I[τ'_succ] := I[τ'_succ] ∪ proj_inv(inv, τ', τ'_succ)
17: I* := {(τ' : inv) | τ' ∈ subtemplates, inv ∈ I[τ'], inv was checked against subsamples}
```

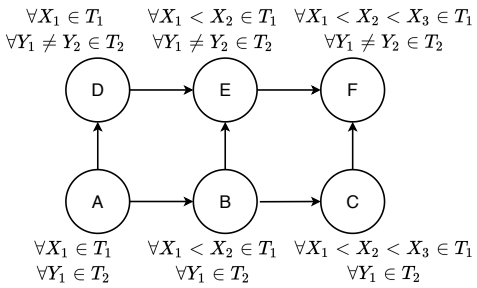


Figure 4: Dependency relations between the six subtemplates derived from the template $\{\forall X_1\ X_2\ X_3 \in T_1, Y_1\ Y_2 \in T_2\}$.

template decomposition we get six subtemplates, as shown in Figure 4. For multiple domains, there may not exist a total ordering of all subtemplates from smallest to largest, so the only requirement for the order of traversal of subtemplates is that the quantified variables in each subtemplate are not a subset of a prior one, such that formulas can always be validated with the smallest possible subtemplate. In Figure 4, $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ is a valid traversal order, while $A \rightarrow D \rightarrow E \rightarrow B \rightarrow C \rightarrow F$ would be invalid because the quantified variables $\{X_1, X_2, Y_1\}$ for subtemplate B are a subset of $\{X_1, X_2, Y_1, Y_2\}$ of subtemplate E. We follow graph terminology and call subtemplates B and D the *successors* of A, and A the *predecessor* of B and D.

Subtemplate projection. Because DistAI uses subtemplates for candidate enumeration instead of templates, the full subsamples of the invariant template need to be projected onto each subtemplate (line 3). This is done similarly to how data samples are projected onto full subsamples using an invariant template, as discussed in Section 3, except in this case, full subsamples are projected onto subsamples using a subtemplate. For example, for this multi-domain protocol, when projecting full subsamples to subtemplate $\{\forall X_1 < X_2 \in T_1,\ \forall Y_1 \in T_2\}$ (node B in Figure 4), there are six possible variable mappings: $\{X_1 \rightarrow X_1, X_2 \rightarrow X_2, Y_1 \rightarrow Y_1\}, ..., \{X_1 \rightarrow X_2, X_2 \rightarrow X_3, Y_1 \rightarrow Y_2\}$. Note that the total order on T_1 needs to be preserved, otherwise there would be 12 possible mappings. Similarly, going to back the Ricart-Agrawala protocol example, when projecting full subsamples of the invariant template τ to subtemplate τ_1, there are two possible variable mappings: $\{N_1 \rightarrow N_1, N_1 \rightarrow N_2\}$.

Subtemplate candidate enumeration. DistAI enumerates and checks all possible candidates for each subtemplate τ' (lines 6-13). Each subtemplate τ' has a certain number of predicates m. For example, for the Ricart-Agrawala protocol using template τ_1, there are three predicates: $requested(N_1,\ N_1)$, $replied(N_1,\ N_1)$, $holds(N_1)$. DistAI adds the m predicates p_1, p_2, ..., p_m and their negations $\neg p_1, \neg p_2, ..., \neg p_m$ to the literal pool $\mathcal{P}_\tau$ (line 7).

Given a formula size with the maximum number of literals l, DistAI enumerates all subsets of $\mathcal{P}_\tau$ with size at most l as candidate invariants. For example, if $m = 3$ and $l = 1$, there would be six candidate invariants: $p_1, \neg p_1, p_2, \neg p_2, p_3, \neg p_3$. By default, DistAI initially sets $l = 3$, and iteratively increases it later in the refinement process (see Section 5). We only consider invariants in the form of disjunctions of literals since invariants with conjunctions can simply be split into multiple invariants. If a candidate invariant C includes both a predicate and its negation, it will be discarded. If not, DistAI checks the validity of C against the subsamples. If C is satisfied by all subsamples for the subtemplate, C is added to the set of generated invariants, which we refer to as the invariant set.

DistAI exploits symmetry to prune the candidate enumeration space. Whenever an invariant is learned, we permute the quantified variables with the same type and emit equivalent candidates without needing to check if they are satisfied by the subsamples. For example, under the subtemplate $\{\forall\ X \neq Y \in T_1, A \neq B \in T_2\}$, if $p(X,Y) \vee \neg q(Y) \vee r(X,A,B)$ is an invariant, then $p(Y,X) \vee \neg q(X) \vee r(Y,B,A)$, along with two other formulas, are also invariants.

Enumeration is ordered by the number of literals in the candidate invariants, and any candidate that is weaker than an invariant already added to the invariant set is skipped (lines 8-11). For example, if we already know $p \vee \neg q$ is an invariant, then for any predicate r, $p \vee \neg q \vee r$ is guaranteed to be a valid but weaker invariant, and can be skipped in the enumeration. Based on Figure 3, applying this enumeration procedure to the Ricart-Agrawala protocol with subtemplate $\{\forall N_1\}$ and $l = 2$ results in the following two generated invariants:

$$\neg requested(N_1,N_1) \qquad \neg replied(N_1,N_1)$$

Invariant projection. After finding all candidates on one subtemplate, DistAI calculates the projection of the candidates on each successor, then propagates the projection and moves on to enumerate the next subtemplate (line 14-16). This reduces the cost of validating candidates using larger subtemplates against their subsamples. For example, for the Ricart-Agrawala protocol, suppose we have learned two invariants $\neg requested(N_1, N_1)$ and $\neg replied(N_1, N_1)$ under subtemplate $\tau_1 = \{\forall N_1\}$. Before enumerating candidates under subtemplate $\tau_2 = \{\forall N_1\ N_2\}$, we know the following four candidates must hold under τ_2 because they are projections of the learned invariants under the simpler template τ_1:

$$\neg requested(N_1,N_1) \qquad \neg replied(N_1,N_1)$$
$$\neg requested(N_2,N_2) \qquad \neg replied(N_2,N_2)$$

As a result, these four candidates can simply be added to the invariant set under τ_2 without enumerating and validating them against any subsamples (line 16). Any weaker candidates will also be skipped, further reducing the cost of enumeration. For example, $\neg requested(N_1,N_1) \vee holds(N_1)$ can be skipped since it is weaker than $\neg requested(N_1,N_1)$.

Strongest possible invariant set. Finally, after traversing all subtemplates, DistAI unions together the subtemplate invariant sets to form the initial set of generated invariants (line 17) which will be fed to IVy. Since all possible candidate invariants have been considered for each subtemplate, we can prove that, for any invariant *inv* (in the form of disjunctions of

literals) under template τ with a maximum number of literals of no more than l, there must exist an invariant inv' in the constructed invariant set such that $inv' \Rightarrow inv$. General invariants can be converted into conjunctive normal form (CNF) and then split into multiple invariants in the form of disjunctions of literals. Thus, the initial invariant set for a subtemplate is a *strongest possible* invariant set that is guaranteed to be at least as strong as the inductive invariant if there are no more than τ variables, also known as quantifiers, and l literals.

Intuitively, since each subtemplate provides the strongest possible invariant set, the invariant checked by IVy, which is constructed using the conjunction of all invariants across the union of subtemplate invariant sets, should also be the strongest with regard to the subsamples. In practice, when unioning the invariant sets, we can exclude invariants generated by projection from predecessors because they can be implied by their original counterparts. We formalize this in Theorem 1:

Theorem 1. *Let I^* be the output of Algorithm 2. For any invariant set I under template τ with a maximum number of literals of no more than l, if I is satisfied by every subsample, then $I^* \Rightarrow I$.*

Proof. First we consider a variant of Algorithm 2 where Line 17 does not exclude invariants generated by projection. We prove this by contradiction. Suppose there exists I under template τ with a maximum number of literals of no more than l, I is satisfied by every subsample and $I^* \not\Rightarrow I$. Consider any invariant $\tau' : inv \in I$ but $\tau' : inv \notin I^*$. Recall each individual invariant is a disjunction of literals, assuming CNF. Since I is satisfied by every subsample, $\tau' : inv$ is also satisfied by every subsample. If we reach lines 12-13 in Algorithm 2, it will be added to the invariant set. The only possibility of $\tau' : inv \notin I^*$ is that the branch condition at line 10 evaluates to $true$. However, this indicates that a subset of inv is already in the invariant set. The subset of inv implies inv (e.g., $p \vee q \Rightarrow p \vee q \vee r$). So we still have $I^* \Rightarrow \tau' : inv$. To conclude, every $\tau' : inv \in I$ but $\tau' : inv \notin I^*$ can be implied by I^*, which is a contradiction to $I^* \not\Rightarrow I$.

Now we exclude invariants generated by projection and get a new I^*_{new}. From lines 15-16, every excluded invariant can be implied by another invariant in its predecessor subtemplate, so we can show $I^* \Leftrightarrow I^*_{new}$, thus completing the proof. □

Constants and function symbols. Although the discussion above assumes a literal can only be a predicate or its negation, DistAI also supports constants and function symbols as literals. For example, given a template $\{\forall X\ Y \in T\}$ and a constant $c \in T$, DistAI considers $X = c$ and $Y = c$ as two independent predicates and reasons about them like any other predicate. As another example, given a template $\{\forall X_1\ X_2 \in T_1, Y_1 \in T_2\}$ and a function $f : T_1 \to T_2$, DistAI can introduce $Y_2 = f(X_1), Y_3 = f(X_2)$ and treat Y_2, Y_3 as variables like Y_1.

5 Monotonic Invariant Refinement

When DistAI feeds the enumerated invariants to IVy, IVy may find that the conjunction of the invariants and the safety property are not inductive and return a list of invariants that failed. This is likely to happen at least for the initial invariants that DistAI enumerates as its sampling is not guaranteed to be complete. Because sampling is not complete and is primarily to improve performance, DistAI may generate invariants that would not hold if sampling was done for more protocol instances. In general, when IVy indicates that an invariant fails, it is difficult to know whether the solution is to weaken or strengthen the invariant. Prior work uses different methods to evade this challenge but gives no fundamental solution [19, 35].

DistAI provides a simple and clean solution to this problem by starting with the strongest possible invariants and ensuring that the invariants remains the strongest possible ones throughout the refinement process. For each invariant that fails, which we refer to as a broken invariant, DistAI applies *mimimum weakening* to the invariant. The candidate invariant space becomes strictly smaller after each failure. DistAI ensures that the conjunction of the weakened invariants will remain stronger than the eventual invariants that must be added to the safety property to make it inductive, if it is expressible under the current template τ and maximum number of literals l. The overall process is guaranteed to converge to find the inductive invariant.

Algorithm 3 Minimum Weakening Algorithm.

Input: Invariant set
$I[\tau']$ for each subtemplate τ', and the broken invariant $\tau'_0 : inv$
Output: Updated invariant set $I[\tau']$ for each subtemplate τ'

```
1:  I[τ'_0] := I[τ'_0] \ {inv}
2:  if inv.length < l then
3:      for literal ∈ valid_literals(τ'_0) do
4:          if literal ∉ inv then
5:              new_inv := inv ∪ {literal}
6:              if ¬ check_subset_exists(new_inv, I[τ'_0]) then
7:                  I[τ'_0] := I[τ'_0] ∪ {new_inv}
8:  for τ'_succ ∈ successors(τ'_0) do
9:      new_invs:=proj_inv(inv, τ'_0, τ'_succ)
10:     for new_inv ∈ new_invs do
11:         I[τ'_succ] := I[τ'_succ] ∪ {new_inv}
```

Algorithm 3 shows the minimum weakening algorithm used, given an initial invariant set and a broken invariant. We denote an invariant as $\tau' : inv$, where τ' is the subtemplate under which inv is found and inv is the invariant expressed as a disjunction of literals. The algorithm consists of three steps. First, DistAI removes the broken invariant from the initial invariant set. When IVy returns that $\tau'_0 : inv$ fails, DistAI removes $\tau' : inv$ from the invariant set that was initially passed to IVy (line 1).

Second, DistAI finds all weakened versions of the broken invariant and adds them back to the invariant set. A weakened version of $\tau' : inv$ is created by add one more literal via disjunction to inv (lines 2-7). For example, suppose $inv = p \vee \neg q$ is

rejected by IVy. Recall that during the invariant enumeration, since $p \vee \neg q$ was considered as an invariant, for all literals r, the candidate $p \vee \neg q \vee r$ would be skipped. Now, for any literal r, $p \vee \neg q \vee r$ becomes a meaningful invariant. DistAI updates the invariant set by adding the weakened invariants back to the invariant set as long as they can not be implied by some other invariant that is already in the invariant set (e.g., $p \vee r$). If the broken invariant has reached the maximum number of literals, this second step will be skipped.

Third, DistAI projects the broken invariant to higher subtemplates, and adds all such projections. For each successor τ'_{succ} of τ'_0, DistAI adds all the projections of *inv* on τ'_{succ} to the invariant set (line 8-11). To see why this is necessary, consider the following candidate invariant in some leader election protocol:

$$\forall X \in T.\ \neg leader(X). \tag{9}$$

This asserts no one can be a leader. This invariant may fail in IVy because the SMT solver observes a system state $\{leader(i_1), \neg leader(i_2), i_2 < i_1\}$ (suppose T has total order). Recall that DistAI uses a traversal order to enumerate invariants under different subtemplates, and the invariants from smaller subtemplates will be projected to larger subtemplates to avoid repeated enumeration. So previously, the following two candidate invariants, under a larger subtemplate $\{\forall X < Y \in T\}$, were skipped because they could be implied by Invariant (9).

$$\forall X < Y \in T.\ \neg leader(X) \tag{10}$$
$$\forall X < Y \in T.\ \neg leader(Y). \tag{11}$$

But now, after Invariant (9) is invalidated and removed, we need to reconsider Invariants (10) and (11) and add them to the candidate invariant set. We validate the new invariants using IVy again. If successful, DistAI outputs the current invariant set as the inductive invariant, otherwise we will enter the next refinement round. In this case, if the distributed protocol has the property that only the greatest user can be the leader, then Invariant (11) will be invalidated in a later round, while Invariant (10) is likely to be correct and remain valid to the end.

The three-step minimum weakening procedure guarantees after any number of refinement rounds, the invariant set is always a strongest possible one that is satisfied by *all* the subsamples. This "strongest possible" property implies that throughout refinement, the invariant set is always stronger than the correct invariant set required for an inductive invariant, so whenever an invariant fails, we should always weaken the broken invariant. The guarantee can be formally stated as:

Theorem 2. *Let I^* be the invariant set after n refinement rounds, and $B_n = \{\tau'_1 : inv_1, \tau'_2 : inv_2, ..., \tau'_n : inv_n\}$ be the broken invariants in each round. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_n, then $I^* \Rightarrow I$.*

Proof. We prove this by induction on the number of rounds. The base case is simple. In Round 0, there is no broken invariant, and the statement degenerates to Theorem 1. Now we focus on the induction case. Suppose after k refinement rounds, we get invariant set I^*_k. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_k, then $I^*_k \Rightarrow I$.

Now we come to round $k+1$. We prove by contradiction. Suppose we have an invariant set I under template τ with no more than l literals such that 1) I is satisfied by every subsample, 2) I does not imply any broken invariant in B_{k+1}, and 3) $I^*_{k+1} \not\Rightarrow I$. Consider any invariant $\tau' : inv$ such that $I \Rightarrow \tau' : inv$ but $I^*_{k+1} \not\Rightarrow \tau' : inv$. From the induction hypothesis, we know $I^*_k \Rightarrow \tau' : inv$. Let $\tau'_{k+1} : inv_{k+1}$ be the invalidated invariant in round $k+1$. From the algorithm, $\tau'_{k+1} : inv_{k+1}$ is the only removed invariant in this round. Since each invariant is a disjunction of literals, we can show $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$. In other words, the hypothetical "missing" invariant must be implied by the removed invariant. We further know either *inv* includes more literals than *inv*, or τ' includes more quantified variables not in τ'_{k+1}, or both. Otherwise we have $\tau' : inv \Rightarrow \tau'_{k+1} : inv_{k+1}$. Then $\tau' : inv = \tau'_{k+1} : inv_{k+1}$, a contradiction to $I \Rightarrow \tau' : inv$ and I does not imply the broken invariant $\tau'_{k+1} : inv_{k+1}$.

Now we consider the two cases separately. 1) *inv* includes a literal p not in inv_{k+1}. Consider the formula $F = \tau_{k+1} : inv_{k+1} \vee p$. From $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $F \Rightarrow \tau' : inv$. However, F is added to the new invariant set I^*_{k+1} unless it can be implied by existing invariants (Line 3-7 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow F \Rightarrow \tau' : inv$. 2) τ' includes a quantified variable X not in τ'_{k+1}. Consider the formula $G = \tau'' : inv_{k+1}$, where τ'' is τ' extended with X. Again, from $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $G \Rightarrow \tau' : inv$. However, G is added to the new invariant set I^*_{k+1}(Line 8-11 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow G \Rightarrow \tau' : inv$. In both 1) and 2), we reach $I^*_{k+1} \Rightarrow \tau' : inv$, which means the "missing" invariant is already implied by the existing invariant set output by the algorithm, a contradiction. □

Intuitively, Theorem 2 ensures that starting from a too strong invariant set, the minimum weakening steps never over-weaken the invariants and "bypass" the correct invariants in between. Combined with Theorem 1, which guarantees that monotonic refinement indeed starts from the strongest invariant set, we have the following corollary:

Corollary 1. *If there exists a correct invariant set expressible with template τ and maximum number of literals l, then the refinement procedure will terminate with one such invariant set within a finite number of rounds, otherwise the refinement procedure will terminate with a broken safety property.*

Sometimes, the weakened versions of a broken invariant are all discarded in the end. Then, minimum weakening provides no benefits versus just removing the broken invariants. In practice, DistAI first applies only the first step of minimum weakening — removing the broken invariants. Then if failed, DistAI applies refinement again with the first and second step. If failed again, DistAI applies the standard three-step

minimum weakening in Algorithm 3. This practice optimizes performance when the weakened versions of a broken invariant are all discarded while maintaining Theorem 2 and Corollary 1.

If available, DistAI can use counterexamples to check weakened invariants, only adding them if they satisfy the counterexamples. However, DistAI's refinement procedure currently does not use them because obtaining counterexamples from IVy for an entire invariant set is extremely inefficient. When IVy is configured to return a counterexample, it halts early and returns the counterexample once it identifies the first broken invariant in a set. Using IVy in this configuration would force DistAI to weaken broken invariants one at a time and perform many redundant SMT checks of the invariant set through IVy, instead of weakening all failed invariants at once between each IVy call.

Convergence and Feedback loop. Since the invariant set is weakened after each refinement round, we can prove that the refinement procedure terminates in a finite number of rounds, resulting in an inductive invariant set.

If the safety property has never been violated during the refinement process, the resulting set is the correct inductive set and can derive the desired safety property. If, at any point, the safety property is violated and needs to be weakened (when all other candidates are weak enough), it means that the correct invariant set cannot be expressed under the current formula size, with its per-domain template size and maximum number of literals. DistAI will then increase the formula size, increasing the per-domain template size or the maximum number of literals, and relearn the invariants. By default, DistAI increases the formula size by alternating between increasing the maximum number of literals or increasing the template size, the latter by increasing the number of quantified variables for each domain in the template. For example, in Figure 4, i) we first increase the maximum number of literals by one, ii) if it fails, increase the template size by adding a new variable in type T_1, iii) if fails again, add another new variable in T_2, iv) and if still fails, increase the maximum number of literals by one again.

After increasing the formula size, we redo sampling, enumeration, and refinement. Since any invariant contains a finite number of quantified variables and a finite number of literals, the feedback loop will eventually reach a template and literal size large enough to express the correct invariant set if one exists. Once a sufficient formula size is reached, Corollary 1 guarantees that a correct invariant set will be generated. Therefore, DistAI provides the following end-to-end convergence guarantee:

Theorem 3. *If the safety property of a protocol is provable with a $\exists$-free invariant set, then DistAI will terminate with one such invariant set in finite time.*

Theorem 3 guarantees conditional convergence of DistAI. However, if the safety property does not hold for the protocol or existential quantifiers are necessary to prove it correct, DistAI may continue in the feedback loop forever.

6 Evaluation

To demonstrate its effectiveness at determining inductive invariants, we implemented and evaluate DistAI on a collection of 14 distributed protocols, including all 7 protocols previously evaluated with I4 [18]. The implementation consists of 1.6K lines of Python code for protocol simulation and sampling and 1.6K lines of C++ code for enumeration and refinement. For comparison, we also evaluated I4 and FOL-IC3, in both cases using the implementations created by the original authors. All experiments were performed on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM, and a 512GB Intel SSD Pro 600p. Table 1 shows the results for each protocol, including the number of domains and relations for each protocol as indicators of protocol complexity.

DistAI outperforms both I4 and FOL-IC3 in terms of the number of protocols for which it infers the correct invariants. DistAI automatically infers the correct invariants for 13 out of the 14 protocols, only failing for Paxos, on which both I4 and FOL-IC3 also fail. I4 only solves 9 protocols, and FOL-IC3 only solves 3 protocols using its default setting, which searches over all first-order logic formulas, but improves to solving 9 protocols if an option is enabled that limits the search space to only $\forall$ quantifiers. Each approach was allowed to run for an entire week, 168 hours, per protocol before timing out, more than two orders of magnitude longer than the worst runtime reported in Table 1.

DistAI and I4 only time out trying to solve Paxos, but FOL-IC3 times out on many protocols. This is because DistAI only uses the SMT solver for validating rather than generating invariants, I4 uses a model checker to generate invariants only for a specific, small instance, while FOL-IC3 invokes the SMT solver to generate invariants for the general protocol, multiple times for each invariant, which is undecidable in general and very expensive in practice. FOL-IC3 performs worse with the default setting since the formula search space is larger and the SMT solver performs worse for formulas with existential quantifiers. In fact, FOL-IC3 fails for database chain replication, decentralized lock, and distributed lock with the default setting because Z3, the underlying SMT solver, fails and reports `unknown`, indicating that the formula generated by FOL-IC3 does not fall in the supported decidable fragment of first-order logic. In contrast, DistAI never generates an undecidable formula.

Although both DistAI and I4 fail to solve Paxos, a complex and realistic consensus protocol, the reasons for the failures are different. I4 fails because its model checker is unable to produce any candidate invariants. Model checking is complex and quite resource intensive, and I4's authors report its model checker runs out of memory trying to solve Paxos [18]. In contrast, DistAI produces candidate invariants, but it fails because it does not support invariants with existential quantifers, which Paxos requires; I4 also has this limitation. Upon failed refinement, DistAI keeps increasing the formula size until it times out or exhausts memory. By manual

Distributed Protocol	Domains	Relations	Variables	Literals	Refinements	Invariants	DistAI time(s)	I4 time(s) final	I4 time(s) total	FOL-IC3 time(s) ∀	FOL-IC3 time(s) default
asynchronous lock server [5]	2	5	3	2	0	12	1.1	generalize fail[s]		6.9	-[*]
chord ring maintenance [24]	1	8	3	4	48	163	52.8	586.1[‡]	594.4	-[*]	-[*]
database chain replication [24]	4	13	7	4	158	66	58.8	20.2[‡]	63.1	-[*]	Z3 fail
decentralized lock [9]	2	2	4	2	150	16	9.4	generalize fail[s]		37.1[d]	Z3 fail
distributed lock [24]	2	4	4	3	82	45	12.6	152.1[‡]	204.7	1451.3	Z3 fail
hashed sharding [11]	3	3	5	2	0	15	1.1	nondet fail[s]		9.2	-[*]
leader election [24]	2	3	6	3	0	17	1.9	4.9[‡]	4.9	26.3	-[*]
learning switch [24]	2	4	4	3	8	71	27.6	10.5[‡]	12.4	-[*]	-[*]
lock server [24]	2	2	2	2	0	1	0.8	0.5[‡]	0.8	0.5	2.1
Paxos [13, 15, 23]	4	9	-	-	-	-	-[*]	-[*]	-[*]	-[*]	-[*]
permissioned blockchain [17]	4	10	6	3	2	13	4.9	blackbox fail[s]		21.2	-[*]
Ricart-Agrawala [26]	1	3	2	2	0	6	0.9	0.8	0.8	0.7	3.2
simple consensus [11]	3	8	5	3	19	50	23.3	41.8	68.7	-[*]	-[*]
two-phase commit [18]	1	7	2	3	3	30	1.9	3.1[‡]	8.0	3.4	7.9

[*] Time out after 1 week.

[‡] I4 runtimes on our machine are similar (6 out of 7 protocols slightly faster) to those previously reported for I4 [18].

[s] "generalize fail" means I4's implementation fails to convert invariants from the AVR model checker to generalized universally quantified invariants. "nondet fail" means failed on nondeterministic initialization. "blackbox fail" means error triggered on reasoning of blackbox functions.

[d] FOL-IC3 initially completed in less than a second, but this turned out to be incorrect due to a bug in the mypyvy protocol specification used by FOL-IC3, which does not exist in the Ivy protocol specification used by DistAI and I4.

Table 1: Evaluation results on 14 distributed protocols from multiple sources.

inspection, we find that DistAI infers all ∃-free invariants for Paxos. FOL-IC3 supports finding invariants with existential quantifiers, but it also fails to solve Paxos, the one protocol in our evaluation with existential quantifiers.

The most common reason overall why I4 fails to solve protocols is its dependency on modeling checking a small size implementation of the protocol to generate candidate invariants. I4 also fails to infer the correct invariants for decentralized lock and asynchronous lock server because it cannot generalize the candidate invariants generated by the model checker for a small size implementation to universally quantified invariants. Although I4 succeeds on lock server, it fails on asynchronous lock server because the latter explicitly models packet loss in the network, resulting in more complex invariants.

DistAI takes a fundamentally different approach that does not require model checking a finite instance. DistAI operates in formula space, allowing it to enumerate invariants that hold for any instance size. It optimizes the enumeration by running protocol simulations across different size systems, but does not rely on the simulations to find candidate invariants, only to reduce the number of invariants it needs to enumerate. By taking this data-driven approach, it is able to produce better initial invariants to achieve greater success with more protocols and guarantee success if there are no invariants with existential quantifiers. Unlike I4, DistAI is simple and self-contained, avoiding the need for, and dependence on, a complex external model checker that, like all complex software, may have bugs.

Permissioned blockchain is another example that demonstrates the effectiveness of DistAI. It has a blackbox Byzantine broadcast protocol as a subprocedure. In permissioned blockchain, n users have a synchronized clock. At epoch E, only one user n_E, the round-robin leader of the epoch, can (optionally) propose a block, if it has found a valid one extending its longest chain. The epoch leader uses the Byzantine broadcast protocol to broadcast the block in the P2P network. An honest user always adds all outstanding transactions in the block it proposed and follows the Byzantine broadcast protocol, while an adversary can neglect certain transactions, delay block proposal, and send conflicting block messages to any node at any epoch, regardless of who the leader is. A Byzantine broadcast protocol satisfies *agreement*, if all honest users always share the same eventual result regardless of the leader is honest or not. A Byzantine broadcast protocol satisfies *validity*, if when the leader is honest, all honest users will eventually decide on the message of the leader. A blockchain satisfies *consistency*, if at any epoch, all honest users have the same view of the blockchain (i.e., no forks or orphaned blocks). That is, for any two honest users at any time, a block is either confirmed by both, or confirmed by none. A blockchain satisfies *liveness*, if all transaction will be confirmed within a finite number of epochs.

DistAI successfully proves that for any Byzantine broadcast procedure that satisfies agreement and validity, the resulting permissioned blockchain satisfies consistency and liveness[1]. The Byzantine broadcast procedure is described by pre-conditions and post-conditions in IVy without the need for an executable implementation. When simulating the blackbox Byzantine procedure, DistAI simply picks a random state that satisfies the post-condition as the execution result. This random selection may leave corner cases uncovered, but the eventual correctness is guarded by the SMT solver, and we monoton-

[1] We prove a variant of the liveness property — if the leader of epoch T is honest, then all transactions before T will be confirmed at T. The original liveness property cannot be encoded as a safety property, thus falling out of the scope of DistAI, I4, and FOL-IC3.

Distributed Protocol	Sample	Enumerate	Refine	Total
asynchronous lock server	0.6	0.0	0.5	1.1
chord ring maintenance	11.1	1.2	40.5	52.8
database chain replication	36.3	0.1	24.4	58.8
decentralized lock	2.1	0.1	7.2	9.4
distributed lock	0.8	0.1	11.7	12.6
hashed sharding	0.6	0.1	0.4	1.1
leader election	0.8	0.1	1.0	1.9
learning switch	19.4	0.3	7.9	27.6
lock server	0.5	0.0	0.3	0.8
permissioned blockchain	3.5	0.1	1.3	4.9
Ricart-Agrawala	0.5	0.0	0.4	0.9
simple consensus	18.8	0.4	4.1	23.3
two-phase commit	0.5	0.1	1.3	1.9

Table 2: DistAI runtime breakdown in seconds for each protocol.

ically weaken the invariant set to reach a correct solution. In contrast, the use of a blackbox procedure poses difficulty and triggers errors in I4. We should note DistAI solves permissioned blockchain for not one, but any valid implementation of the Byzantine broadcast protocol because it does not depend on or require its implementation, a key benefit of our approach.

DistAI also outperforms both I4 and FOL-IC3 in terms of the time required to infer the correct invariants. For I4, we report both the runtime for the final instance size on which the correct invariant is generated, as reported in [18], as well as the total runtime, which includes trying increasingly larger instance sizes that fail until the final instance size succeeds. Except for learning switch, DistAI is about the same or faster than I4 for all of the protocols solved by I4, up to an order of magnitude faster. The runtime comparisons between DistAI and I4 are conservative as they do not include the time required for concretization [18], a step required by I4 to manually introduce additional constraints to the protocol to limit the search space of the model checker. DistAI is also faster than FOL-IC3 for all but the two simplest protocols solved by FOL-IC3, in many cases by more than one to two orders of magnitude. This is because SMT queries are expensive and FOL-IC3 uses them extensively.

Table 1 also shows for DistAI the number of invariants identified for the correct invariant set, the total number of refinement steps required (i.e., the number of times Algorithm 3 is called), the total number of quantified variables of all domains in the final template used, and the maximum number of literals used for each protocol. Most protocols have a maximum number of literals of 2 or 3, and in two cases 4. This validates our key assumption that inductive invariants of distributed protocols should be human-readable and concise. DistAI uses invariant refinement to address missing cases during sampling for all but the five simplest protocols, Ricart-Agrawala, hashed sharding, leader election, lock server, and asynchronous lock server, in which no refinement is needed as the subsample set is complete and the correct invariant is learned without refinement.

Runtime breakdown. Table 2 provides a breakdown of the total runtime using DistAI for each protocol. One can see the bottleneck is either sampling or refinement, but not enumeration. Sampling is expensive when the argument space for actions is sparse because DistAI randomly selects arguments so it can end up trying many arguments that are invalid for each set of valid arguments, increasing the simulation runtime. This is the reason why sampling is most expensive for database chain replication, a protocol that guarantees serializability and atomicity for distributed databases. A transaction is split into subtransactions that operate sequentially on data that is sharded across multiple nodes. For one subtransaction to commit, it must operate on the correct node and satisfy a set of constraints (e.g., no uncommited previous writes). Most randomly selected subtransactions will not satisfy these constraints. As a result, sampling spends significant time finding valid arguments because most arguments are invalid.

Sampling is also expensive for learning switch, the only protocol for which DistAI is slower than I4. One reason is the argument space for actions is sparse, so it takes a while to find a data sample, but the other is because the subsample space is too large to explore. With learning switch, each node maintains a routing table that matches destination addresses to outbound ports (i.e., neighbors), and updates the table upon receiving new packets. It has a 3-ary single-domain relation $route\_tc(N_1,N_2,N_3)$, which means the routing trace from N_2 to N_1 includes N_3. Under template $\forall N_1\ N_2\ N_3$, this single relation yields 27 predicates ($route\_tc(N_1,N_1,N_1)$,$route(N_1,N_1,N_2)$,...). There are 60 predicates across all relations, meaning that each subsample is a 60-bit vector, so the candidate subsample space has size 2^{60}. Although valid subsamples are sparse, DistAI generates 33K subsamples before it cannot find anymore and terminates. This takes a while.

Refinement can be the dominant factor in performance, as is the case for chord ring maintenance, database chain replication, and distributed lock, but Table 2 shows that DistAI is successful overall at avoiding substantial SMT query costs as refinement runtime, which includes the cost of IVy checking the initial candidate invariants, is modest in most cases.

Figure 5 shows how sampling helps reduce the cost of refinement for the simple consensus protocol. DistAI has provable guarantees to find the correct invariants for any sample sizes, but if the number of samples is too small (e.g., 100 in Figure 5), it takes much longer due to many more SMT queries on refinement. Increasing the number of samples increases sampling time roughly linearly but decreases refinement time roughly exponentially. However, once a minimum threshold of samples is met, it becomes more of an even tradeoff. Sampling can be faster by obtaining fewer samples, but because more corner cases are missing, the refinement process takes longer to "fix" the invariants through monotonic weakening. Conversely, more samples require more time to simulate the protocol, while the refinement process will be faster. The default sampling parameters, discussed in Section 3 and used for all experiments, resulted in 50K samples for the simple consensus protocol.

We also reran the protocol experiments with DistAI for other

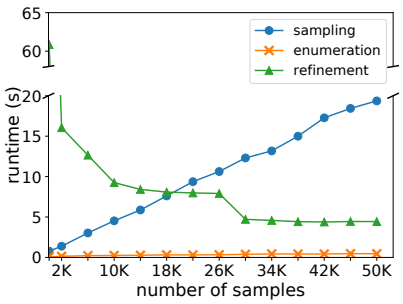


Figure 5: Runtime breakdown of DistAI on simple consensus.

sampling parameters, ranging from $MA = 25$ to $MA = 100$, and $SD = 2$ to $SD = 5$. In all cases, DistAI was able to solve the same 13 protocols with mostly similar runtimes and in the worst case, three times slower runtimes than the defaults. Detailed runtimes are omitted due to space constraints.

7 Related Work

Much work [10, 14, 16, 28, 31–33] has shown how to verify the correctness of distributed protocols and distributed systems given inductive invariants, but they rely on a user or external system to provide them. Various approaches have explored learning invariants for distributed protocols. Dinv [8] identifies and tracks critical variables in distributed systems, and then infers likely correct invariants over these variables with Daikon [2], a data-driven invariant learning tool. The invariants inferred using Dinv are not guaranteed to be valid and may not be inductive. Phase-PDR$^{\forall}$ [5] showed how to generate invariants for distributed protocols if users can provide phase structures. In contrast, DistAI is fully automated and does not require users to provide any additional knowledge about the protocol.

More recently, approaches have been developed for learning inductive invariants for distributed protocols. I4 [18] is the first. Its key idea is to use model checking on a finite protocol instance to generate candidate invariants. Although generated invariants by model checking some small instance always generalized in [18], this is not guaranteed. Our evaluation shows several protocols for which generalizing fails. I4 does not support existential quantifiers and also requires a manual concretization step. In contrast, DistAI is fully automated and provably guaranteed to learn inductive invariants without existential quantifiers. FOL-IC3 [11] can learn invariants with existential quantifiers by invoking an SMT solver to generate a candidate formula that can separate a positive and a negative example set. However, its heavy use of an SMT solvers slows its performance to the point that in practice, it fails to find inductive invariants for protocols that are efficiently handled by DistAI.

SWISS [9] is concurrent work that searches for invariants by template enumeration and checking candidate invariants with SMT queries. It does not do sampling, enumerating strongest possible invariants, or monotonic refinement, but does incorporate existential quantifiers in its invariant templates and uses counterexamples to prune the formula search space. In its reported evaluation, SWISS finds a correct existentially quantified invariant for Paxos, but fails or takes orders of magnitude more time than DistAI to find correct invariants for many other protocols listed in Table 1, despite being multithreaded.

Many automated invariant inference tools have been built for systems verification. Most of these tools focus on finding invariants in sequential programs with loops. Traditional methods use symbolic reasoning to infer invariants [6, 12], while recently data-driven methods using execution traces and/or counterexamples have shown promise. Guess-and-check, Numinv, and G-CLN recast invariant inference as a curve-fitting task on execution traces, and learn loop invariants represented by polynomials of program variables [20, 27, 29, 34]. ICE-DT and LoopInvGen (PIE) apply decision tree learning and PAC learning on counterexamples and iteratively refine the invariants [7, 21, 22]. FreqHorn exploits both syntax and data in its inference tool and learns $\forall$-quantified array invariants [3, 4]. Recently, data-driven invariant inference has been used in other domains, such as solving CHC clauses [35] and proving properties on inductive algebraic data types [19]. None of these methods consider nondeterminism in concurrent or distributed settings, thus they cannot be directly applied to distributed protocols.

8 Conclusions

DistAI is a fully automated, data-driven methodology for learning inductive invariants for distributed protocols. DistAI uses data samples from protocol simulation to enumerate the strongest possible set of candidate invariants, then feeds them to an SMT solver to check if adding them to the safety property is inductive. If any invariants fail, DistAI refines them by monotonically weakening the invariant set and tries again with the solver until it eventually succeeds. Starting with small invariant formulas and strongest possible invariants based on data from protocol simulation avoids large and frequent SMT queries, improving performance. Starting with strongest possible invariants makes refinement via monotonic weakening possible, enabling DistAI to provably guarantee that it will learn the correct inductive invariant set without existential quantifiers in finite time. Our evaluation shows that DistAI successfully learns inductive invariants for real distributed protocols and outperforms alternative methods, solving almost 50% more protocols and doing so up to one to two orders of magnitude faster.

Acknowledgments

Manos Kapritsos provided helpful comments on earlier paper drafts. This work was supported in part by an Amazon Research Award, a Guggenheim Fellowship, an NDSEG Fellowship, DARPA contract N6600121C4018, an NSF CAREER award, and NSF grants CNS-2052947 and CCF-1918400.

References

[1] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pages 193–206, August 2015.

[2] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, December 2007.

[3] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained Horn clauses using syntax and data. In *Proceedings of the 18th Conference on Formal Methods in Computer Aided Design (FMCAD '18)*, pages 1–9, October 2018.

[4] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, pages 259–277, July 2019.

[5] Yotam MY Feldman, James R Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, pages 405–425, July 2019.

[6] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, pages 813–829, July 2013.

[7] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, page 499–512, January 2016.

[8] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 1149–1159, May 2018.

[9] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proccedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, pages 115–131, April 2021.

[10] 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)*, pages 1–17, October 2015.

[11] 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 '20)*, page 703–717, September 2020.

[12] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 328–343, November 2010.

[13] LESLIE LAMPORT. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[14] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of International Symposium on Distributed Computing (DISC '11)*, pages 211–224, 2011.

[15] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, page 357–370, January 2016.

[17] Kuan-Ching Li, Xiaofeng Chen, Hai Jiang, and Elisa Bertino. *Essentials of Blockchain Technology*. CRC Press, 2019.

[18] 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 Symposium on Operating Systems Principles (SOSP '19)*, pages 370–384, October 2019.

[19] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 1–15, June 2020.

[20] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 11th*

Joint Meeting on Foundations of Software Engineering (FSE '17), pages 605–615, August 2017.

[21] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, page 42–56, June 2016.

[22] Saswat Padhi, Rahul Sharma, and Todd Millstein. Loop-invgen: A loop invariant generator based on precondition inference. *arXiv preprint arXiv:1707.02029v4*, October 2019.

[23] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 1, pages 1–31, October 2017.

[24] 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 '16)*, pages 614–630, June 2016.

[25] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*, pages 460–465, October 2020.

[26] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[27] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *Proceedings of 8th International Conference on Learning Representations (ICLR '20)*, March 2020.

[28] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *Proceedings of the ACM on Programming Languages (POPL)*, volume 2, pages 1–30, December 2018.

[29] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, pages 574–592, March 2013.

[30] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P '17)*, pages 521–538, May 2017.

[31] 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 '18)*, page 662–677, June 2018.

[32] 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)*, pages 357–368, June 2015.

[33] 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 *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CCP '16)*, pages 154–165, January 2016.

[34] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 106–120, June 2020.

[35] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 707–721, June 2018.

A Artifact Appendix

Abstract

An accompanying artifact includes all DistAI source code as well a docker image. Instructions are provided to reproduce the results in Table 1, Table 2, and Figure 5. The artifact can also be used to learn inductive invariants for other distributed protocols written in IVy.

Scope

The docker image can reproduce Table 1, Table 2, and Figure 5. The file `https://github.com/VeriGu/DistAI/blob/master/docker_usage.md` provides instructions to set up and use the docker. Alternatively, one can build DistAI from source, and reproduce the DistAI results in Table 1, Table 2,

and Figure 5. The README file (https://github.com/VeriGu/DistAI/blob/master/README.md) describes how to use DistAI to learn inductive invariants for other distributed protocols written in IVy.

Contents

The README file describes the structure of the artifact. The `src-py` and `src-c` directories include the Python portion and C++ portion of the source code. The `benchmarks` directory includes IVy specifications for the 14 protocols used in the evaluation.

Hosting

The artifact is hosted on GitHub in the repository https://github.com/VeriGu/DistAI. Future updates will be pushed to the master branch, and we encourage you to use the latest version available.

Requirements

The docker image has all dependencies installed. The installation guide (https://github.com/VeriGu/DistAI/blob/master/install.md) provides instructions to build DistAI from source. Note that IVy only works on Python 2, while the source code of DistAI is written in Python 3 and C++. The artifact has been tested on Ubuntu 20.04.1 LTS with ms-ivy 1.7.0, Python 2.7.18, and Python 3.8.5.

GoJournal: a verified, concurrent, crash-safe journaling system

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

Mark Theng
MIT CSAIL

Ralf Jung
MPI-SWS

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Abstract

The main contribution of this paper is GoJournal, a verified, concurrent journaling system that provides atomicity for storage applications, together with Perennial 2.0, a framework for formally specifying and verifying concurrent crash-safe systems. GoJournal's goal is to bring the advantages of journaling for code to specs and proofs. Perennial 2.0 makes this possible by introducing several techniques to formalize GoJournal's specification and to manage the complexity in the proof of GoJournal's implementation. *Lifting predicates* and *crash framing* make the specification easy to use for developers, and *logically atomic crash specifications* allow for modular reasoning in GoJournal, making the proof tractable despite complex concurrency and crash interleavings.

GoJournal is implemented in Go, and Perennial is implemented in the Coq proof assistant. While verifying GoJournal, we found one serious concurrency bug, even though GoJournal has many unit tests. We built a functional NFSv3 server, called GoNFS, to use GoJournal. Performance experiments show that GoNFS provides similar performance (e.g., at least 90% throughput across several benchmarks on an NVMe disk) to Linux's NFS server exporting an ext4 file system, suggesting that GoJournal is a competitive journaling system. We also verified a simple NFS server using GoJournal's specs, which confirms that they are helpful for application verification: a significant part of the proof doesn't have to consider concurrency and crashes.

1 Introduction

Storage systems, such as file systems, need to be carefully structured to not lose persistent user data, even in the face of application and whole-system crashes. They often achieve this *crash safety* property by delegating writing to storage to a *journaling system*, which exposes an API for executing an operation such that its writes appear on disk atomically. The journaling system simplifies implementing the storage system's logic: to atomically modify a set of objects, the file system simply writes to them one at a time within a single journal operation. The result is that each storage operation is atomic with respect to crashes.

While a journaling system exposes a simple API, its implementation must address crash safety and also be concurrent for good performance. Maintaining correctness in the presence of both concurrency and crashes is challenging. For example, in pursuit of performance, journaling systems often avoid holding locks while performing I/O, but reasoning about the correctness of such optimizations requires considering what happens if one thread's disk reads interleave with another thread's disk writes, and what happens when the system crashes anywhere during that interleaving.

This paper presents GoJournal, a Go package that provides the first formally verified concurrent journaling system. To verify GoJournal, we developed Perennial 2.0, an extension to the Perennial [5] framework with several features designed to enable modular reasoning about concurrent, crash-safe systems. In this work we set a goal of giving GoJournal a specification that *reflects the simplicity of using a journal for crash atomicity*. GoJournal can be used by an application like a file system or a key-value store. As long as the application follows a locking discipline for its on-disk state, such as per-file locks for a file system, proving the correctness and crash-safety of that implementation on top of GoJournal should involve largely *sequential* reasoning, despite the fact that the application has multiple concurrent threads and can crash at any time.

Realizing this goal raises two challenges: specifying GoJournal in a way that makes application reasoning sequential, and proving GoJournal's implementation correct. The specification makes reasoning about an operation sequential with a *lifting* interface where the proof has an abstraction of a "checked out" private fragment of the disk that the operation appears to synchronously modify. At commit time the private fragment is "checked in", at which point it is durable and can be exposed to other threads. The journal guarantees the operation is atomic by delaying all writes to commit time, so the developer should not need to explicitly reason about crash safety until commit time. Perennial 2.0 supports a new technique called *crash framing* to formalize the intuition that during an operation the developer need not explicitly consider crash safety.

The second challenge lies in proving GoJournal itself. This is difficult because we desire modularity to make the system's proof tractable, which requires giving suitable specifications to the internal interfaces of the system. While the user-visible interface of GoJournal is simple, the internal interfaces of a high-performance journaling system are hard to specify and fit together. To address this challenge, Perennial 2.0 contributes *logically atomic crash specifications* which enable natural specifications of system layers in terms of a transition system with atomic transitions for the public methods. These specifi-

cations include a *crash transition* to describe what happens to the state of a layer during a crash. Such specifications make it possible to build upper layers of the system on top without worrying about implementation details of how atomic transitions are achieved. This separation of concerns lines up with the modularity in the implementation; the proof layers divide up the reasoning along the same lines that the code divides up functionality among Go packages.

To test the performance and completeness of GoJournal, we built GoNFS, a functional (but unverified) NFSv3 server that can be mounted through the Linux NFS client. GoNFS imports GoJournal and uses it to achieve crash consistency for NFS operations. We focus on NFSv3 because it is widely used in practice, its performance matters for applications, and it has a crash-safety and correctness specification in the form of RFC 1813 [2]. The crash-safety properties are advanced; for example, the protocol supports unstable writes which let the implementation delay flushing them to disk.

On a combination of microbenchmarks and a software-development workload, GoNFS achieves at least 90% of the throughput of Linux's in-kernel NFS server exporting ext4 running on either a RAM disk or fast NVMe storage. On slower SSD storage without using unstable writes GoNFS gets 20% of Linux's throughput due to inefficient I/O. GoJournal's concurrency is crucial to performance: the throughput of GoNFS scales with the number of clients, but if GoJournal is modified to execute sequentially (as in previous verified storage systems), even with 20 clients GoNFS achieves only double the throughput of a single client.

To demonstrate that GoJournal's specifications enable effective verification of client applications, we implemented and verified a simplified NFS server, which we call SimpleNFS, covering the core operations, such as READ, WRITE, GETATTR, and SETATTR (which can shrink and grow a file). By using GoJournal's specifications, the proof for SimpleNFS largely involves crash-free reasoning (only 44 lines of code, out of a total of 462, require explicit reasoning about crashes). This translates into a lower proof overhead: SimpleNFS requires 3,749 lines of proof for 462 lines of Go code. GoJournal itself requires 25,797 lines of proof for 1,345 lines of Go code.

The contributions of this paper are (1) GoJournal, a concurrent journaling system with a machine-checked proof of correctness and crash-safety; (2) the Perennial 2.0 framework, with extensions to the original Perennial framework that enable modularity and crash-free reasoning on top of GoJournal; and (3) SimpleNFS, a verified core of an NFSv3 file server built on top of GoJournal.

Although GoJournal is advanced enough to support a high-performance NFS server, it has some limitations. GoJournal's internals (code and proof) support deferred durability, but for simplicity, GoJournal's top-level specification requires applications to immediately flush committed journal operations, which is sufficient to prove SimpleNFS. GoJournal is also less general than JBD2 (e.g., GoJournal does not support floating commit blocks), and less general than database transaction systems (e.g., GoJournal does not support undoing journaled operations). While GoJournal provides atomic updates for crash consistency, it does not implement automatic concurrency control. Objects accessed by a journal operation cannot be concurrently accessed by another thread. GoJournal provides a verified library for locking objects tracked by the journal, which clients can use to implement concurrency control.

2 Related work

To the best of our knowledge, GoJournal is the first verified concurrent, crash-safe journaling system. The verification of GoJournal builds on a large body of previous work, as described in the rest of this section.

2.1 Perennial 2.0 vs Perennial 1.0

The verification approach we take is based on a new version of our earlier Perennial [5] framework, so we draw a contrast between the two here. The new implementation is conceptually similar in that it supports reasoning about concurrency and crash-safety, it is implemented on top of the Iris [17, 18] concurrency verification system, and it uses Goose [6] to enable verification of Go programs by translating them into a model in Perennial 2.0. However, to make verification of GoJournal feasible, we had to re-write many core parts of the framework. To clarify which framework is being referenced we will write Perennial 1.0 for the original framework and Perennial 2.0 for the new one in this section, in order to highlight the new features Perennial 2.0 supports. The rest of the paper generally refers only to Perennial 2.0.

Some of Perennial 2.0's features are needed to support the GoJournal top-level specification and enable verification on top of this interface. The reason this problem is complicated is because the journal does not make operations automatically atomic but requires the caller to correctly manage ownership, and Perennial 1.0's refinement specifications do not give a good way to talk about ownership. The top-level specification of GoJournal relies on *crash framing* (§5.5) and *crash-aware locks* (§5.4) to enable application proofs that reason about ownership of durable data.

Perennial 2.0 also scales to a larger system than the mail server verified in Perennial 1.0. One of the challenges with the larger system is that it has many internal layers that need their own specifications, so that the proof can be carried out modularly. Normally a separation logic or refinement-based specification would be sufficient, but we need internal specifications that capture the crash and concurrent behavior of each internal library. To that end Perennial 2.0 incorporates a new specification style which adds *crash atomicity* to the logically atomic specification styles developed in earlier work [10, 15, 27]. Modularity in the proof was necessary to scale verification to all of GoJournal's performance optimizations and concurrency.

Method	Description	Spec
`func Begin() *Op`	Start operation	§5.2
`func (*Op) ReadBuf(addr Addr, sz uint64) *Buf`	Read a buffer	§5.3
`func (*Buf) SetDirty()`	Mark a buffer as modified	§5.3
`func (*Op) OverWrite(a Addr, sz uint64, data []byte)`	Write without reading	§5.3
`func (*Op) Commit(wait bool) bool`	Commit by appending to in-memory log. If `wait=true`, also wait until changes are on disk.	§5.6
`func Flush() bool`	Flush in-memory log	
`func (*Lockmap) Acquire(i uint64)`	Acquire `i`th lock	§5.4
`func (*Lockmap) Release(i uint64)`	Release `i`th lock	§5.4

Figure 1: GoJournal interface and API for lockmap. Not shown are auxiliary interfaces for initialization; checking operation size; etc.

At the same time, GoJournal's specification allows the proof of SimpleNFS to mostly avoid reasoning about crashes.

2.2 Related verification frameworks

Crash-safe systems. Any crash-safe system must reason about the possible states after a crash, and several prior works have formalized this in different ways for *sequential* crash-safe systems. FSCQ [7, 8] uses Crash Hoare Logic (CHL) to specify crash behavior through a crash condition, which describes the state of a system if a crash happens during execution of a function. Alternatively, a number of systems verify crash safety using refinement reasoning [4, 12, 14, 26], but none support the combination of concurrency and crash-safety.

Although they are not concurrent, some of these systems address other aspects of performant storage systems that are not found in GoJournal. DFSCQ [7] verifies a high-performance file system built on top of a logging system with asynchronous disks and log-bypass writes, which are challenging optimizations that GoJournal does not support. VeriBetrKV [14] verifies a key-value store based on B$^\epsilon$ trees, a data structure that also underlies BetrFS [16]. GoJournal and SimpleNFS use simple data structures; the challenge lies in accounting for concurrent accesses.

Concurrent systems. In addition to specifying behavior at intermediate crash points, Perennial 2.0's specifications describe the atomic commit points of concurrent operations. A range of verification techniques have been used to address this kind of challenge in concurrent systems. AtomFS [29] uses a framework called CRL-H (concurrent relational logic with helpers) to verify a concurrent in-memory file system implemented in C. Refinement-based systems such as CSPEC [3], Armada [23], and Concurrent CertiKOS [13] typically prove that a function implements an atomic operation at a more abstract layer. However, in GoJournal, many internal APIs provide operations that are only atomic if the caller owns some data. This kind of conditional atomicity is easy to express in Perennial 2.0 using separation logic, but hard to express as a precondition in a transition system.

Concurrent, crash-safe reasoning. Program logics other than Perennial have been developed for formal reasoning about concurrent, crash-safe systems. Fault-Tolerant Concurrent Separation Logic (FTCSL) [24] extends the Views [11] concurrency logic to incorporate crash-safety. POG [25] is a program logic for reasoning about the interaction of x86-TSO weak-memory consistency and non-volatile memory. Neither logic has a mechanism for modular proofs of layers, which we found essential to scale verification to a system of GoJournal's complexity. Both are restricted to pen-and-paper proofs, whereas both Perennial 1.0 and 2.0 have machine-checked proofs.

A specification called the Push/Pull model of transactions [19] is similar to the *lifting* technique in the journal system's specification (§5.2) — the core problem addressed is that a journal operation atomically modifies a small number of objects, but other objects can change between the start of the operation and when it commits. The Push/Pull model also discusses reasoning on top of the specification, using Lipton's reduction [22] rather than separation-logic ownership to handle concurrency. However that work is about on-paper specifications and proofs, while we also prove an implementation meets our specification and proved SimpleNFS on top.

3 System design

The verified artifact of this paper is GoJournal, a Go package that gives clients an abstraction of a disk with crash-safe writes. This section aims to convey what the journal is, why its implementation deserves verification, and how systems can be built using it. First, §3.1 explains how a developer uses GoJournal to write a concurrent storage system, informally laying out what the package's requirements and guarantees are. Then, §3.2 explains how the journal is implemented.

3.1 Programming with GoJournal

Developers use the journal to turn several storage operations into an atomic journal operation that commits to disk using the GoJournal interface listed in Figure 1. `Begin` starts a journal operation, returning a `*Op` object, which keeps track of the objects read or written in the operation. An object is addressed by the `Addr` struct, which names a block address and bit offset within the block. SimpleNFS has objects for on-disk blocks

```
1  func NFS3_WRITE(args WRITE3args) WRITE3res {
2    inum := fh2ino(args.File)
3    if !validInum(inum) {
4      return WRITE3res{Status: NFS3ERR_INVAL}
5    }
6    inode_locks.Acquire(inum)
7    reply := NFS3_WRITE_locked(args, inum)
8    inode_locks.Release(inum)
9    return reply
10 }
11
12 func NFS3_WRITE_locked(args WRITE3args,
13     inum Inum) (reply WRITE3res) {
14   op := Begin()
15   if !NFS3_WRITE_op(op, args, inum, &reply) {
16     return
17   }
18   if txn.Commit(true) {
19     reply.Status = NFS3_OK
20   } else {
21     reply.Status = NFS3ERR_SERVERFAULT
22   }
23   return
24 }
```

Figure 2: RPC handler for NFS WRITE showing locking and committing a journal operation.

and on-disk inodes, while the complete NFS server also uses objects for individual allocator bits.

`ReadBuf` reads an object into an in-memory `*Buf` struct, returning the latest value of the object within this journal operation. If the operation hasn't read the object yet, it reads the latest value from disk (or from a recently committed operation). A journal operation can modify the returned buffer in place and then mark the buffer as dirty with `SetDirty`. To overwrite an object without reading it the application can call `OverWrite`. When the operation is fully prepared, the application commits it atomically using `Commit`; setting `wait=true` additionally forces the journal to flush the results to disk. In either case the writes in the operation appear together on disk or not at all even if the system crashes. The application can also call `Flush` to make the journal persist several committed but unstable operations to disk.

While GoJournal provides crash-safe atomic updates to disk with this interface, it is the developer's job to provide concurrency control to prevent concurrent operations from manipulating the same on-disk objects. In a file system a common strategy for concurrency control is to use a per-file lock that protects both the file metadata and any data blocks associated with the file, and this strategy is the one used by GoNFS and SimpleNFS. To make it easier for a file system to maintain these locks, GoJournal includes a lockmap library that behaves as if it were a large array of locks but with a more memory-efficient implementation; the Guava Striped documentation describes the idea well [1].

Figure 2 and Figure 3 show how SimpleNFS uses the GoJournal API and the lockmap. The server runs each NFS request in a separate Go thread running a single journal operation. Figure 2 shows the RPC handler for an NFS WRITE RPC, in particular acquiring a per-inode lock (lines 6 and 8) and preparing an operation starting at line 14.

```
1  func NFS3_WRITE_op(op *Op, args WRITE3args,
2      inum Inum, reply *WRITE3res) bool {
3    ip := ReadInode(op, inum)
4    count, ok := ip.Write(op, args.Offset,
5          args.Count, args.Data)
6    ... // set count and status
7  }
8
9  func (ip *Inode) Write(op *Op, off uint64,
10     count uint64, data []byte) (uint64, bool) {
11   if count != uint64(len(data)) ||
12      util.SumOverflows(off, count) ||
13      off+count > disk.BlockSize ||
14      off > ip.Size {
15     return 0, false
16   }
17
18   buf := op.ReadBuf(block2addr(ip.Data),
19         NBITBLOCK)
20   copy(buf.Data[off:], data)
21   buf.SetDirty()
22   if off+count > ip.Size {
23     ip.Size = off + count
24     ip.WriteInode(op)
25   }
26   return count, true
27 }
28
29 func (ip *Inode) WriteInode(op *Op) {
30   op.OverWrite(inum2Addr(ip.Inum),
31     INODESZ*8, ip.Encode())
32 }
```

Figure 3: `NFS3_WRITE_op` prepares a journal operation `op` for the WRITE RPC.

The handler is split into several nested functions for ease of verification. Figure 3 shows how the WRITE RPC's journal operation of type `*Op` is prepared. For example, lines 18–21 read and modify the block data, while line 30 modifies the inode. The combination of per-file locking and using the journal for disk access frees the developer from thinking about either concurrency or crashes during the entire `NFS3_WRITE_op` code, which we will show is also the case in the proof using Perennial's specification techniques in §5.

For ease of explanation, SimpleNFS has the limitation that each file consists of only one block, but note that WRITE modifies two on-disk objects: the inode and the block owned by the file; the two together must be written atomically, which the proof shows using the GoJournal specification. Also note that there is no explicit locking of blocks; ownership of the data block is implicit because a block can belong to only one file.

Layer	Description
JRNL	In-memory object operations
OBJ	Journaling sub-block writes
WAL	Whole-block write-ahead logging
CIRCULAR	Circular log structure

Figure 4: GoJournal layers.

3.2 GoJournal implementation

The journal is structured into several layers, as shown in Figure 4. At a high level, the system is split into two halves. The low-level half is a write-ahead log that behaves like a disk with an atomic multiwrite operation, which appears to update multiple disk blocks simultaneously even if the system crashes. The upper half, called the object system, allows callers to perform read and write operations on objects smaller than a block ("sub-block" objects). Writes are buffered in memory until the caller chooses to commit, at which point a multiwrite to the write-ahead log commits the writes to disk.

The write-ahead log is implemented by organizing the disk into a small, fixed-size circular buffer and a remaining data region. Data is first atomically *logged* to the circular buffer and then eventually *installed* to the data region, to free space in the circular buffer. Reads first go through the circular buffer (which is cached for efficiency) and then access the data region.

The object system maintains a list of buffers of data read or written by each journal operation. Reads first check the write-ahead log's cache since they must observe committed operations. To commit, the object layer gathers all the dirty buffers and submits them as a multiwrite to the write-ahead log. To allow reading and writing objects that are smaller than a block, the object layer assembles these into block writes by doing a read-modify-write sequence.

Because disk writes are slow, for good performance the journal executes many tasks in parallel. Committing new journal operations in memory, logging operations from memory to disk, waiting for operations to be made durable, and installing logged writes all happen concurrently. Concurrency ensures that in-memory operations need not wait for any in-flight disk reads or writes, and that many disk reads and writes can happen at the same time. Finally, to reduce the number of disk writes, the write-ahead log implements two optimizations. Multiwrites are combined and written together ("group commit"), and if they update the same disk block multiple times, only the most recent update of that disk block is written to the log ("absorption"). Concurrency makes these optimizations useful even for synchronous operations, which can be committed together and absorbed if they are issued concurrently.

Concurrency in the write-ahead log complicates not just its internals but also reasoning about the multiwrite abstraction built on top. One difficulty is that reading requires checking the log's in-memory cache and then falling back to the disk, but the disk read happens without a lock. If a multiwrite commits after the read misses in the cache, then the disk read will not observe the latest value. The write-ahead log specification specifies that reading the installed value might return an old view of the disk, and the object layer can handle this weak specification with an invariant that guarantees the object being read has not been modified since that old view.

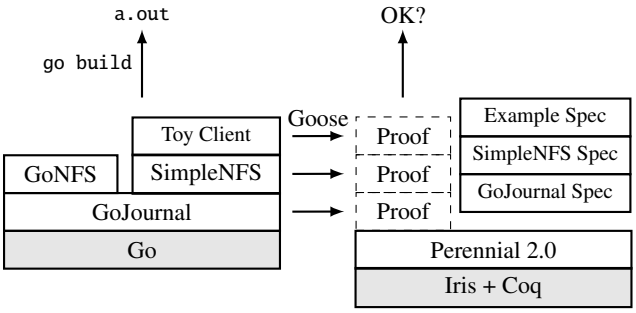


Figure 5: Overview of Perennial, GoJournal, SimpleNFS, and GoNFS.

The object layer implements sub-block access on top of the write-ahead log's block-level multiwrites. Objects accessed by an operation must be locked, so supporting fine-grained access is necessary to allow operations to run concurrently even if they happen to access the same disk block. For example, a file system might pack inodes into a block, and locking an inode should not prevent concurrent operations for other inodes in the same block. The object-layer implementation is able to execute reads and writes during an operation without any additional locks, but something more is needed to commit. Imagine a situation where between reading some disk block and writing it an unrelated object was modified in the same block; committing the modified block would overwrite the concurrent modification, losing data. The code addresses this with a global commit lock that prevents concurrent modifications while reading the blocks to be written.

4 Verification overview

Figure 5 gives an overview of how GoJournal and systems building on it are verified using Perennial. On the left of the figure is the executable code, which is written in Go. On top of GoJournal, we have implemented two NFS servers to evaluate GoJournal along different dimensions. GoNFS is a functional NFS server that is sufficient to run real applications, which we use to assess GoJournal's scalability and performance. Meanwhile, SimpleNFS is a verified, core subset of GoNFS's functionality, which evaluates the usability of GoJournal's specs for building verified systems on top of it.

On the right side of the figure is the verification stack. The verification builds on the Perennial 2.0 framework, which is itself implemented in the Iris framework in the Coq proof assistant. To reason about executable code, a tool called Goose translates a Go implementation into a model that we can prove specifications about in Perennial. Perennial provides a model of execution for Go code that incorporates crash-safety and

concurrency, which includes a model of the disk (with atomic, synchronous reads and writes of 4KB sectors) as well as a model of crashes and recovery (crashes at arbitrary points during execution, and jumping to specific boot code for recovery after a crash).

GoJournal's top-level specification describes its API in terms of an extension of concurrent separation logic, with pre-, post-, and crash conditions. These specifications capture the behavior of individual Go functions: if the function is run in a state satisfying its precondition, then the final state will satisfy the postcondition, and if the system crashes the state will satisfy the crash condition. The specification for the journaling API is described in detail in §5. We demonstrate the usefulness of this specification by proving correctness of the SimpleNFS server using logically atomic crash specifications (§6.2). The top-level theorem for SimpleNFS states that its RPCs atomically follow transitions of a state machine formalizing the NFSv3 protocol (based on RFC 1813 [2]).

As described in FSCQ and Argosy [4, 8], crash conditions can be used to reason about recovery procedures, even crashes during recovery. A recovery procedure can safely be re-run after a crash if its specification is *idempotent*: its crash condition should imply its precondition. As an end-to-end check of the crash specs in SimpleNFS and GoJournal, showing they support recovery correctly, we prove an idempotent specification for a toy example client on top of SimpleNFS, establishing that it can successfully execute even if SimpleNFS crashes and recovers an arbitrary number of times.

The proof of GoJournal's specification depends on a number of assumptions. We assume that the disk writes 4KB blocks atomically, even on crash, and assume that the code executes according to the Perennial model generated by Goose. The specification relies on the caller to provide concurrency control; the proof of SimpleNFS checks that locking is performed correctly, but GoNFS is unverified and we trust that its concurrency control is correct in order to make operations atomic (though this does not say they correctly implement the NFS specification).

5 Specifying GoJournal

The goal of GoJournal's specification is to support convenient reasoning about atomic operations, like the NFS WRITE implementation in Figure 2 and Figure 3. In this section we walk through how the specification guarantees atomicity for the caller without forcing the caller to do much application-specific reasoning about concurrency or crashes.

The key to this specification is tracking resources, like the disk blocks making up a file, as they flow through the steps of the proof. We start by reviewing how separation logics like Perennial represent these resources, and how specifications in the logic track logical *ownership* of resources (§5.1). The specification for GoJournal introduces resources that distinguish between a journal operation's local view of an object and the durable, on-disk representation; obtaining either resource requires the caller to use correct synchronization, as required by the journal's implementation. *Lifting* provides a way to translate a locked object from its on-disk view to a local view within the operation (§5.2). While preparing a journal operation, reads and writes modify the local view (§5.3). Finally, committing an operation writes its updates to disk, so the specification asserts that the local view becomes a view over durable state.

To take full advantage of the durable and operation-local views of journal objects, the proof of WRITE uses two new techniques introduced by Perennial 2.0: crash-aware locking (§5.4) and crash framing (§5.5). With these techniques, the proof of `NFS3_WRITE_op` uses entirely sequential reasoning for preparing the journal operation, even though concurrent operations might write to disk and its disk writes are buffered rather than synchronous. Finally, §5.7 summarizes how the proof techniques combine to prove correctness and crash-safety for the NFS WRITE example.

5.1 File representation

First, in both designing the code and writing the proof, the NFS server must establish a disk layout to arrange its data in terms of disk objects. The disk layout is expressed using a separation-logic *representation invariant*, a predicate which connects the logical (specification-level) contents of files to the objects (inodes and blocks) that encode those files.

Representation invariants over the state of the journal use a "points-to predicate" $a \mapsto o$, which serves two purposes: it asserts that the address a (of type `Addr`) contains an object o (which is represented by the `*Buf` type in the API), and it represents exclusive *ownership* over the address a. When a thread has $a \mapsto o$ in its precondition, ownership allows the proof to assume that the value at address a does not change until the thread gives up ownership, and that it will not be read by other threads. Locks help threads transfer ownership so a thread only retains exclusive ownership during a critical section.

The SimpleNFS proof connects each file to its representation with the following representation invariant:

$$\begin{aligned}&\texttt{file\_rep}(i, data) \triangleq \exists meta, \exists blk,\\ &\quad i \mapsto meta * meta.\texttt{blkno} \mapsto blk \wedge\\ &\quad meta.\texttt{size} = \texttt{length}(data) \wedge \texttt{prefix}(data, blk)\end{aligned}$$

Informally the representation invariant says the file i with logical contents $data$ is represented by some metadata $meta$ stored at the inode number i and a data block at $meta$.`blkno`. It then says the file's bytes are a $meta$.`size`-length prefix of the data block.

This definition uses the separating conjunction $P * Q$ (pronounced "P and separately Q"), which says that two predicates hold over disjoint state. For example, this asserts the inode and its data block are stored separately. To initialize the system the caller must prove that the `file_rep` predicates hold separately

for each file, that is, $\texttt{file\_rep}(1, data_1) * \texttt{file\_rep}(2, data_2) * \cdots$. Here the separating conjunction asserts files are represented disjointly, so that when a thread modifies one file it is guaranteed not to affect data in other files.

5.2 Lifting

The key idea of GoJournal's specification is to consider two view of the disk: a conceptual in-memory view that a buffered journal operation observes, as well as an on-disk view that reflects what would be on disk after a crash. Parts of both views are constantly changing as other threads commit operations concurrently, so we use separation logic to define a local view that contains only objects locked by and involved in a journal operation. Because the journal operation logically owns these objects, the caller can use sequential reasoning—disk objects have the same value throughout—and can commit all of the objects written in the operation at the end without fear of interfering with concurrent journal operations. The specification makes this informal reasoning concrete using *lifting*, which we use to refer to this strategy of transferring ownership to and from the on-going operation.

To do anything with the journal, a thread must first `Begin` an atomic operation:

$$\{\text{True}\}$$
$$\quad \texttt{Begin}()$$
$$\{\textbf{ret}\ op, \texttt{is\_op}(op) * \texttt{durable\_pred}(op, \text{True})\}$$

The specification above is a Hoare triple for the `Begin()` function. It says that executing `Begin()` starting with its precondition (in this case True) will run without errors and if it terminates it will return *op* along with the postcondition, namely $\texttt{is\_op}(op) * \texttt{durable\_pred}(op, \text{True})$. The `is_op` part of the post-condition simply says that *op* is a valid `*Op` object. The $\texttt{durable\_pred}(op, \text{True})$ clause is what tracks the on-disk data "underneath" a journal operation, which would be left behind if the operation aborted; since the operation starts out with an empty local view, it starts out with no on-disk footprint, written as True.

The different views of a journal operation are tracked using *ghost state* in Iris. Ghost state is separate from the physical state of the program—the contents of memory and disk—and is only manipulated by the proof. The journaling system's proof introduces ghost state for durable state of the system, including an $a \mapsto_d o$ predicate for ownership over individual objects. Note that an object is expressed through ghost state because the block holding the object might be located in the on-disk log or in the data region, and ownership of an object says nothing about other objects in the same disk block.

The proof also introduces a similar $a \mapsto_{op} o$ predicate for the local view of operation `op`, and it is this ownership that is needed for reads and writes. A caller obtains these predicates with a logical operation we call *lifting* that converts ownership of $a \mapsto_d o$ into $a \mapsto_{op} o$, granting the ability to read and write.

To make it easier to work with lifting, the specification allows lifting an entire *predicate P* and transforms all of its points-to facts simultaneously, which we denote this paper denotes by switching subscripts. For example, we re-use the definition `file_rep` from §5.1 for both a file laid out on disk and a file as owned by a journal operation, which we denote with $\texttt{file\_rep}_d$ and $\texttt{file\_rep}_{op}$ respectively. The specification for lifting a generic predicate *P* is:

$$\{P_d * \texttt{durable\_pred}(op, Q_d)\}$$
$$\quad noop$$
$$\{P_{op} * \texttt{durable\_pred}(op, P_d * Q_d)\}$$

Since lifting is purely logical (it only modifies ghost state), we write it as a Hoare triple for a no-op, much like how Dafny and F* lemmas are simply methods with pre- and post-conditions but no code [21: §12.2.3].[1] The outcome of lifting is to expand the memory covered by the journal operation to incorporate P_d. Observe that `durable_pred` is expanded to "snapshot" P_d, which tracks that if the operation were to abort or crash, the durable P_d that we started with would still hold. The on-disk values do not change over the course of a buffered journal operation (as expected, since these are in-memory writes). The key part of the postcondition, however, is P_{op}: the $a \mapsto_{op} o$ predicates within P_{op} (e.g., the $i \mapsto_{op} meta$ within $\texttt{file\_rep}_{op}(i, data)$) give the caller the right to read and write objects from within the operation, as we will see in §5.3.

5.3 Reads and writes

The specification for `OverWrite` describes the effect of writing to the local memory of a buffered journal operation:

$$\{\texttt{is\_op}(op) * a \mapsto_{op} o * \texttt{buf\_obj}(buf, o')\}$$
$$\quad op.\texttt{OverWrite}(a, buf)$$
$$\{\texttt{is\_op}(op) * a \mapsto_{op} o'\}$$

The precondition includes $\texttt{buf\_obj}(buf, o')$ to say that the in-memory buffer buf encodes the object to be written o'. The `is_op` predicate is both required and returned by the specification, which reflects the fact that `OverWrite` operates on the in-memory state covered by this predicate.

The specification for `ReadBuf` is more subtle. `ReadBuf` returns a buffer that the caller is allowed to modify in-place, which has the side-effect of updating the in-memory state of the ongoing journal operation, which will in turn be committed by `Commit`. Figure 3 shows an example, where lines 18–20 modify a read buffer in-place. The specification captures this

[1] In case the reader is already familiar with Iris, these Hoare triples represent what is usually called a "view shift" in Iris.

behavior as follows:

$$\begin{array}{l}\{\texttt{is\_op}(op) * a \mapsto_{op} o\} \\ \quad op.\texttt{ReadBuf}(a) \\ \left\{\begin{array}{ll} & \texttt{buf\_obj}(buf, o) * \\ \textbf{ret}\ buf, & (\forall o', \texttt{buf\_obj}(buf, o') \mathrel{-\!\!*} \\ & \quad \texttt{is\_op}(op) * a \mapsto_{op} o') \end{array}\right\}\end{array}$$

This states that, when `ReadBuf` finishes, it returns a buffer *buf* and two resources: $\texttt{buf\_obj}(buf, o)$ says the buffer has the old object o, while the second is a separating implication or *wand* $-\!\!*$. The wand says that if the caller modifies the buffer to produce $\texttt{buf\_obj}(buf, o')$ for some other data o' (or leaves it unchanged, picking $o' = o$), it can get back the $\texttt{is\_op}(op)$ predicate, along with a $a \mapsto_{op} o'$ fact indicating that a has been modified in-place to the new data o'.[2] The wand is just another resource that the caller can invoke at the right time in the proof (e.g., after the call to `SetDirty` in Figure 3 on line 21).

5.4 Crash-aware locking

As seen in Figure 2, the NFS server acquires a per-file lock (within the lockmap) to prevent concurrent access to the same disk object. Each lock logically protects both the file metadata stored in its inode and the data block pointed to by the inode. The usual specification for a lock in concurrent separation logic says that it protects some lock invariant, guaranteeing that this invariant holds upon acquiring the lock and conversely obliging the caller to prove the lock invariant to release. This invariant may claim ownership of resources which are then owned by clients during their critical section. The file i's lock in SimpleNFS protects roughly $\texttt{file\_rep}_d(i, data)$, where we write d to indicate the file is laid out on disk; we make the invariant more precise later when we connect it to crash safety.

This lock specification, however, is insufficient to prove that the SimpleNFS server maintains all relevant invariants when the system crashes. The specification makes no guarantees about the protected data during a critical section—however, a crash while the lock is held exposes any durable data that was protected by the lock. The lock specification fails to express that the lock holder should keep the durable data in a state that can be recovered from after a crash.

To solve this problem, Perennial 2.0 contributes a new specification for locks called *crash-aware lock specifications* that is useful for protecting durable data like $\texttt{file\_rep}_d$. We proved this specification both for ordinary locks (`*sync.Mutex` in Go) and for the stripes in the lockmap, but here we present just the lockmap version. With this specification, the proof associates not just a lock invariant but also a *crash obligation* $I_c(\texttt{i})$ to each file. Like the ordinary lock specification, acquiring the lock gives the caller access to the lock invariant $I(\texttt{i})$, but unlike that spec, this specification also obliges the caller to prove the crash obligation $I_c(\texttt{i})$ at every intermediate step. The proof enforces this using crash specifications: $\{P\}\ e\ \{Q\}\{Q_c\}$ is like a Hoare triple but it has an extra predicate Q_c, the crash condition, describing what holds if the system crashes during e's execution. When the caller wants to prove something about code that acquires a lock using the crash-aware specification, it must do so with $I_c(\texttt{i})$ in its crash condition for the critical section:

$$\begin{array}{l}\{P * I(\texttt{i})\}\ \texttt{f()}\ \{Q * I(\texttt{i})\}\{I_c(\texttt{i})\} \\ \vdash \{P\}\ \texttt{Acquire(i); f(); Release(i)}\ \{Q\}\end{array}$$

In exchange for the extra work of having to prove a crash specification, the crash-aware lock spec guarantees that the lock's crash obligation holds at crash time, ready to be used by new threads spawned following the crash.

One final subtlety in the specification is that Perennial distinguishes between the disk while running d_k and the new disk following a crash d_{k+1}, where k is a so-called *generation number*. This creates a distinction between the invariant protected by the lock (in generation k) and the crash obligation (in the next generation):

$$\begin{array}{l} I(i) \triangleq \exists data, \texttt{file\_rep}_{d_k}(i, data) \\ I_c(i) \triangleq \exists data, \texttt{file\_rep}_{d_{k+1}}(i, data)\end{array}$$

It is important that on crash the developer show `file_rep` holds in the post-crash generation d_{k+1}, because any ephemeral resources in the current generation do not survive to the next. Any in-memory state the system requires has to be reconstructed from only the durable state.

5.5 Crash framing

As we have seen, acquiring a crash-aware lock imposes that the crash obligation holds at every step until the crash lock is released. For example, the developer must show that the crash obligation $I_c(i)$ holds at every step of `NFS3_WRITE_locked`. However, much of the code for `NFS3_WRITE_locked` resides in `NFS3_WRITE_op`, which modifies only in-memory state. This presents an opportunity to simplify the proof: because no durable state is modified, the developer should not need to think about crashes at each individual step.

Perennial 2.0 formalizes this using the *crash framing* technique, expressed in the following rule:

$$\begin{array}{l}\{P\}\ \texttt{f()}\ \{Q\} \\ \vdash \{I_c * P\}\ \texttt{f()}\ \{I_c * Q\}\{I_c\}\end{array}$$

Informally, this rule says that if we currently own the crash condition I_c, we can temporarily "give up" access to that ownership when proving `f()`. In exchange, the crash condition is removed from our proof obligation: it is sufficient to prove a regular crash-free Hoare triple for `f()`. I_c is not available for the proof of `f()` (this is the "giving up" aspect of crash framing), but the proof can continue to use I_c after the call to `f()` returns.

[2] To simplify the presentation, we have omitted the obligation that forces the caller to call *buf*.`SetDirty`() before getting back `is_op`.

The proof of NFS3_WRITE gets access to $I(\texttt{i})$ by acquiring the ith lock, lifts the $\texttt{file\_rep}_{d_k}$ predicate into its buffered operation, and then immediately uses the crash framing rule to give up access to $\texttt{durable\_pred}(op, \texttt{file\_rep}_{d_k})$ and prove the crash condition for the duration of NFS3_WRITE_op (which only manipulates the in-memory $\texttt{file\_rep}_{op}$). The crash framing rule gives back the durable_pred predicate at the end of the operation, which is required to reason about commit.

5.6 Commit

The remainder of the proof after preparing $\texttt{file\_rep}_{op}$ with the new data is to reason about committing the operation with the new file. The code commits this operation using the following specification for Commit:

$$\begin{array}{l}\{Q_{op} * \texttt{is\_op}(op) * \texttt{durable\_pred}(op, P_d)\} \\ \quad op.\texttt{Commit}(\text{true}) \\ \{\textbf{ret}\ ok, \text{if } ok \text{ then } Q_d \text{ else } P_d\} \\ \{P_d \vee Q_d\}\end{array}$$

This specification nicely captures how Commit works: if we started with data P_d on disk, then modified it to Q_{op} in memory, then if Commit succeeds the new data Q_d is on disk. If Commit fails (which happens if the journal operation is too large to fit on disk) then the data reverts back to P_d. On crash either of these could happen, depending on when the crash occurs.[3]

The caller will sometimes start an operation and then abort it, say due to encountering an error. The API has no method for this because aborting is a purely logical operation that restores ownership of the on-disk objects:

$$\{\texttt{durable\_pred}(op, Q_d)\}\ noop\ \{Q_d\}$$

The Commit proof internally executes the same logical operation when the commit fails in order to return the original durable data.

5.7 Summary

The combination of above features mean the developer is mostly left with sequential crash-free reasoning about how each operation (for example, each NFS3 RPC implementation) transitions from the representation invariant in one state to another, following the transition system of the specification. We illustrate that proof flow using the NFS3_WRITE call in Figure 2 as an example.

First, the function starts a journal operation and acquires a lock on i. Then the proof requires some purely mechanical work to lift the lock invariant (§5.2) and frame the crash obligation (§5.5). Next, the developer proves the correctness of the sequential code. This proof does involve the bulk of the application code, but it requires neither worrying about concurrency (since reads and writes operate on the exclusive ownership of $a \mapsto_{op} o$) nor about crash safety (since crash framing has dismissed any crash obligations while reasoning about the in-memory operations on the *Op).

The sequential code must prove that the reads and writes with ReadBuf, SetDirty, and OverWrite transform $\texttt{file\_rep}_{op}(i, data)$ to produce $\texttt{file\_rep}_{op}(i, data')$, where $data'$ is the correct state of the file as described by the transition of the formalized NFS state machine for a write. The new file representation with contents $data'$ is the Q_{op} in the precondition to Commit's specification, while P_d is the old file with contents $data$ on disk (snapshotted while lifting).

If the system doesn't crash and Commit returns true, then the operation succeeds, producing a new file representation $\texttt{file\_rep}_{d_k}(i, data')$. If the operation fails (say due to not fitting in the log), then Commit returns the old representation invariant with contents $data$. On crash, either of these two is possible, but not some inconsistent combination of the two, guaranteeing crash atomicity.

The proof for NFS3_WRITE wraps up by releasing the lock. Whether or not Commit succeeds, we have a file with some contents: $\exists data, \texttt{file\_rep}_{d_k}(i, data)$; this is exactly the lock invariant $I(i)$ required to release the lock.

6 Verifying GoJournal

GoJournal consists of multiple layers, as described in §3.2. This section provides some highlights of the complexity involved in GoJournal's implementation, along with the proof techniques required to formally reason about that complexity.

6.1 Write-ahead logging (WAL)

The write-ahead log layer is responsible for updating multiple disk blocks (a multiwrite) atomically. Each multiwrite is a list of updates, where an update consists of a disk block number and the new data to write in that block. A background logger thread moves multiwrites from an in-memory buffer to an on-disk log. To make this atomic, the logger first writes the contents of a multiwrite in a log entry, and then updates a designated header block to indicate the entry is complete. If a crash happens before the header is updated, none of the multwrite's updates are applied; if a crash happens after the header update, the multiwrite will be applied during recovery. Meanwhile, an installer thread applies entries in the log to the disk, clearing space for new multiwrites. If a crash happens before the updates in an entry are fully installed, recovery installs the updates again from the on-disk log.

The write-ahead log implements two optimizations related to combining multiwrites. Two or more multiwrites can be *group committed* by logging them together, which still guarantees their atomicity. If multiwrites being committed together update the same block, the first update can be *absorbed* and replaced with the second. These optimizations trigger both for multiwrites that are committed without waiting for durability and also for concurrent, synchronous multiwrites.

[3] For $op.\texttt{Commit}(\text{false})$, which does not flush to disk right away, GoJournal provides a lower-level spec that allows expressing the more complex resulting crash condition.

Internal abstract state: logical log. To prove the write-ahead log layer correct, GoJournal represents the state of the write-ahead log as a logical list of multiwrites, as shown in Figure 6. Multiwrites before `memStart` have already been installed, and their log entries do not physically exist in memory or on disk. Multiwrites from `memStart` to `diskEnd` are already logged on disk. Multiwrites from `diskEnd` to `nextDiskEnd` are currently being logged from memory to disk. Finally, multiwrites between `nextDiskEnd` and `memEnd` are purely in-memory, and are eligible for absorption.

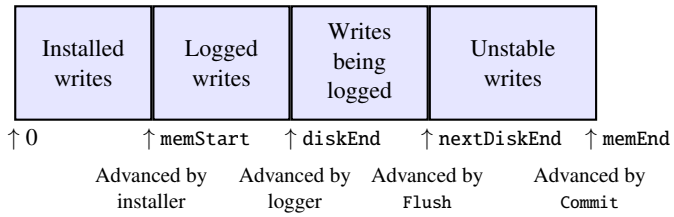


Figure 6: The logical write-ahead log. Vertical arrows indicate designated positions in the logical log. Labels below the arrows indicate what thread or function is responsible for advancing that logical position to the right.

This representation allows GoJournal to precisely specify how concurrent operations modify this abstract state, and how the state changes on crash. For example, although the installer thread performs many disk writes to install multiwrites, its only effect on the abstract state is that it advances `memStart`. Similarly, the logger thread's only change to the abstract state is to advance `diskEnd`. Calling `Flush()` advances `nextDiskEnd`, freezing the data to be logged, then waits for the logger to advance `diskEnd` up to that point. Committing a new multiwrite simply appends it at `memEnd`. Finally, on crash, an arbitrary suffix of the log from `diskEnd` onwards is discarded.

External abstract state: durable lower bound. Although the details of the logical log are important for proving the WAL layer, the caller (i.e., the OBJ layer) does not need to know about installation, group commit, etc. To abstract away these details, the WAL provides a simplified state as its interface, as shown in Figure 7. The simplified state consists of the same list of multiwrites, together with `durable_lb`, which is a lower bound on what set of multiwrites will be preserved on crash. Using a lower bound instead of precisely exporting `diskEnd` means that this abstract view does not need to change if the logger thread adds more multiwrites to disk in the background, and thus hides this concurrency.

Lock-free logging and installation. For performance, GoJournal has dedicated threads that perform logging and installation. However, these threads do not hold any locks while reading or writing to disk. To allow these threads to run concurrently, GoJournal uses two separate header blocks, as shown in Figure 8. One header block (owned by the installer thread) stores the start of the on-disk log, and another header block (owned by the logger thread) stores the end of the on-disk log.

```
Record update := { addr: u64; data: Block; }.
Record State :=
  { multiwrites: list (list update);
    (* at least durable_lb elements are durable *)
    durable_lb: nat; }.

Definition mem_append (ws: list update) :
    transition State unit :=
  modify (set multwrites (fun l => l ++ [ws]));
  ret tt.

(* non-deterministically pick how many
   multiwrites survive the crash. *)
Definition crash : transition State unit :=
  durable <- suchThat (fun s i => durable_lb s ≤ i);
  modify (set multiwrites (fun l => l[:durable]));
  modify (set durable_lb (fun _ => durable));
  ret tt.
```

Figure 7: Parts of the specification for the WAL interface.

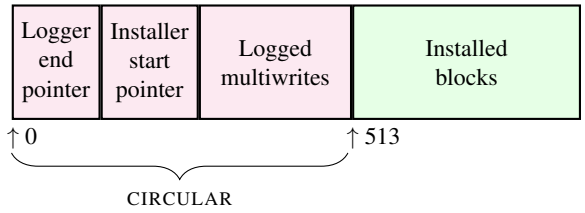


Figure 8: The physical write-ahead log.

This lets the installer and logger concurrently advance their pointers (`memStart` and `diskEnd` respectively) without locks.

Although the logger and installer threads can perform lock-free disk writes, they must still coordinate with one another. For example, the installer cannot run ahead of the logger thread, and the logger thread must coordinate with threads that are appending new multiwrites in memory. GoJournal's proof uses the notion of *monotonic counters* to reason about the safety of the logger and installer's lock-free operations.

The logger thread needs to check that `memStart` is far enough along that the log will have space for the new multiwrite. The proof gets a *lower bound* on the `memStart` variable while holding a lock, which remains true even after releasing the lock. Even though `memStart` might grow after the initial check, the log will only have more space and thus the multiwrite will still fit.

The installer has a similar lock-free region that also reasons using a lower bound. The installer retrieves the updates from the current `memStart` to `diskEnd` in order to start installing them to disk. When the installer eventually trims the log, it needs to be sure not to advance beyond the current logger position, which the proof demonstrates using a lower bound on `diskEnd` from when the logger initially started.

6.2 Logically atomic crash specifications

Throughout the GoJournal stack we specify internal layers using a transition-system specification, such as the examples

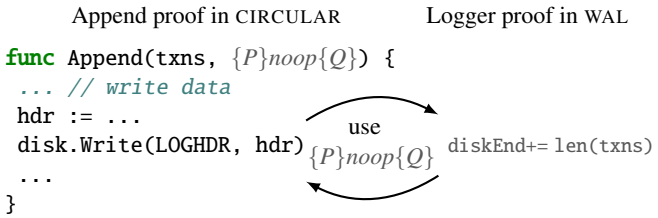


Figure 9: Illustration of how the proof of `Append` executes a logical callback $\{P\}noop\{Q\}$. The logger passes a callback that adds `len(txns)` to `diskEnd`.

illustrated in Figure 7 for the WAL layer. Perennial formalizes what it means for the code in a layer to implement a transition using Hoare triples in a style we call *logically atomic crash specifications*. While the precise encoding involves some technical details of Iris, we explain here the intuition behind these specifications as well as why they are useful.

As a motivating example, consider the moment when the logger thread commits a new batch of multiwrites to the physical log in order to advance the durable point `diskEnd` in the logical log of the WAL layer. It does this by calling into the `Append` method of the CIRCULAR layer, which appends to the small buffer of logged multiwrites. The code for `Append` commits at some internal step when it writes the header block and makes the data valid, and it is at this instant that the logical log's `diskEnd` should be incremented. How can we verify `Append` in the CIRCULAR layer separately from the WAL layer, while still executing the right update in the logger proof?

Logically atomic specifications achieve this separation by having the precondition to `Append` take a logical *callback* [15], which the proof promises to "execute" at the commit point. This callback is a Hoare triple of the form $\{P\}$ *noop* $\{Q\}$, where P and Q are later selected by the logger proof to update the `diskEnd` ghost state of the logical log, as shown in Figure 9. This specification for `Append` provides modularity in that the `Append` proof does not need to know about the logical log and its `diskEnd`, and the logger proof does not need to worry about why `Append` is atomic. A key advance of Perennial's logically atomic crash specs lies in additionally capturing the crash behavior in this callback style, so as to enable a complete proof of crash safety across layers.

6.3 Concurrency within a block (OBJ)

GoJournal's OBJ layer allows the caller to issue reads and writes that are smaller than a full block. This finer granularity helps increase concurrency: for example, the NFS file server packs multiple inodes into a single disk block, and OBJ allows threads to concurrently read and write multiple inodes even if they share a disk block.

At commit time, OBJ's `Commit` may need to perform an "installation read" and read a full block, update the range that was modified by the caller as part of a journal operation, and write back the full block using the WAL layer. To ensure correctness of this read-modify-write operation, `Commit` uses a lock to serialize all commit operations. However, `Read` operations are lock-free: they can execute concurrently with one another and concurrently with `Commit`.

Lock-free reads pose a verification challenge because the disk block can be modified during the read. Consider the example shown in Figure 10, where a single disk block stores many inodes. Inode 1 initially contains the value A, while inode 4 contains B. Thread 1 is committing a write of B' to inode 4 in that block, while thread 2 concurrently reads inode 1 from the same block. To read inode 1, thread 2 will read the entire block, and then copy out the part of the block corresponding to inode 1. The block seen by thread 2 will differ depending on whether thread 1's write happens before or after the read, but inode 1 will contain A in either case.

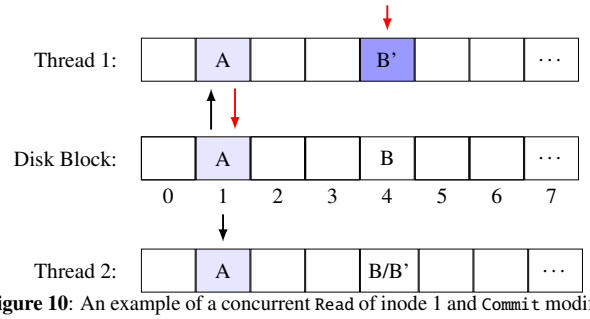


Figure 10: An example of a concurrent `Read` of inode 1 and `Commit` modifying inode 4 in the OBJ layer.

Formally reasoning about the `Read` operation requires the OBJ layer to connect the $a \mapsto_{op} o$ predicate about a disk object (such as an inode) to the disk block containing that object at the WAL layer. However, due to the race condition described above, the `Read` implementation might observe many possible values of the containing disk block. As a result, it is important for the OBJ invariant to relate the $a \mapsto_{op} o$ predicate not just to the latest value of the containing block, but to all recent contents of that block. Specifically, the invariant for $a \mapsto_{op} o$ requires that all recent writes to a's block (since `Read(a)` started) must agree on the part of the block storing o. As a result, regardless of what block happened to be read, the caller is guaranteed to see the correct object o.

7 Implementation

Perennial 2.0 is a re-write of the Perennial 1.0 framework [5], implemented on top of Goose [5, 6], Iris [17, 18], and Coq [28]. Figure 11 shows the lines of specifications and proof for Perennial. Perennial extends the Iris Proof Mode [20] to support convenient interactive proofs in Coq for crashes and Perennial's atomic crash specifications.

Perennial's program logic for crashes provides the formal foundations for framing away crash conditions and for atomic crash specifications. Lifting is implemented as part of the helper libraries. Ghost resources implement lock-free concurrent reasoning, including monotonic counters (to track log positions in Figure 6) and multi-versioned disks (to track logical disk contents at crash time for disk-object ownership).

Component	Lines of Coq
Helper libraries (maps, lifting, tactics)	5,760
Ghost state and resources	5,125
Program logic for crashes	9,375
Total	20,260

Figure 11: Lines of specs and proofs for Perennial.

	Lines of code (Go)	Lines of proof (Coq)	Ratio
CIRCULAR	109	1,905	17×
WAL-STS	555	10,125	23×
WAL	—	2,854	
OBJ	133	2,971	22×
JRNL-STS	121	1,261	24×
JRNL	—	1,640	
LOCKMAP	118	864	7×
Misc.	311	4,177	13×
GoJournal total	1,345	25,797	19×
GoNFS	3,911	*Not verified*	—
SimpleNFS	462	3,749	8×

Figure 12: Lines of code and proof for the components of GoJournal and for SimpleNFS. Ratio is the proof:code ratio, a rough measure of verification overhead.

Using GoJournal, we implemented GoNFS and its core verified subset, SimpleNFS. Both implementations can be mounted by the Linux NFS client, which translates file-system calls into NFS RPCs. GoNFS is sufficiently complete that it can run fsstress and fsx-linux tests through the Linux NFS client.

The breakdown of lines of code and proof by layer, as seen in Figure 12, shows a proof-to-code ratio of about 20× for the layers that involve tricky crash safety and concurrency reasoning. Notably the SimpleNFS proof is relatively short due to the GoJournal implementation and specification largely hiding crash reasoning. The WAL and JRNL layers are split into two parts for proof purposes; the layers labeled "STS" are specified with an atomic state-transition system while the next layer presents an easier-to-use ownership-based interface using separation logic. The write-ahead log's proof is largely in establishing its atomic transitions, while half of the top-level GoJournal proof is proving its separation logic specification as described in §5.

All of the proofs for Perennial, GoJournal, and SimpleNFS are checked by Coq, and we used `Print Assumptions` to verify that the proofs are complete. The code is publicly available.[4]

8 Evaluation

This section empirically answers several questions:

- Is GoJournal sophisticated enough to support real storage systems and to achieve good performance? (§8.1)
- Is GoJournal's concurrency important for storage systems to achieve high performance? (§8.2)
- Are Perennial's verification techniques important for proving the correctness of GoJournal (§8.4) and for enabling application developers to prove their code on top of GoJournal (§8.3)?
- How much effort is required to prove the correctness of GoJournal and applications on top of GoJournal? (§8.5)
- Does verification help developers avoid bugs? (§8.6)

8.1 GoJournal is functional and performant

To evaluate whether GoJournal is sophisticated enough to support real storage systems and to achieve good performance, we measure the performance of GoNFS using three benchmarks: the LFS smallfile and largefile benchmarks, as well as a development workload, consisting of running `git clone` on the xv6 source-code repository [9] and compiling it with `make`. These benchmarks were also used by DFSCQ [7], a previous state-of-the-art verified file system. As a comparison point for GoNFS, we run the Linux kernel NFS server exporting an ext4 file system. The ext4 file system writes data through the journal (using the `data=journal` mount option), so that both systems provide the same crash-safety guarantees. The GoJournal implementation supports atomic but unstable writes, which match the semantics of unstable NFS `WRITE` operations. While all the internal layers of the proof support unstable writes, the separation logic specification presented in §5 does not, so we conducted the evaluation without using unstable writes in GoNFS.

We ran the benchmarks on Linux 5.12.3, using its NFS client to mount both GoNFS and the Linux NFS server. The experiments are run on two machines, a desktop with a relatively slow SSD and an EC2 machine with a fast NVMe disk. The desktop has an Intel Xeon E5-2640 20-core CPU at 2.4 GHz, 64 GB of RAM, and a 256 GB Samsung 850 PRO SSD, which we use to measure in-memory performance with no disk bottleneck as well as the impact of relatively slow storage. The EC2 instance is an i3.metal, which has 72 vCPUs, 512 GB of RAM, and a local 1.9 TB NVMe SSD, which we use to measure performance on fast storage with good random-access performance. To reduce variability we limit the experiment to a single socket, disable turbo boost, disable processor sleep states, and disable Spectre mitigations in the kernel.

We first evaluate GoNFS's performance with a single client issuing requests. Figure 13 shows the results on the Intel Xeon desktop with both file systems backed by RAM, to avoid any I/O overhead — GoNFS takes a simple Go interface for the disk, which we implemented with a large array, while ext4

[4] GoJournal is available at `https://github.com/mit-pdos/go-journal` while GoNFS and SimpleNFS are at `https://github.com/mit-pdos/go-nfsd`.

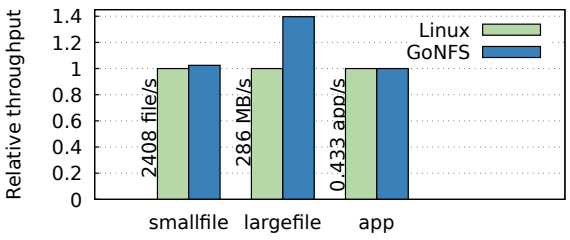


Figure 13: Performance of Linux NFS and GoJournal + GoNFS for `smallfile`, `largefile`, and `app` workload, on a RAMdisk. On an NVMe disk GoNFS achieves at least 90% of Linux's throughput.

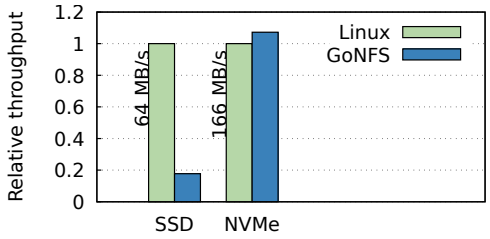


Figure 14: Performance of `largefile` depends on the storage medium. Linux takes advantage of unstable writes to write a large amount of data between barriers but GoNFS flushes to disk frequently.

uses a file in tmpfs.[5] GoNFS achieves at least the throughput of ext4 across the different workloads.

On both the NVMe and slower SSD, GoJournal's performance relative to ext4 is similar on the `smallfile` and `app` workloads (not plotted), again achieving at least 90% of the throughout of ext4. However, GoNFS performance on the `largefile` benchmark is sensitive to disk I/O characteristics, as shown in Figure 14. On the faster NVMe device, GoNFS's large file performance is comparable to ext4's, but on the slower SSD, it drops to under 20% of ext4's throughput. The reason is that the `largefile` benchmark produces a large number of parallel, unstable writes to the same file. GoNFS runs them sequentially due to a per-inode lock, and then journals sequentially because it ignores the unstable write flag. A disk barrier on the SSD takes about 2 milliseconds, so getting good disk throughput requires writing a large amount of data before issuing a barrier, and the 64 KB batch size is insufficient to get the maximum SSD write throughput. Re-running the experiment with unstable writes enabled in GoNFS raises its throughput to 90% of ext4's.

8.2 GoJournal concurrency improves performance

To test whether the concurrency of GoJournal is important for performance we measure the aggregate throughput of GoNFS with an increasing number of clients that run the `smallfile` benchmark. We run the experiment on a physical disk instead of an in-memory file system so that while a thread is waiting for the disk another thread can run. We compare the performance of GoNFS to that of Linux ext4, and to a single-threaded version of GoNFS that has no concurrency.

Figure 15 shows the results on an EC2 i3.metal instance with an NVMe SSD. Both GoNFS and Linux ext4 take advantage of concurrent requests to increase throughput. The single-threaded GoNFS does just barely improve performance, from parallelization among the clients and NFS server, but this amounts to less than 2× throughput with 20 clients than with one. Even with one client, GoNFS achieves 35% higher throughput than single-threaded GoNFS due to concurrency between the RPC thread, the logger thread, and the installer thread. GoNFS achieves higher throughput than Linux ext4, but it is hard to pin down the reason why, because there are many differences in the designs. One possibility is that Linux ext4 does not have concurrent logging and installation (but GoJournal does); another possibility is that ext4 waits for outstanding transactions to finish before flushing to disk (but GoJournal does not).

Figure 16 shows the scaling of GoNFS and Linux, this time on the Xeon desktop with a slower SSD. While GoNFS obtains comparable performance for 7 or fewer cores, Linux scales linearly beyond while GoNFS does not. The scaling in this case primarily comes from batching writes from concurrent clients, resulting in better disk write throughput. GoJournal is not as careful about this, sometimes committing a small amount of data rather than gathering many multi-writes and issuing them together. The NVMe experiment in Figure 15 uses storage with fast enough random-write access that CPU efficiency is more important than issuing large sequential writes; while a disk barrier takes 2 milliseconds on the SSD it takes only 30 *microseconds* on the NVMe disk.

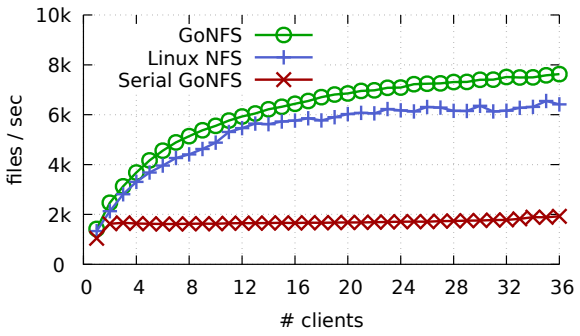


Figure 15: Combined throughput of multiple parallel `smallfile` microbenchmarks, each creating files in different directories, on an NVMe SSD.

8.3 Journaling atomicity simplifies proofs

Many storage systems use journaling because they simplify the implementation in terms of crash safety: the only point at which durable state is modified is when an operation commits. A goal of GoJournal is to carry this insight into proofs, so that a storage system using the journal can prove an operation is atomic using reasoning about durable storage only at the commit point.

[5] Running GoNFS on tmpfs performs slightly worse due to the around 1 microsecond syscall overhead of each disk operation, which ext4 does not incur since everything happens within the kernel.

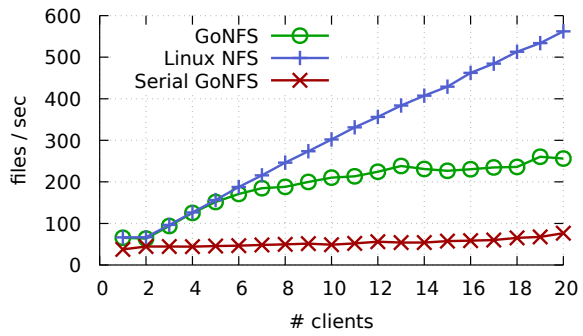


Figure 16: Combined throughput of multiple parallel `smallfile` microbenchmarks, each creating files in different directories, on a (slow) SSD.

One measure of how well GoJournal achieves this goal is the lines of code in SimpleNFS that require reasoning about durable state. SimpleNFS consists of 462 lines of verified code. Only 44 lines of code require proofs to explicitly consider durable state, using crash conditions. In Figure 3, for example, crash reasoning is only needed for lines 6–8 when acquiring and releasing with the crash-aware lock specification. All of the other code does not require reasoning about durable state; it suffices to prove simple crash-free specifications that have a pre- and post-condition, but no crash condition. This formal reasoning is enabled by two techniques from Perennial: lifting disk-object ownership and crash framing.

8.4 Perennial enables modular crash reasoning

Atomic crash specifications are crucial for enabling modular reasoning about crash safety. In GoJournal, atomic crash specs are used at many layer boundaries. Out of the layers shown in Figure 4, CIRCULAR, WAL, OBJ, and JRNL all provide atomic crash specifications, which are used by the layer above. One benefit of atomic crash specs is that they allowed us to develop these layers independently, using the specifications of lower layers before their implements were fully proven, as one would expect of any good API.

The modularity in Perennial largely follows the same structure as the code. Figure 12 shows that the WAL and JRNL proofs were split into an atomic transition specification about the code and a proof-only abstraction on top, but the bulk of the division was due to boundaries in the code that made the implementation manageable. Using separation logic it was easy to prove data structures (like the striped lockmap) and individual utility functions and use their abstract specifications elsewhere in the proof.

8.5 Proof effort

Figure 12 shows the lines of code and lines of proof for GoJournal and SimpleNFS. The hardest part of GoJournal lies in the WAL layer, which has significant lock-free concurrency, and requires careful reasoning about crashes and recovery. This is reflected in WAL's relatively high lines of code, lines of proof, and proof:code ratio. In contrast, SimpleNFS leverages GoJournal's atomicity, and ends up with a much smaller proof relative to its code size.

8.6 Verification prevents bugs

When developing GoJournal, we wrote unit tests to quickly find problems before starting verification, but they did not catch all bugs. While proving GoJournal, we found a subtle bug in absorption. When appending a new transaction in memory, GoJournal has an optimization called absorption where earlier writes to the same address are replaced with the new values. However, we discovered a race condition, where the logger thread could have been already flushing those earlier writes to disk, leading to unpredictable disk contents depending on the order of absorption vs logging. We fixed this issue by introducing the `nextDiskEnd` boundary, as shown in Figure 6: the logger thread only logs up to `nextDiskEnd`, and absorption is only allowed to modify values after `nextDiskEnd`.

9 Conclusion

GoJournal is the first concurrent crash-safe journaling system with a machine-checked proof, built on top of the Perennial 2.0 framework. GoJournal uses Perennial's techniques, including lifting and crash framing, to carry over the atomic benefits of journaling to its formal specification. This enables storage applications to use mostly crash-free reasoning in their proofs. For example, in the verified SimpleNFS server, only 44 lines of code, out of 462, required crash reasoning. GoJournal is sophisticated enough to implement a functional (but unverified) NFSv3 server, GoNFS, that achieves 90% of the performance of a Linux ext4 NFSv3 server on a development workload, far higher than any previous verified file systems, and GoJournal's concurrency enables GoNFS to scale with concurrent client requests. To simplify GoJournal's proofs, Perennial provides logically atomic crash specifications, which capture the crash properties of internal interfaces as single logical transitions, enabling modular proofs for GoJournal's internal layers.

Acknowledgments

We are grateful for feedback from many people that improved this paper, especially Alexandra Henzinger, Jonathan Behrens, Henry Corrigan-Gibbs, Jon Howell, the anonymous reviewers, and our shepherd, James Bornholt. This research was supported by NSF awards CNS-1563763 and CCF-1836712, and by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

References

[1] Dimitris Andreou. Striped (Guava: Google core libraries for Java 19.0). `https://guava.dev/releases/19.0/api/docs/com/google/common/util/concurrent/Striped.html`.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[3] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018.

[4] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1037–1051, Phoenix, AZ, June 2019.

[5] 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)*, pages 243–258, Huntsville, Ontario, Canada, October 2019.

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, January 2020.

[7] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[8] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[9] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. `http://pdos.csail.mit.edu/6.828/xv6`.

[10] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, pages 207–231, Uppsala, Sweden, July–August 2014.

[11] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 287–300, Rome, Italy, January 2013.

[12] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming*, 131:3–21, 2016.

[13] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan 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)*, pages 646–661, Philadelphia, PA, June 2018.

[14] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, Banff, Alberta, Canada, November 2020.

[15] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, Austin, TX, January 2011.

[16] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, Santa Clara, CA, February 2015. USENIX Association.

[17] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[18] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[19] Eric Koskinen and Matthew Parkinson. The Push/Pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 186–195, Portland, OR, June 2015.

[20] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, January 2017.

[21] K. Rustan M. Leino, Richard L. Ford, and David R. Cok. Dafny reference manual. `https://github.com/dafny-lang/dafny/raw/master/docs/DafnyRef/out/DafnyRef.pdf`, December 2020.

[22] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.

[23] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[24] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 169–188, Pohang, South Korea, November–December 2015.

[25] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. Persistent Owicki-Gries reasoning: A program logic for reasoning about persistent programs on Intel-x86. In *Proceedings of the 2020 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Chicago, IL, November 2020.

[26] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[27] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22nd European Symposium on Programming (ESOP)*, pages 169–188, Rome, Italy, March 2013.

[28] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*, July 2020.

[29] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helper for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

A Artifact

A.1 Abstract

The artifact has the code for three tasks: calculating lines of code, running performance experiments, and checking the proofs. Since the artifact is packaged as a virtual machine, the generated graphs do not look exactly like the ones in the paper, but they do demonstrate respectable performance and that everything runs correctly.

A.2 Scope

The artifact will reproduce Figure 11 and Figure 12 (the lines of code tables). It has the code to run the performance evaluation, generating Figure 13 and Figure 16. To back up the claim that the proofs verify, we include the Coq source code and compilation instructions. For convenience the source code already includes the Goose-generated Perennial model of the source code, so the artifact also includes instructions on regenerating this output from scratch.

The paper includes a broader array of graphs than the artifact scripts generate, because it combines data from two benchmarking machines. The performance evaluation was expanded after artifact evaluation to include these more detailed results.

Note that the performance is highly sensitive to your machine and SSD's performance characteristics. We ran the paper's experiments with a litany of techniques to control variance, such as disabling turbo boost and using a single socket (as described in §8.1); until we did this, results were variable, and often hurt GoJournal more than Linux. The artifact is packaged as a VM which doesn't have the same careful setup, but we still believe it is useful because the VM setup documents the software requirements to run the benchmarks.

A.3 Contents

The artifact consists of a virtual machine with all the software required pre-installed and a checkout of the GoNFS source code, which has all the evaluation scripts.

A.4 Hosting

You can obtain the artifact's VM image via Zenodo DOI 10.5281/zenodo.4657115. The artifact instructions are at `https://github.com/mit-pdos/go-nfsd/tree/master/artifact`, as well as the Vagrantfile used to generate the VM image.

A.5 Requirements

The virtual machine uses VirtualBox. We configured it with 8GB of RAM (though 4GB is probably fine) and 4 cores; more cores might improve scalability numbers, although more clients help saturate the SSD even if you have fewer cores than clients. If the drive hosting the image is a hard drive, the "SSD" performance numbers will look quite bad.

Running the evaluation natively requires a variety of software that is documented by the VM provisioning scripts, which are in the `mit-pdos/go-nfsd` repo alongside the instructions.

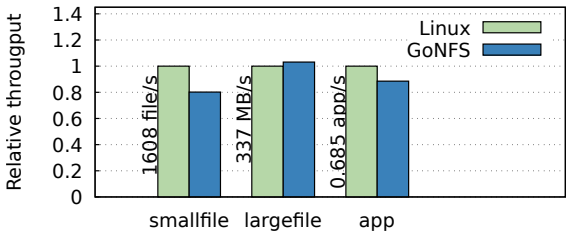
A.6 Results from artifact VM

On a MacBook Pro with a 2.4 GHz Intel i9-9980HK, we obtained the performance results in this section from running the artifact in a virtual machine. These experiments use the default VM configuration, with 8GB of RAM and 4 cores, on a host with 8 cores.

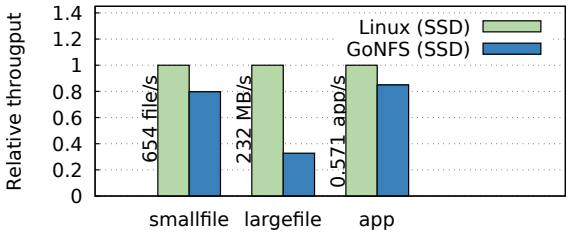
Figure 17 shows the results of running the microbenchmarks on this hardware configuration. Figure 17a is analogous to Figure 13. Figure 17b includes the `largefile` results shown in Figure 14. Between these two figures we see more variability on `smallfile` than when run on physical hardware. The `largefile` results are as expected, since the SSD in this machine has performance somewhere in between the SSD in the desktop machine and the NVMe drive from an i3.metal instance.

Figure 17c shows the results of running the largefile benchmark across a variety of software configurations, all on an SSD; these were not directly shown in the paper. From these results we concluded that GoNFS can get good performance, if using unstable writes. The "Linux (sync)" configuration uses ext4 in `data=journal` mode but additionally mounts the NFS share with the `sync` flag, which changes the benchmark to a completely sequential and synchronous one. In this configuration Linux's optimizations do not kick in and it obtains the same performance as GoNFS using stable writes.

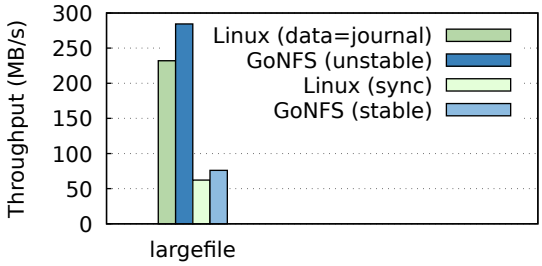
Finally, Figure 18 shows scalability of the `smallfile` benchmark, analogous to Figure 16. Even though this disk gets much better throughput and has a barrier latency of only 0.4 ms (in the virtual machine) rather than 2 ms, the experiment has the same trend as on the slower SSD.



(a) `bench.pdf` (RAM)



(b) `bench-ssd.pdf` (SSD)



(c) `largefile.pdf` (SSD)

Figure 17: Microbenchmarks and app benchmark run inside a VM.

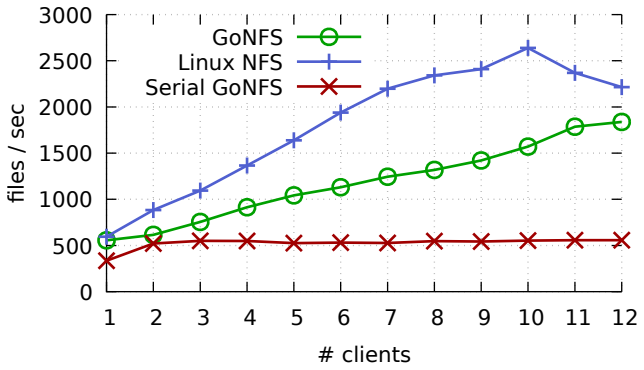


Figure 18: `scale.pdf`, showing scalability of the `smallfile` benchmark. The VM had 4 CPU cores for this experiment.

STORM: Refinement Types for Secure Web Applications

Nico Lehmann
UC San Diego

Rose Kunkel
UC San Diego

Jordan Brown
Independent

Jean Yang
Akita Software

Niki Vazou
IMDEA Software Institute

Nadia Polikarpova
UC San Diego

Deian Stefan
UC San Diego

Ranjit Jhala
UC San Diego

Abstract

We present STORM, a web framework that allows developers to build MVC applications with compile-time enforcement of centrally specified data-dependent security policies. STORM ensures security using a *Security Typed ORM* that refines the (type) abstractions of each layer of the MVC API with logical assertions that describe the data produced and consumed by the underlying operation and the users allowed access to that data. To evaluate the security guarantees of STORM, we build a formally verified reference implementation using the Labeled IO (LIO) IFC framework. We present case studies and end-to-end applications that show how STORM lets developers specify diverse policies while centralizing the trusted code to under 1% of the application, and statically enforces security with modest type annotation overhead, and no run-time cost.

1 Introduction

We trust web applications with our most sensitive data: our finances, health records, email, or even our participation in political protests. While application developers go to great lengths to protect this data, today's approach to safeguarding sensitive data by sprinkling access control checks throughout the application is not working. Even companies with dedicated security teams are failing. For example, in 2018 Facebook accidentally allowed third-party applications to access the photos of 6.8 million users without their explicit permission [1]. This was not their first (nor last) leak. And Facebook is not unique: *sensitive data exposure* and *broken access control* are—and have been for almost a decade—on the OWASP top ten list of most common web application vulnerabilities [2, 3].

To fundamentally address this class of bugs, we need to reduce the amount of code developers need to get right. One promising approach to doing this is to *centralize policy specification*, i.e., specify data access control policies in a centralized place, and enforce policies *automatically*. This could reduce the code developers need to get right from the whole application—as a single missing check could introduce a vulnerability—to the policy specification code.

Centralizing policy specification is not a new idea. Several web frameworks (e.g., HAILS [4], JACQUELINE [5], and LWEB [6]) already do this. These frameworks, however, have two shortcomings that have hindered their adoption. First, they enforce policies *at run-time*, typically using dynamic information flow control (IFC). While dynamic enforcement is better than no enforcement, dynamic IFC imposes a high performance overhead, since the system must be modified to track the provenance of data and restrict where it is allowed to flow. More importantly, certain policy violations are only discovered once the system is deployed, at which point they may be difficult or expensive to fix, e.g., on applications running on IoT devices [7].

Second, these frameworks are *invasive*—they typically require modifications to the language runtime and database object-relational mapping (ORM). For example, JACQUELINE uses a *faceted* ORM and runtime to keep track of multiple facets of any individual value and only shows the right facet to the right user (e.g., when reading a password, a user can see their own password but get a default facet when trying to read another user's password). HAILS and LWEB, on the other hand, use *labeled values* at the ORM and language level to restrict the flow of sensitive, labeled data. This means that developers need to write code that is aware of faceted or labeled values, i.e., they need to write code that is aware of the underlying IFC enforcement mechanism. Worse, this invades policy specification. For example, in HAILS, developers can't simply write declarative policies, they often need to use the low-level APIs used to track and enforce IFC to, for example, inspect and manipulate labeled values [4, 8]. This not only increases the amount of *code* they need to get right, but also makes it hard to get the *policy* right since manipulating labeled values is still an IFC expert—and not web developer—task.

We built the STORM web framework to address these shortcomings. With STORM, users specify all security policies in a declarative language, alongside the *data model*, the description of the application database schema. Policies are *logical assertions* that describe which users are allowed to view, insert, or update particular rows and columns of each

table in the database. STORM enforces these policies *statically*, at compile-time—and *non-invasively*, without translating them to labels or facets. This means that (1) STORM does not impose any run-time overhead, (2) developers can catch bugs due to policy violations (e.g., where the application incorrectly handles sensitive data) early, and (3) they don't need to understand the details of the underlying enforcement mechanism to specify or audit policy code.

STORM statically enforces policies using *refinement types* [9]: types decorated with logical assertions that can constrain values, e.g., to say that an `Int` is non-negative or that a `User` is the author of a `Paper`. Our key insight is to refine STORM's API with logical assertions that describe the data produced and consumed by the underlying operation and the users allowed access to that data. We use this insight to realize STORM via four contributions.

1. Design (§ 3) Our first contribution is a novel design that enriches the data model with a declarative policy—the *refined data model*—to generate an application-specific ORM layer, which STORM annotates with refinement types that reflect the security policies. To our knowledge, this is the first framework to statically and unobtrusively enforce policies previously thought to only be expressible using runtime enforcement.

2. Implementation (§ 5) Our second contribution is an implementation of STORM in Haskell that uses LIQUIDHASKELL, an off-the-shelf refinement type checker to *statically and automatically verify* whether the application code using the security-typed ORM—e.g., code handling user requests and rendering HTML responses—adheres to the policies. STORM does this without imposing any invasive changes to the language runtime or database ORM interface. At most, developers write (untrusted and verified) light-weight type annotations to help the checker prove their code does not leak.

3. Verification (§ 6) Our third contribution is a formally verified reference implementation that proves that the STORM API is secure by showing how to reduce a well-typed STORM program into an LIO [10] program that never throws security exceptions. This allows us to carry over the previously mechanized non-interference results from LIO [6, 10] to show that well-typed programs cannot leak or corrupt sensitive data.

4. Evaluation (§ 7) Our final contribution is an empirical evaluation of the *expressiveness* of STORM's policy mechanism, the programmer *effort* needed for static enforcement, and, ultimately, of the *reduction* in the amount of code the developer has to get right to not leak data in real web applications. First, we show that our centralized policy specification approach is expressive enough to describe, often more naturally, a large suite of policies from the literature. Second, we use STORM to write statically verified implementations of several case studies from the literature, including those that had previously only been amenable to dynamic policy enforcement, and show that the effort is modest: the programmer need only write 1 line of refinement type signatures per 20–30 lines of code (LOC). Third, we use STORM to build and deploy two new end-to-end web applications for collaborative text editing and video-based social interaction, that have been used at our university and at several academic workshops, respectively. We demonstrate that STORM distills the code that the developer has to get right to compact, auditable policies (under 70 LOC) that comprise under 1% of the application.

System	Audit	Static	Uninvasive	IFC
SWIFT [11]	✗	✓	✗	✓
SELINKS [12]	✓*	✓	✗	✗
RESIN [13]	✗	✗	✓*	✗
URFLOW [14]	✓	✓	✓	✓*
IFDB [15]	✓	✗	✗	✓
HAILS [4]	✓*	✗	✗	✓
JACQLN [5]	✓	✗	✗	✓
LWEB [6]	✓*	✗	✗	✓
DAISY [16]	✓	✗	✗	✓
STORM	✓	✓	✓	✓

Figure 1: We compare STORM to previous web frameworks along various design goals. **Audit**: are the policies centralized and easily auditable; **Static**: is the enforcement at compile-time; **Uninvasive**: does enforcement require changes to the run-time; and **IFC**: does the framework enforce information flow control. We write ✓*for almost-met goals.

2 Goals & Related Work

We designed STORM with several goals in mind. First, the framework should provide *information flow control* (IFC) security to prevent not only explicitly bad data flows, but also implicit leaks where publicly viewable results are conditioned on sensitive data. Second, the framework should enable a centralized, and hence, easily *auditable* policy specification. Third, to find errors early, provide design-time feedback and avoid run-time overhead, the framework should permit *static* enforcement via *automatic*, compile-time verification. Fourth, the framework should not require *invasive* modifications to language run-times, database ORMs, or libraries. STORM builds on previous work, summarized in Figure 1, which have made great strides towards these goals.

IFC There are many flavors of IFC with different trade-offs [17, 18]. Systems differ in *when* they enforce IFC: at run-time via labels [10, 19–22], faceted values [19, 22, 23], secure multi-execution [24], or at compile-time via types [12, 25–28], or static analysis [29], or a hybrid combination [30–32]. Even within the same category, these systems differ in granularity of enforcement—from fine-grained to coarse-grained [33, 34], and the kinds of policies users can specify [35]. The SWIFT [11] system uses a static IFC type system [30] to enforce compile time security, but does not integrate with database ORMs, and hence, lacks centralized

auditable specifications. IFDB [15] and DAISY [16] show how to perform fine-grained IFC within DB systems, but are not static, and focus on databases—and are thus not complete frameworks for building applications. STORM draws inspiration from the HAILS [4], LWEB [6], and JACQUELINE [5] frameworks which enforce auditable IFC policies that are associated with the application's data model. However, these approaches all perform dynamic enforcement and require invasive changes to the DB layer or run-time.

Static Several static frameworks express data-dependent policies using dependent types [36–38, 38, 39], labels [11, 12, 28, 40], or first-order logic formulas [41]. All the above require the programmer to sprinkle policy specifications across the application controller and view code, which is error prone and makes auditing difficult. SELINKS [12] centralizes policies within special functions that un/wrap data with labels, but requires invasive changes to the DB and run-time to propagate labels and does not prevent implicit leaks. URFLOW [14] enables verification of centralized and auditable specifications without requiring invasive changes, by using a bespoke symbolic execution algorithm to statically verify that the generated SQL queries are (semantically) contained in some allowed set. However, to statically compute the SQL queries, URFLOW requires programmers to write their applications in a domain-specific language (DSL). Further, URFLOW's approach is insufficient for full IFC as it misses implicit flows through SQL queries (as illustrated in § 3.4). In contrast, STORM enforces full IFC via off-the-shelf refinement type checking for a general purpose language with a rich ecosystem with tools and libraries for networking, databases, data serialization, etc. STORM uses a statically typed API for monadic IFC in the style of [42, 43], specifically, the approach of LIFTY [44], a core calculus that shows how to track IFC with logical refinement types. Unlike STORM, LIFTY cannot be used to build secure applications: it does not have database APIs, a language to specify centralized policies, formal guarantees for data-dependent policies, or even a way to write executable code.

System-based Security Several frameworks employ privilege separation to run application components with least privilege [45–49]. Others like RESIN [13] and QAPLA [50] restrict access to data by modifying the run-time to use fine-grained discretionary access control, or use cryptography to provide data confidentiality, authenticity, and integrity in the presence of compromised application components [51–53], or use proxies to implement web application firewalls [54, 55]. While some of these approaches, e.g., the use of cryptography are complementary to our approach, without IFC, they cannot prevent leaks that STORM eliminates by construction.

3 Design

We illustrate the design of STORM with a `WishList` application where users can share wishes with followers. STORM uses the *model-view-controller* (MVC) paradigm, where an application has three key elements: *models* which describe the persistent data important to the application, typically stored in a database (DB) and accessed via an *Object-relational mapping* (ORM); *views* which describe how the data corresponding to, e.g., users' requests are rendered on webpages via some combination of CSS, HTML and JavaScript; and *controllers* that respond to user's requests by suitably querying the DB via the models API, to produce an HTML or JSON results.

3.1 Auditable Policies via Refined Models

The key innovation in STORM is to centralize data-dependent security policies with the data model, in a *refined models* file.

Models & Policies Figure 2 shows the refined models file for the `WishList` app. The left column describes the data schema, as a collection of three tables `User`, `Wish` and `Follower`. Each row of the `User` table comprises the user's `name` and `email` address. Each row of the `Wish` table has an `owner` that identifies the `User` that the wish belongs to, a text `description` of the wish, and a numeric `price`. Each row of the `Follower` table describes a tuple where `user1` *follows* `user2`, with the `status` column indicating whether a follow-request has been initiated (`"pending"`), accepted (`"ok"`) or rejected (`"no"`). STORM lets the programmer specify policies that govern which DB rows can be inserted and which DB columns can be read or updated. A *policy* is a predicate over a `row` and `user` that is `True` if the `user` has access and `False` otherwise. The policy predicate can refer to all the columns of the row (whose column the policy is attached to) and so the values of those other columns can be used to determine whether the user has access. For example, we specify that `Wish`es can only be inserted by their owners via the policy `@IsOwner` which holds when the `user` equals the `owner` of the row. Similarly, we specify that each `Wish`'s `description` and `price` should only be read by the `owner` unless they are explicitly public via the policy `@Public` which holds when the `user` is the owner or the `level` is `"public"`. (Ignore the shaded `Follow` for now: we will return to it in § 3.3.) Finally, we specify that only the owner is allowed to update the `description` and `price`.

Default Policies The programmer can associate **`default`** policies with all the rows and columns not explicitly constrained otherwise. For example, `Allow` grants access to all users, while `Deny` grants access to none. Hence, **`default`** `read @Allow` and **`default`** `insert, update @Deny` say that (unless otherwise specified) anyone can `read` every column, and no one can `insert` rows or `update` columns.

3.2 Access Control

Let's see how STORM enforces the `Public` policy. Figure 3 shows a controller `showWishes` that responds to a request to display the wish list for a given `user`. (For now, *ignore* the shaded code.) The controller uses the models API to create a `Query` of the form `Owner ==. user`, which it executes using the ORM

```
User                                    declare follows : UserId → UserId → Bool
  name    Text
  email   Text                          def IsOwner(row: Wish, user: User):
                                          row.owner == user.id
Wish
  owner   UserId                        def Public(row: Wish, user: User):
  descr   Text                            IsOwner(row, user) || row.level == "public"
  level   Text
  price   Int                           def Follow(row: Wish, user: User):
                                          row.level == "follower" && follows(user.id, row.owner)
  insert                  @IsOwner
  read    [descr,price]   @PublicFollow def PublicFollow(row: Wish, user: User):
  update  [descr,level]   @IsOwner        Public(row, user) || Follow(row, user)

Follower                                def OkFollows(row: Follower):
  user1   UserId                          row.status == "ok" ⇒ follows(row.user1, row.user2)
  user2   UserId
  status  Text                          def IsPending(row: Follower, user: User):
                                          row.user1 == user.id && row.status == "pending"
  assert                  @OkFollows
  insert                  @IsPending    def OkOrNo(old: Follower, new: Follower, user: User):
  update  [status]        @OkOrNo         old.user2 == user.id && new.status `in` ["ok", "no"]

default read              @Allow        def Allow(row: a, user: User): True
default insert, update    @Deny         def Deny(row: a, user: User): False
```

Figure 2: Refined Models: A centralized specification for the Wishlist App

```
showWishes user = do
  viewer  <- authUser
  let pub  = Level ==. "public"
  let chk  = if viewer == user then true else pub
  let qry = Owner ==. user &&. chk
  wishes <- select qry
  descrs <- mapM (project Descr) wishes
  respond (show descrs)
```

Figure 3: A `showWishes` controller. The highlighted code is needed for conformance with the `Public` policy.

API function `select` to get all the DB `Wish` rows belonging to `user`. Next, it extracts the `description` column for each row by invoking the ORM API function `project` with the name of the desired field. Finally, the controller uses the view API function `respond` to send the descriptions to the session user.

Enforcement Recall that the policy `Public` stipulates that `descriptions` should only be visible to the `owner` unless the `level` is `"public"`. Indeed, the `showWishes` controller, sans the shaded parts, is dodgy as the current session user could be asking for someone *else's* wishes! STORM detects this error at compile time, by: (1) inferring that the `qry` will return all rows owned by `user`, (2) using the policy on `Descr` to determine that the `project`'s results depend on values that are *allowed* to be viewable only by `user` (unless marked `"public"`), and then (3) complaining that by calling `respond` the results can be observed by the *sessionUser* who may be different than `user`.

We can fix `showWishes` by modifying the query when `user` is different than *sessionUser*. The modifications are shaded in Figure 3. First, we use the view API's `authUser` function to get the current session (`viewer`), which we use to add a `chk` clause to the DB query. When the target `user` *is* the session user, the `chk` clause is the trivial query `true` (which holds of all rows). However, if the target `user` is different, then the `chk` clause stipulates that the `level` column be `"public"`. The type checker infers that `qry` returns all rows owned by the session user, but *only* the public rows of *other* users. Hence, the type checker determines that the subsequent data release via `project` and `respond` conforms to the `Public` policy.

3.3 Information Flow Control

Next, let's see how STORM lets the programmer enforce IFC policies that (1) span values *across* different rows and tables, and (2) restrict how data flows to *multiple* users who may be unknown at the point where the data is accessed.

Policy Let us add social capabilities to our application by letting users have *followers* with whom they can share their wishes. We model this notion as a many-to-many `Followers` relationship table and then add `"follower"` as a new possible

level value. Now the access to a particular Wish depends on data residing in another row, in another table—a record existing in the Followers table. STORM lets the programmer specify this requirement simply by changing the read policy for descr and price to @PublicFollow which is defined on the right in Figure 2. The key insight to specifying such a cross-table policy is that the existence of a Follower record *witnesses* the *follows* relationship between two users. The refined-models in Figure 2 makes this notion manifest as follows. First, at the top, we **declare** the relationship as a binary predicate follows between two UserIds. Second, the line assert @OkFollows says that for each row of the Follower table, the follows predicate holds between user1 and user2 *if* the status is "ok". Third, we use the predicate to define the Follow policy that says that when a wish's level is restricted to "follower" then the viewer user must be a follower of the wish owner. Finally, we use Follower to define a new policy PublicFollow that governs who is allowed to read the descr and price fields. This new policy captures our informal requirement about the three levels of viewers: "public", "private" and "follower".

Controller Continuing with the social aspect of the application, a nice feature would be to send an email notification containing a user's (non-"private") wish list, to all of the user's followers, a few days before that user's birthday. Our application implements this feature in the notifyFriends controller in Figure 4. The code starts by selecting the list of non-private wishes and projecting out their descriptions into the list descrs. Next, we query the DB to determine the list of followers flwUsrs. Finally, we use sendMail containing the wish decriptions descrs to all the users in flwUsrs.

Enforcement In the first phase notifyFriends accesses sensitive information that should only be made available to a data-dependent set of users who are, at that point, still to be determined. However, STORM's models API tracks this fact by combining the semantics of the wshQ query with the read policy associated with Descr to infer that only the followers of user are *allowed* access to the results of the first sub-computation that creates descrs. In the second phase, STORM's models API tracks the semantics of the flwQ query to determine that flws is a set of valid follows-tuples, and hence, that each user in flwUsrs is a valid follower of user. In the final phase, the signature for sendMail in STORM's view API checks that all the recipients in flwUsrs have the right access, and hence verifies the controller. If the programmer forgot the Status ==. "ok" clause, type checking would fail as flws would contain pairs with pending status, and hence, flwUsrs would contain possible non-followers outside the set allowed access by the first phase.

3.4 Implicit Flow Control

Next, let's see how STORM prevents *implicit* IFC violations involving publicly viewable data that was generated *conditioned* upon data the recipient should not be privy to. Recall, from Figure 2, that each wish has a price that should only be read

```
notifyFriends user = do
  -- Get list of wishes
  let wshQ = Owner ==. user &&.
             Level <-. ["public","follower"]
  wishes  <- select wshQ
  descrs  <- mapM (project Descr) wishes
  -- Get list of followers
  let flwQ = User1==. user &&. Status ==. "ok"
  flws    <- select flwQ
  flwIds  <- mapM (project User2) flws
  flwUsrs <- select (UserId <-. flwIds)
  -- Notify followers
  sendMail flwUsrs (show descrs)
```

Figure 4: A notifyFriends controller. The highlighted code eliminates the IFC violation of the PublicFollow policy.

```
usersWithExpensiveWishes min = do
  let qry = Price ≥. min &&. Level ==. "public"
  wishes <- select qry
  users  <- mapM (project Owner) wishes
  respond  (show (nub users))
```

Figure 5: A usersWithExpensiveWishes controller: The highlighted code eliminates the implicit flow violating the PublicFollow policy. The nub function removes duplicates from a list.

per the PublicFollow policy, i.e., by everyone (if "public"), by followers (if "follows") or else, only by the owner. The code in Figure 5 implements a controller that shows the session user a list of all the users that have a wish whose price exceeds the min threshold. (For now, ignore the shaded code.) If a programmer is not careful, they may think this code conforms to the application's policy as it returns a list of wish owners and owner is a publicly viewable column governed by the **default** read @Allow policy.

Enforcement However (absent the shaded code) STORM is unimpressed, as the list of expensive wishes was obtained by conditioning over the sensitive price column. STORM's models API tracks that the qry accesses the Price field, and infers that the result of the DB computation select qry should only be observed by users that satisfy the PublicFollow policy. Thus, when responding to the session user on the last line, STORM reports an error as it cannot prove that the session user satisfies the PublicFollow policy. To fix the code we must restrict the Price comparison to the wishes that the session user is allowed to access, for example, to all "public" wishes, as shown by the shaded diff in Figure 5. Now, as detailed in § 5.1, the type checker uses the models API to track the semantics of qry to infer that the results of the select computation may be made available to *all* viewers, thus verifying that the code conforms to the application's centralized policy.

4 Brief Intro to Refinement Types & IFC

STORM is implemented using two foundational blocks: Refinement types (§ 4.1) and Compositional IFC (§ 4.2).

4.1 Refinement Types

Refinement types let the programmer decorate the source program's types with logical assertions from a decidable logic to specify *subsets* of values of the decorated type [56, 57]. For example, the non-negative integers can be specified as

```
type Nat = {v:Int | 0≤v}
```

Pre- and Post-Conditions The user can write pre- and post-conditions for functions by refining the input and output types of functions. For example, `sum` adds the integers $0 \ldots n$

```
sum :: n:Nat → {v:Nat | n≤v}
sum 0 = 0
sum n = let t = sum (n-1) in n + t
```

We assign `sum` a refined function type, comprising an *input* type (pre-condition) that says that the function should only be invoked on non-negative integers, and an *output* type (post-condition) that says the result is a non-negative integer lower-bounded by the input n. Refinement type checking proceeds be generating a *verification condition* (VC), a logical formula whose validity implies the program type checks [9, 39, 58–60].

Bounded Refinements Generic APIs require a means of abstracting over particular policies and invariants of individual applications. We do so using *bounded* refinements [61] which allow (1) abstracting over the refinements (like type variables `<A ...>` abstract over concrete types) and (2) constraining the refinements with which the variables can be instantiated (like subtyping bounds `<A extends ...>` constrain type instantiation). For example, we can type the function composition operator `compose f g x = f (g x)` as

```
compose :: (Cmp f g r) ⇒ (y:b → {v:c | f(y,v)})
                       → (z:a → {v:b | g(z,v)})
                       → (x:a → {v:c | r(x,v)})

where Cmp f g r ≐ ∀x,y,z. g(x,y) ⇒ f(y,z) ⇒ r(x,z)
```

In the above, f, g and r are (abstract) *refinement variables*. The specification says that `compose` takes as input two functions that respectively map their argument y (resp. z) to an output v that satisfies the assertion $f(y,v)$ (resp. $g(z,v)$), and returns as output a function that maps its input x to a value v that satisfies the assertion $r(x,v)$. The abstract refinements f, g and r are related by the refinement bound $Cmp\ f\ g\ r$ which states that r is the relational composition of f and g. The signature is generic and precise in that it abstracts over the concrete post-conditions established by the arguments to `compose` while still letting us characterize the semantics of the result. Further, the (Horn clause) structure of the bound ensures that type checking remains decidable. Thus, we can use an SMT solver to automatically verify

```
sum2 :: n:Nat → {v:Nat | n≤v}
sum2 = compose sum sum
```

by automatically inferring that the refinement variables f, g, and r can all be instantiated to the refinement $\lambda\, n\, v \rightarrow n \leq v$.

4.2 Compositional IFC

Next, we give a high-level overview of the method used by STORM to enforce IFC in a compositional manner.

Primitive Operations and Computations An *application* is a collection of request *handlers*. Each handler is the sequential composition of a set of primitive *operations* that either read from or write to the database or send results to some users. For example, consider the handler e_{14} illustrated in Figure 6 that is composed from the primitive operations $e_1, \ldots, e_4$ as:

$$e_{12} = \textbf{do}\ e_1; e_2 \qquad e_{34} = \textbf{do}\ e_3; e_4 \qquad e_{14} = \textbf{do}\ e_{12}; e_{34}$$

Thus e_{12}, e_{34} and e_{14} are *computations* built from primitive operations using the sequential composition $(;)$ operator.

Authorizees and Observers Each primitive operation either *reads* data, e.g., from the database, that only a subset of users, the *authorizees*, are allowed to view, or *writes* data, e.g., to the network, thus providing it to a subset of recipients, the *observers*. For example, suppose that in the handler in Figure 6, the operations e_1 and e_2 read sensitive data with authorizees $auth_1$ and $auth_2$ respectively. Similarly, assume that e_3 and e_4 write data to observers obs_3 and obs_4 respectively.

Information Flow Control requires that whenever some primitive operation e_i reads data that is restricted to authorizees $auth_i$, all *subsequent* operations e_j only write data to observers obs_j that are contained in $auth_i$. For example, the handler in Figure 6 respects the given security policy if

$$\begin{array}{ll} obs_3 \subseteq auth_1 & obs_4 \subseteq auth_1 \\ obs_3 \subseteq auth_2 & obs_4 \subseteq auth_2 \end{array} \tag{1}$$

To enforce IFC we could expand each handler out into its sequences of primitive operations and then do the inclusion checks, e.g., via symbolic execution [14]. Sadly, this approach runs aground when there is a combinatorial explosion of paths through the handlers, or with loops or recursion which generate infinitely many possible computations.

Compositional Enforcement STORM circumvents path explosion using a two-step compositional approach [42, 44, 62], where each computation e is typed as $\langle auth, obs \rangle$ where $auth$ (resp. obs) under-approximates (resp. over-approximates) the authorizees (resp. observers) of e. First, STORM assigns the primitive operations the types

$$\begin{array}{ll} e_1 :: \langle auth_1, \emptyset \rangle & e_3 :: \langle \bar{\emptyset}, obs_3 \rangle \\ e_2 :: \langle auth_2, \emptyset \rangle & e_4 :: \langle \bar{\emptyset}, obs_4 \rangle \end{array}$$

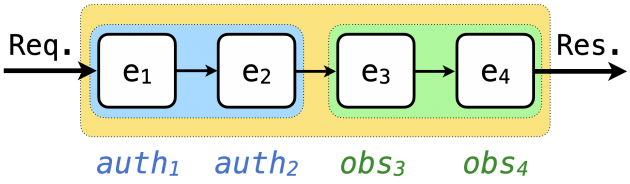


Figure 6: A request handler that sequences the primitive operations $e_1 - e_4$ with authorizees $auth_i$ and observers obs_j.

where $\emptyset$ and $\bar{\emptyset}$ are the empty and universal sets of users. Next, STORM assigns the ; operator a type that ensures that whenever we compose two computations e and e': (a) The observers of e' are *contained* in the authorizees of e, i.e., $obs' \subseteq auth$ (b) The authorizees of $e;e'$ are the *intersection* of those of e and e', i.e., $auth \cap auth'$, and (c) The observers of $e;e'$ are the the union of those of the sub-computations, i.e., $obs \cup obs'$. The implementations of e_{12} and e_{34} yield the (trivial) constraints $\emptyset \subseteq auth_1$ and $obs_4 \subseteq \bar{\emptyset}$, and types

$$e_{12} :: \langle auth_1 \cap auth_2, \emptyset \rangle \qquad e_{34} :: \langle \bar{\emptyset}, obs_3 \cup obs_4 \rangle$$

Finally, when we compose e_{12} and e_{34} to get the computation e_{14} we get the constraint $obs3 \cup obs_4 \subseteq auth_1 \cap auth_2$ which is equivalent to the IFC constraints (1). Next, let us see how our implementation represents the authorizees and observers with refinements and uses a typed API to compute, propagate and check those sets to enforce centralized security policies.

5 Implementation

We designed STORM to enable compile-time enforcement of centralized, data-dependent policies without any modification to the run-time. To achieve these goals, our design requires: (1) An expressive, data-dependent way to associate DB *fields* with the authorizees allowed access to those fields. (2) A way to connect DB *queries* with the authorizees allowed access to the query results. This set of users depends on the data in the underlying rows, so we also need to characterize the values of the rows returned by the query. (3) A way to *aggregate* the authorizees and observers across computations. (4) A way to ensure that observers who are *provided* sensitive data are a subset of the users authorized by the policy.

STORM achieves the above goals by *refining* the type abstractions (API) provided by each MVC layer with logical assertions that describe the *invariants* of the data processed by the operations, and the *policies* that govern access to that data. This is tricky as the assertions must simultaneously satisfy three properties. First, they must be *precise* to capture the semantics of the policies and DB operations. Second, they must be *generic* to enable reuse across many different web applications. Third, they must be *decidable* so applications can be automatically verified by SMT solvers. Next, we introduce the three principal data types of the STORM API (Figure 7) and use them to design a precise, generic and decidable API.

Policies A STORM *policy* is a binary predicate on a DB row and user, which we represent as a predicate of type `row → user → Bool`. The policy is data dependent as the predicate can use the `row`'s values to determine if a `user` is authorized. For example, Figure 2 specifies the policy *Public* as a predicate on the `Wish` row and a `user`. Each policy is attached to a *column* of a row specified in the ORM description in the `models` file. For example, in Figure 2 we attach the policy *Public* to the `description` column to specify that the description should only be viewable to users other than the `owner` when the row's access `level` is `"public"`.

Fields ORM libraries typically represent individual database columns as their own datatypes. STORM uses the PERSISTENT library [63] which represents each DB column as a type `Field row val` where `row` represents the underlying row (table), and `val` represents the value of the column itself. For example, in the code below, the DB table on the left is translated to the *fields* `Owner`, `Descr` and `Level` which respectively represent the corresponding DB columns as plain program values.

DB Table	**ORM Fields**
`Wish`	
`owner UserId`	`Owner :: Field Wish UserId`
`descr Text`	`Descr :: Field Wish Text`
`level Text`	`Level :: Field Wish Text`
`price Int`	`Price :: Field Wish Int`

Policies* in *Fields STORM's first pillar is a refined `Field` that represents policies at the type-level, by parameterizing the datatype with two abstract refinements (Figure 7):

pol: `row → user → Bool` *sel*: `row → val → Bool`

The refinement *pol* is instantiated with the policy attached to the `Field`; *sel* is a selector predicate that provides a type-level description of the value of the corresponding column. STORM uses the models file in Figure 2 to automatically generate the following types for `Owner`, `Descr`, `Level` and `Price`

```
Field ⟨⊥, λr v → v=r.owner⟩ Wish UserId
Field ⟨PublicFollow, λr v → v=r.descr⟩ Wish Text
Field ⟨⊥, λr v → v=r.level⟩ Wish Text
Field ⟨PublicFollow, λr v → v=r.price⟩ Wish Int
```

Thus, STORM's refined fields provide a uniform mechanism to lift data-dependent specifications up into types.

Queries Modern ORMs, going back at least to LINQ [64], allow the user to use `Field`s to build *queries*, e.g., of type `Query row` to represent query objects (or ASTs, not the results themselves) that access the DB table represented by `row`. STORM introduces a way to refine the types of the query API to track, at the type-level, the authorizees of the query results. As the policies are data-dependent, our API must also track the values of the rows in the query results. STORM achieves these goals via the second pillar of its API, a type that represents each DB `Query` parameterized by two refinements (Figure 7):

pol: `row → user → Bool` *inv*: `row → Bool`

```
data Field ⟨pol: row → user → Bool, sel: row → val → Bool⟩ row val
data Query ⟨pol: row → user → Bool, inv: row → Bool⟩ row
data RIO   ⟨auth: user → Bool, obs: user → Bool⟩ val
```

Figure 7: The central types of the STORM API

As with `Field`, the refinement *pol* denotes the authorizees for each `row` returned by the query. Crucially, our query building API will ensure that *pol* intersects the authorizees across all the columns read by `Query`, not only those for the particular fields that are ultimately viewed by the viewers. This allows STORM to track implicit flows when filtering over sensitive columns, e.g., in the `usersWithExpensiveWishes` controllers from Figure 5. The refinement *inv* is an assertion that holds of every `row` returned by the query. The *inv* refinement enables type-level tracking of the query semantics which is essential for data-dependent policies. For example, the type

`Query` ⟨*PublicFollow*, $\lambda r \rightarrow r.level = \text{“}public\text{”}$⟩ `Wish`

describes a query on the `Wish` table, where (1) the query's results may only be accessed when the `level` is `"public"` or by the `owner`'s followers, and (2) each returned `row`'s `level` column has the value `"public"`.

Computations Standard ORMs use a monadic type to represent computations with side-effects. Haskell's `IO val` describes computations that access the DB, send email or network responses to yield a `val` value. The last pillar of STORM's API is the monadic `RIO` type that describes handlers and is parameterized with two refinements that track the authorizees and observers of the underlying computations (Figure 7):

auth: `user` → `Bool` *obs*: `user` → `Bool`

STORM ensures that in every `RIO` ⟨*auth*, *obs*⟩ `val` computation (1) *auth* is an under-approximation of the authorizees of the data the computation depends upon, and (2) *obs* is an over-approximation of the observers to whom the computation provides data. STORM then prevents leaks by ensuring that when sub-computations e_1 and e_2 are sequenced, the observers of e_2 are contained in the authorizees of e_1 (§ 3.3).

5.1 Model API

STORM's models API lets applications compose `Field`s to build a `Query` and then to execute each `Query` to obtain an `RIO` computation that provides access to DB values (Figure 8).

Query Operators Standard ORMs let the programmer write atomic queries using relational operators that test whether the value of a column equals (or disequals, exceeds, etc.) some run-time program value. For example, `Level ==. "public"` in Figure 3 denotes a `Query` that will return all `Wish` rows whose `Level` column is `"public"`. Similarly, `Price` $\geq$`. min` in Figure 5 is a `Query` that will return all `Wish` rows whose price column exceeds the value of `min`.

Compile-time enforcement poses three challenges. First, the constructed `Query`'s type must track the policy describing the set of users who are allowed access to the `Field`s upon which the query result depends. Second, the constructed `Query`'s type must capture the invariant that each row returned by the query will, in fact, have the corresponding field-value equal-to `"public"`, or greater than `min`, etc. Finally, we must achieve the above in a generic fashion that abstracts over the underlying DB column, so that the programmer can reuse the operators like `==.` across different tables.

Refined Query Operators We solve the above challenges with the types for the refined query operators *equals* `(==.)`, *not-equals* `(/=.)`, *less-than* `(<=.)`, *element-of* `(<-.)` in Figure 8. For example, the signature for the equality operator `(==.)` says that given (1) a `Field` indexed by a *pol*icy and *sel*ector, and (2) a comparison `value` satisfying a property p, the operator returns as output a `Query` with the same *pol*icy as the input `Field` where the resulting rows are guaranteed to satisfy the *inv*ariant. The crucial equality relationship is specified by the bound $FldEq\ sel\ inv\ p$ which says that

$$\forall r, fv, v.\ sel(r, fv) \Rightarrow p(v) \Rightarrow fv = v \Rightarrow inv(r) \qquad (2)$$

Recall that each `Field`'s *sel*-ector predicate characterizes the value of the `Field` in a given `row`. That is, $sel(r, fv)$ holds when the value of the `Field` in `row` r is fv. Thus, the bound (2) says that for any `row` r, the invariant $inv(r)$ holds whenever the field's value fv equals any value v that satisfies p. To get a different comparison, e.g., less-than or disequality, we need only modify the $=$ relationship in the bound to $\leq$ or $\neq$ respectively.

Query Combinators ORMs let us use combinators to build complex queries from simpler ones. For example, the query `Level ==. "public" &&. Price` $\geq$`. min` in Figure 5 returns all `Wish` rows whose `Level` is `"public"` *and* `Price` exceeds `min`.

Compile-time enforcement requires the combinators' signatures meet two goals. First, the combined `Query`'s *pol*icy predicate should be the *intersection* of the users allowed access to each sub-query. Second, the combined `Query`'s *inv*ariant should be the conjunction (for `(&&.)`) or disjunction (for `(||.)`) of the sub-query invariants.

Refined Query Combinators We achieve the above with the signatures for `(&&.)` and `(||.)` in Figure 8. The conjunction combinator `(&&.)` takes two input sub-queries of type `Query` ⟨pol_1, inv_1⟩ `row` and `Query` ⟨pol_2, inv_2⟩ `row` respectively, and returns a `Query` ⟨$pol_1 \sqcup pol_2$, *inv*⟩ `row`. The output `Query`'s *pol*icy is the *join* of the two inputs, i.e., the set of authorized users is the intersection of those allowed by pol_1 and pol_2.

```
(==.) :: (FldEq sel inv p) ⇒ Field ⟨pol, sel⟩ row val → val⟨p⟩ → Query ⟨pol, inv⟩ row
  where
    FldEq sel inv p ≐ ∀r,fv,v.sel(r,v) ⇒ p(fv) ⇒ (fv=v) ⇒ inv(r)

(&&.) :: (And inv1 inv2 inv) ⇒ Query ⟨pol1, inv1⟩ row → Query ⟨pol2, inv2⟩ row → Query ⟨pol1⊔pol2, inv⟩ row
  where
    pol1⊔pol2 ≐ λr u → pol1(r,u) ∧ pol2(r,u)
    And p q r ≐ ∀x.p(x) ⇒ q(x) ⇒ r(x)

select :: (PolAuth pol inv auth) ⇒ Query ⟨pol, inv⟩ row → RIO ⟨auth, ⊤⟩ [row⟨inv⟩]
  where
    PolAuth pol inv auth ≐ ∀r,u.inv(r) ⇒ auth(u) ⇒ pol(r,u)

project :: (PolAuth pol inv auth) ⇒ Field ⟨pol, sel⟩ row val → row⟨inv⟩ → RIO ⟨auth, ⊤⟩ val
  where
    PolAuth pol inv auth ≐ ∀r,u.inv(r) ⇒ auth(u) ⇒ pol(r,u)

join :: (Auth1 sel1 sel2 pol1 inv auth, Auth1 sel1 sel2 polq inv auth, Auth2 sel1 sel2 pol2 inv auth, SelOn sel1 sel2 on) ⇒
        Field ⟨pol1, sel1⟩ row1 val → Field ⟨pol2, sel2⟩ row2 val → Query ⟨polq, inv⟩ row1 →
        RIO ⟨auth, ⊤⟩ [(row1⟨inv⟩, row2)⟨on⟩]
  where
    SelOn sel1 sel2 on ≐ ∀r1,r2,v.sel1(r1,v) ⇒ sel2(r2,v) ⇒ on(r1,r2)
    Auth1 sel1 sel2 pol inv auth ≐ ∀r1,r2,v,u.sel1(r1,v) ⇒ sel2(r2,v) ⇒ inv(r1) ⇒ auth(u) ⇒ pol(r1,u)
    Auth2 sel1 sel2 pol inv auth ≐ ∀r1,r2,v,u.sel1(r1,v) ⇒ sel2(r2,v) ⇒ inv(r1) ⇒ auth(u) ⇒ pol(r2,u)
```

Figure 8: Selected functions from STORM's Models (ORM) API

The bound *And inv*$_1$ *inv*$_2$ *inv* states that the output `Query`'s *inv*ariant is the conjunction of that of the inputs' *inv*$_1$ and *inv*$_2$.

Example: Building Queries Let's see how STORM's API types the query `Level ==. "public" &&. Price ≥. min` from Figure 5. First, by composing the respective `Field` types for `Level` and `Price` with that of the `(==.)` operator, the type checker infers the left and right conjuncts have types

```
Query⟨⊥, λr→r.level="public"⟩ Wish
Query⟨PublicFollow, λr→r.price ≥ min⟩ Wish
```

which `(&&.)` combines to type the conjoined query as

```
Query⟨PublicFollow, λr→r.level="public" ∧ ...⟩ Wish
```

Selecting Rows Lastly, the API has functions to query the database. ORMs export a `select` function that executes a `Query` to return a list of matching `rows`. STORM's API refines the type of `select` to use the `Query`'s *pol*icy and *inv*ariant to determine: (1) the set of users authorized access to the results, and (2) the invariants of the result itself, as the data may then be used to generate subsequent queries. To this end, STORM assigns `select` the signature in Figure 8, which says that it takes as input a `Query` ⟨*pol*, *inv*⟩ `row` and returns as output a computation `RIO` ⟨*auth*, ⊤⟩ `[row`⟨*inv*⟩`]`. That is, the computation produces a list of `rows` where each row satisfies *inv*. The resulting computation's observers are the empty set ⊤ ≐ λ*u*→*false*. However, the computation's authorizees *auth* are defined by the bound *PolAuth pol inv auth* which says a user *u* is authorized to access a row *r* that satisfies the `Query` *inv*ariant *only when* that row and user satisfy the `Query` *pol*icy.

Projecting Fields In standard ORMs, the `rows` returned by `select` are opaque: a `project` operation must be used to extract the value of a given column (`Field`). STORM's API refines the type of `project` to track the authorizees of the extracted value via the signature in Figure 8, which says that `project` takes an input `Field` ⟨*pol*, *sel*⟩ `row val` and a `row`⟨*inv*⟩ and returns a computation `RIO` ⟨*auth*, ⊤⟩ `val`. Like `select` the computation has an empty set of observers (⊤). Further, the signature reuses `select`'s bound to ensure that computations authorizees *auth* are contained within those specified by `Field`'s *pol*icy.

Example: Selection and Projection Recall the `Query` in Figure 5 which looks for all the public `Wish` rows whose price exceeds `min`. As shown in the previous example, the `Query`'s *pol*icy and *inv*ariant predicates were inferred to be

$$pol \doteq PublicFollow \qquad inv \doteq \lambda r \rightarrow r.level = \text{“}public\text{”} \wedge \ldots$$

Thus, at the `select` the type checker infers the authorizees *auth* to be the set of *all* users, as the *inv*ariant implies the *pol*icy predicate. If, as in Figure 5, the `Level ==. "public"` clause was absent, the above implication would not hold, yielding a smaller set of authorizees *auth*. This would would render the handler ill-typed, as it (implicitly) leaks the sensitive `Price` value to observers outside *auth*.

Joining Tables ORMs let the user replace inefficient nested loops over multiple tables with efficient *join* operations.

STORM provides a `join` function that tracks (1) the authorizees of the sensitive data accessed by the query, and (2) the invariants of the resulting rows. STORM's `join` accounts for the policies in both tables via the signature in Figure 8. The type says that `join` takes as input the two `Field`s to join on (the `ON` clause) and a `Query` to filter the results (the `WHERE` clause), and returns a list of record pairs that satisfy the `Query`'s *inv*ariant and the *on* condition. The *on* condition is defined by the *SelOn* bound which says the condition holds for rows r_1 and r_2 if their respective join fields are equal. Further, the resulting computation's authorizees *auth* are defined by the bounds $Auth_1$ and $Auth_2$ which limits *auth* to users authorized to view the join and query fields for the subset of rows selected by the query.

Example: Join Recall the controller in Figure 4 which notifies the followers of a `user` after inefficiently computing them (`flwUsrs`) with two `select` queries: one to access the rows of the `Follower` table and one to get the corresponding rows of `User`. We can efficiently compute `flwUsrs` with a single `join`

```
let joinQ = User1==.user &&. Status==."ok"
flwUsrs  <- join User2 UserId joinQ
```

which returns a list of `(Follower, User)` pairs whose second component are `user`'s followers who can then be notified.

5.2 Controller & View API

Existing ORMs for effect-sensitive languages like Haskell encapsulate controllers and views in a monadic API to distinguish effectful DB and network computations from pure ones. STORM refines the monadic API to track the authorizees and observers of each controller computation.

Controller API The key element of the controller API is the monadic `bind` operator that sequences computations. When c_1 and c_2 are computations, of type `RIO a` and `RIO b` respectively, the expression `bind` c_1 `(λx →` c_2`)` is the computation that runs c_1, *binds* its result of type `a` to `x` *and then* runs c_2. In Haskell and similar languages, sequential blocks

```
do {x1 <- e1; ... xn <- en; e}
```

are translated to

```
bind e1 (λx1 → ... bind en (λxn → e))
```

STORM's signature for `bind` (Figure 9) ensures three properties of any sequential composition `bind` c_1 `(λx →` c_2`)`. *(Leak-freedom)* First, we ensure that c_2 does not leak sensitive information accessed in c_1. That is, we ensure that the observers obs_2 of c_2 are contained in the authorizees $auth_1$ of c_1, via the bound $auth_1 \sqsubseteq obs_2$. *(Authorizee-strengthening)* Second, the the authorizees of the sequenced computation are $auth_1 \sqcup auth_2$: the users authorized to access the data read by *both* sub-computations. *(Observer-weakening)* Finally, the the observers of the sequenced computation are $obs_1 \sqcap obs_2$: the users who are observers of *either* sub-computation.

```
return :: a → RIO ⟨⊥, ⊤⟩ a

bind :: (auth1 ⊑ obs2) ⇒
        RIO ⟨auth1, auth2⟩ a →
        (a → RIO ⟨auth2, obs2⟩ b) →
        RIO ⟨auth1 ⊔ auth2, obs1 ⊓ obs2⟩ b
  where
    auth ⊑ obs ≐ ∀u. obs(u) ⇒ auth(u)

authUser : RIO ⟨⊥, ⊤⟩ {u:User | u = sessionUser}
respond  : Text → RIO ⟨⊥, λu → u = sessionUser⟩ ()
sendMail : [user⟨p⟩] → Text → RIO ⟨⊥, p⟩ ()
```

Figure 9: Selections from STORM's Controller & View APIs

View API STORM's view API provides a function `authUser` whose signature (Figure 9) states that it returns the identity of the currently authenticated session user. Handlers can use this function to determine suitable responses to HTTP requests, e.g., by constructing and executing DB queries using `authUser` (§ 3.2). The view API has a `respond` function whose signature, shown in Figure 9, specifies that it takes `Text` or `JSON` data and sends it back to the currently authenticated *sessionUser*. Recall that the *Leak-freedom* guarantee provided by the type of `bind` ensures that whenever `respond` is used, the recipient is authorized to view the data used to construct the corresponding `Text` or `JSON` payload. Unlike previous frameworks which require potentially unsafe declassification [6], STORM's view API includes a way to `sendMail` responses to lists of users, where type checking ensures that data is disclosed per the application's centralized policy (§ 3.3).

5.3 Policies and Updates

Non-trivial applications require policies that relate rows *across* tables. (We found 9/11 of the benchmarks in our evaluation require policies that span tables § 7.1.) For example, in the `WishList` app (§ 3.3) we required that only the `owner`'s followers be allowed to read the `description` of a non-public `Wish`. The follower relationship is naturally stored in a separate `Follower` table. Hence, we must support policies that say that access is allowed if *there exists* a particular row in a different table. In the case of `WishList` a `user` can view a `descr` for a `Wish` when there exists a row in the `Follower` table whose `status` is `"ok"` that relates the viewer with the `Wish` owner. The direct way to specify such a policy is with *existentially quantified* refinement predicates, or alternatively to add a *relational join* to the set of logical operations. Unfortunately, both of these approaches take the predicate language out of the efficiently SMT decidable fragment, thus precluding automatic verification.

Witnessing Existentials with Predicates STORM allows cross-table policies by using uninterpreted predicates to provide evidence that certain rows exist in (other) tables. First, the

policy `declare`s there is a suitable relation without providing any definition for it. For example, in Figure 2 we declare a binary `follows` predicate that holds for a pair of users. Second, the policy `assert`s that each record establishes the predicate holds for the tuple of values in the record. This predicate is then added as an *invariant* that holds of every record of the corresponding table. For example, in Figure 2 we `assert` that, e.g., `OkFollows` holds for each `Follower` record. Consequently, the type checker assumes that every term of type `Follower` satisfies the invariant, and hence, provides concrete evidence that the `follows` relationship holds between users in the record's fields, *if* the `status` is `"ok"`. In this way, STORM lets us specify cross-table policies, while ensuring refinements stay decidable.

Predicates vs. Updates Predicates are timeless: once the relationship is established it holds forever. This is problematic, e.g., if the record is updated or deleted, which would require us to similarly invalidate those invariants in the code. We reconcile the tension between timeless predicates and updates by separating two goals: (1) provide security guarantees *locally* within a single controller action, and (2) reflect the effects of updates and deletions *globally* across multiple controller actions. That is, locally, we want that within a single action, a `Alice` should be able to view `Bob`'s wishes only if at *some* point during the action the `Follower` table contained a tuple `(Alice, Bob, "ok")`. However, if during an action, `Bob` revoked access, e.g., by updating the `"ok"` to `"no"`, then in *subsequent* controller actions we must deny `Alice` access.

Soundness via Monotonicity and Erasure Our uninterpreted-predicate method achieves these goals as follows. First, we impose a syntactic restriction that the predicates appear positively (i.e., not under a negation). Implicitly, the predicates are interpreted to be true if they held of *any* database snapshot during the handler action. In other words, the predicates are *monotonic*: i.e., once established, they continue to hold till the end of the action. Second, STORM's compositional design *erases* the assertions at the end of each controller action, as each action is checked in isolation starting with no assertions. That is, the assertions must be re-established by future actions by querying the database, ensuring that if one action updates the database, e.g., to revoke privileges, then accesses will be prevented in subsequent handler actions. Thus, monotonicity lets us soundly enforce the policy locally in an action, and erasure lets us propagate the effects of updates globally across actions, essentially by viewing the predicates as holding *per handler action*.

6 Verification

We establish the security guarantees of STORM in two steps. First, we implement a formally verified Labeled IO (LIO) *library* [10], whose API ensures that well-typed *clients* do not throw dynamic IFC exceptions, i.e., do not leak. Second, we use our typed LIO library to implement λ_{STORM}, a simplified *reference implementation* of the STORM API. (Unlike λ_{STORM}, the full STORM implementation supports tables with arbitrary many columns and SQL types, and implements DB queries using existing ORM libraries backed by SQL databases.) As well-typed λ_{STORM} applications are well-typed LIO clients, we are guaranteed they do not leak.

IFC with Labeled Values In LIO, `Label`s are elements from a lattice whose partial order $\sqsubseteq$ specifies *allowed* flows [10]. LIO secures data by wrapping it with `Label`s indicating the level at which it is visible

```
data Labeled a = {val: a, lbl: Label}
```

LIO enforces IFC by maintaining an *ambient* (or *current*) label l_c which keeps track of the most sensitive value read during the computation. The ambient label l_c *starts* at $\bot$ and is *updated*, i.e., monotonically increased using the labels of the sensitive data accessed during the computation. The system enforces IFC by *blocking* any output to a security level *below* l_c, as this would correspond to an (undesirable) information flow from a high (e.g., `Secret`) level to a low (e.g., `Public`) level. The undesirable flow is blocked via a dynamic IFC exception that aborts the computation.

Refined LIO Computations LIO encapsulates secure computations in a *monadic* interface that systematically creates, propagates, updates labels to enforce IFC. To this end, LIO structures computations as *label-transformers* of type `LIO a` which are functions that take as argument the *current* label l and returns the *updated* label l' and the computation's result: a value of type `a`. λ_{STORM} refines `LIO a` to implement the computation type (§ 5) as

```
type RIO ⟨auth, obs⟩ a =
  {l:Label | l ⊑ obs} → ({l':Label | l' ⊑ l ⊔ auth}, a)
```

The precondition requires that *obs* over-approximates the observers who are given access by the computation's ambient label l. The postcondition ensures that the updated label l' includes the authorizees for the computation.

Verified RIO API We make the `RIO` type abstract, and let developers write secure applications by exposing a monadic API (`bind` and `return`) extended with three operations. (1) `label l v` protects a value `v` by wrapping it with a label `l`. The operation enforces IFC by *checking* that the label `l` is not below the ambient label `lc`. If the check fails, the program `aborts` with a (dynamic) IFC error [10]. (2) `unlabel lv` takes a labeled value `lv` of type `Labeled a` and returns a computation producing the (unwrapped) `a` value. `unlabel` ensures the ambient label is updated at each sensitive data access by raising the ambient label to be at least that of `lv`'s label. (3) `downgrade l k` lets us safely unlabel `Boolean`-valued computations by taking *ceiling* label `l` and a `Boolean`-valued computation `k`, and then executes `k` *at* label `l`, updating the ambient label to `lc` $\sqcup$ `l`: Crucially, if the computation `k`'s label exceeds the ceiling `l`, then `downgrade` returns a *default* value `False`. This ensures that the `True` result is only observed for computations that safely occur *under* the ceiling `l`. We type the `RIO` API with refinements that verify

(a) the λ_{STORM} implementation of the API type-checks, and (b) well-typed clients do not throw IFC exceptions.

Policies For brevity, in λ_{STORM} we assume the DB stores a single type `Val` of primitive values and that each table has exactly two columns. In λ_{STORM}, a data-dependent *policy* is a function that maps DB rows' `Val`ues to `Label`s that protect access to each column

```
type Policy = Val → Val → Label
```

A `Spec` declares the policy for a table via one per column

```
data Spec = {p1:Policy, p2:Policy}
```

Tables A DB `Row` is a pair of labeled values

```
data Row = {f1:Labeled Val, f2:Labeled Val}
```

We define a type for `Row`s that are protected by the `Spec` s via the refinement *sat* s r which states that the row r's columns are labeled per s' policies

```
type RowS s = {r:Row | sat s r}
where sat s r ≐ ∧i∈1,2 s.pi r.f1.val r.f2.val = r.fi.lbl
```

Thus, we implement database `Table`s as a package

```
data Table = {spec: Spec, rows: [RowS spec]}
```

comprising a policy specification *spec*, and a collection of *rows* protected by labels satisfying *spec*. Thus, type checking ensures that every `Table` contains rows that are protected as mandated by the `Table`'s spec.

Verified ORM λ_{STORM} implements the models API (Figure 8) on top of our refined LIO interface in about 800 lines of code. We use `label` and `unlabel` to respectively implement `insert` and `project`. We implement `Query` using an algebraic datatype indexed with predicates that respectively represent the *pol*icy, and *inv*ariant associated with the query. Finally, we use `downgrade` to implement `select`, `update` and `join` and verify their correctness with a reference `eval` function that represents query semantics at the type-level. We use LIQUIDHASKELL to verify [65] that λ_{STORM} implements the API, which, coupled with previously established non-interference results for LIO [6, 10] proves λ_{STORM} applications do not leak.

7 Evaluation

We evaluate STORM by asking three questions: How *expressive* is STORM's policy specification mechanism? (§ 7.1) What typing *burden* does STORM's static verification place on developers? (§ 7.2) Does STORM *reduce* the code that developers need to get right in real applications? (§ 7.3)

7.1 Expressiveness

We evaluate the expressiveness of STORM's specification mechanism porting the *security policies* of nine case studies spanning four state-of-the-art approaches for centralized policy enforcement in web applications, summarized in Table 1: (i) From URFLOW [14] we ported a minimal application for storing `Secrets`; a message `Forum` with fine-grained access-control; a `Calendar` app where users share details of their schedule specifying who may learn details about it; and an anonymous `Poll` app where the creator can draft a poll and later mark it as live; (ii) From HAILS [66] we ported `GitStar`, a code hosting web platform inspired by GitHub; (iii) From JACQUELINE [5] we ported a `Conference` manager that supports designation of roles, paper submissions, assignment of reviews and review submissions; a `Course` manager that allows instructors and students to organize assignments and submissions; a `HealthRecord` Manager based on the HIPAA privacy standards; (iv) From LWEB [6] we ported `BIBIFI`, a web-site to host the "Build it, Break it, Fix it" security-oriented programming contest [67].

System	Benchmark	Model	Policy
URFLOW	`secret`	8	9
	`poll`	14	16
	`calendar`	15	29
	`gradebook`	18	24
	`forum`	19	34
JACQUELINE	`conference`	42	46
	`course`	32	11
	`health`	79	23
HAILS	`gitstar`	16	21
LWEB	`bibifi`	312	101

Table 1: Expressiveness comparison: Numbers are LOC.

URFLOW's specification language is the closest to ours: policies are specified as declarative SQL queries over the DB state, instead of STORM's logical assertions. As such, we found porting URFLOW policies to STORM to be straightforward.

JACQUELINE uses multi-faceted execution to dynamically enforce policies specified as boolean functions. We were able to express all but one policy from the JACQUELINE case studies. The sole exception was a policy from the `Conference` manager where conflicts between PC members and papers are stored in a `PaperPCConflict` table. A PC member can only see the author and the content of a paper if there is *no* conflict present in this table. Our specification language does not support policies that depend on the *absence* of rows, and we thus have to express conflicts differently. Like in URFLOW, policies in STORM are limited to those that can be proven to hold issuing simple queries to the database, including joins, but without using more complex features like grouping or sorting rows, which we leave to future work.

HAILS *and* LWEB use labels to dynamically enforce policies. The policies in their case studies directly ported over to STORM. In many situations we were able to specify the requirements in a more natural and declarative way. Specifically, HAILS and LWEB accommodate data-dependent

policies by querying the database at runtime to associate labels with meaning derived from the database state. For example, to even specify the `Follow` in the wishlist app (§ 3) one needs to query the database to check a corresponding `Follower` record exists. This is a problem. First, they duplicate DB queries as the data returned by these policy queries is often *also* relevant for the application logic. Worse, the queries may leak or fail, making it hard to reason about policy specification. In LWEB, such queries are *trusted* and written outside their declarative policy specification language. But even when they are *not* trusted (e.g., as in HAILS), exceptions in policy specification code due to failed (or unsafe) queries are hard to debug.

7.2 Effort

We evaluate the burden that STORM's static typing puts on the programmer by implementing three case-studies—`WishList` (§ 3) and the `Course` and `Conference` apps from JACQUELINE. We pick these because they have a wide range of policies that were previously thought to only be enforceable dynamically.

`WishList` (§ 3) allows users to save wishes and browse those of other users. We implemented a version with the `PublicFollow` policy which allows access to others' wishes when the wish is public or the user is a follower.

`Conference` [5] models a conference manager with a blind review process. Users can be authors of papers or PC members who write reviews. STORM enforces several policies: only a PC member should be able to view data that could reveal the identity of a reviewer; scores or the overall decision should be viewable by non-PC users only when the PC has made decisions public; even in the public stage, a paper's reviews should only be accessible to PC members or the papers' authors; some data like a paper's text should be visible to the PC or authors, but can be made public if the paper has been accepted.

`Course` [5] is a course management system with two kinds of users: students who enroll in courses, receive assignments and turn in submissions, and instructors who grade submissions and send final scores. STORM enforces policies like: only the instructor of the class or the student can view certain data like the student's final grade for the class; only the instructor or the authoring student can access an assignment submission.

Typing Annotations Static enforcement requires programmers to write some untrusted (and verified) type annotations. STORM uses the off-the-shelf LIQUIDHASKELL checker whose inference engine reduces the typing annotations needed for verification [68]. Hence, programmers need only annotate the *allows* and *gives* labels for top-level controllers with assertions describing the access provided by the controller. Many of these are *trivial* assertion where the computation (1) does not read or output sensitive data and may be typed `RIO` $\langle\bot,\top\rangle$ `a` or (2) is not composed with other sensitive computations and may be typed `RIO` $\langle\top,\bot\rangle$ `a`. The remainder express restrictions specified in policies, as exemplified by the signature for `Conference`'s `getReviews` controller:

$$p\texttt{:Paper} \rightarrow \texttt{RIO}\ \langle \lambda v \rightarrow \mathit{PcOrAuth}(v,p), \top\rangle\ \texttt{[Review]}$$

This says that user v can access the `Review`s of p only if v is on the PC or the decisions have been made and v authored p.

Quantitative Evaluation Table 2 summarizes our quantitative evaluation of the programmer effort needed for static enforcement. For each case-study, we show (1) the total lines of code of the application split across the client (where applicable), server, the DB model, and the policy specification; (2) the typing annotations required to statically verify that the server code conforms to the policy; and (3) the time taken to verify the application. Overall, our results show the programmer overhead is modest: 1 line of type (resp. non-trivial type) annotations every 19 (resp. 29) lines of code across the three case studies. We measured verification times using a commodity laptop running Arch Linux with 16GB of memory and a quad core Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor. While the results show room for improvement the times themselves were acceptable for interactive development: refinement type checking is modular and the developer focuses on one controller at a time, for which verification typically takes a few seconds.

7.3 Auditability

The ultimate proof-of-the-pudding is: *does* STORM *reduce the amount of code the developer has to get right in real web applications?* To answer this question, we built and deployed two new applications: VOLTRON and DISCO. In both applications, the code is divided into a browser-based *client* written using the VUE.JS framework [69] and a STORM *server* that handles and provides sensitive data. The client does not know anything about the security policies: all enforcement is done server-side, where the policies are used to statically restrict how data is provided in response to client requests.

VOLTRON allows instructors to simultaneously view the progress of multiple *groups* of students collaborating on in-class programming exercises. *Administrators* can create new *classes* and assign them an instructor. *Instructors* can then enroll students and assign them to groups. Each *group* is assigned a *hash* which gives them access to a text buffer that is synchronized in real-time using Google's `firebase` service [70], providing collaborative editing. While students can only access their group's buffer, instructors can view all their classes' buffers. VOLTRON has two essential policies: (1) only administrators can create classes and only instructors can enroll students to a class; and (2) a group's buffer is only accessible to the group's members and the class' instructor. We deployed VOLTRON for four month in Fall 2020 and regularly use it in two classes with about 50 and 100 students.

DISCO abbreviates *Distant Socialing*, an application that simulates the "hallway track" for facilitating social interaction in, e.g., a conference or workshop. In DISCO, an *organizer*

Application	LOC					Ver. (s)
	Server	*Models*	*Policy*	*Client*	*Annot.*	
`Conference`	644	25	57	-	43 (32)	79
`Course`	198	24	19	-	5 (1)	20
`WishList`	334	12	21	-	20 (12)	27
`Voltron`	756	32	37	1012	29 (17)	44
`Disco`	859	43	32	4630	43 (16)	120
Total	2851	140	166	5844	125 (72)	290

Table 2: Time (in seconds) to verify each application and lines taken by *Server* code, DB *Model* definitions, *Policy* specification code, *Client* code and typing *Annotations*. Non-trivial typing annotations are shown within parentheses.

can set up video chat rooms for *attendees* to join and talk to each other. Once logged in, attendees find themselves in the "Lobby" where they can see other users currently connected and view their "badges". Users can choose to "join" a room, in which case they enter a video chat with the other participants in that room. Organizers can limit the capacity of rooms and broadcast announcements to all users. Additionally, attendees can directly message each other. The majority of DISCO's policies correspond to some form of access control—e.g., operations like managing rooms and sending invitations are restricted to organizers, and personal details about individuals can only be edited by those users. We do, however, enforce two information flow policies: (1) only the recipient of a direct message is allowed to see its content; and (2) if a user has their visibility set to private, only people currently in their room can see their location.

DISCO was deployed at the Programming Languages Mentoring Workshop (PLMW) in June 2020 and at the Verification Mentoring Workshop (VMW) in July 2020. In the latter, we had about 107 registered users in all and a peak of 55 users using DISCO simultaneously. The application elicited very positive responses from users who wrote: "DISCO is great, it has been fantastic having it as a platform for social interactions at VMW!", "In my experience, DISCO worked amazingly well!", and "DISCO was among the best parts of VMW this year".

Quantitative Evaluation Table 2 compares the size of the policy specification code—that the developer has to get right—with the rest of the web application: the implementation of the server, and additionally the JavaScript clients for VOLTRON and DISCO. We find that for real applications like VOLTRON and DISCO, which require many controllers to implement the application functionality, STORM's policies account for under 4% of the server code, and under 1% if we include the client.

Discussion STORM helped discover an information flow bug in DISCO that arose due to the subtle interaction of two seemingly independent features—and would likely have gone unnoticed otherwise. First, DISCO users can set their *visibility* to private and the UI, accordingly, should not reveal to others when they join a room. Second, each DISCO room has an associated *topic* which is protected by a policy that allows users inside the room to change it. A type error alerted us to a conflict between these policies. In particular, enforcing the topic policy could implicitly reveal the location of an invisible user (violating the first policy). We designed and implemented VOLTRON without using explicit policies, and only added them afterwards. While the process of building VOLTRON took several person-months, the verification process required only minor changes to the code—including the checks that eliminated the implicit leak—and was finished in under two days. Our experience suggests developers informally consider policies when programming and structure code to facilitate verification.

8 Conclusion & Future Work

We presented the STORM framework for writing MVC-style web applications with statically enforced, data dependent security policies. STORM shows how the MVC architecture naturally lends itself to IFC, by centralizing policies as part of the model and then using a type-refined ORM API to track information flow across database queries and handler computations.

The `RIO` monad is the glue that binds together the different elements of STORM to precisely track the *effects*—each computation's authorizees and observers—needed to enforce IFC. In principle, it should be possible to integrate our approach to any language that supports similar fine-grained effect tracking. On the flip side, however, a limitation of our design is that programmers have to structure their controllers in the restricted `RIO` monad which limits the effects available to them. Our evaluation shows how a broad range of effects (database queries, HTTP requests, emails, random number generation) can be integrated into the `RIO` monad which sufficed to build real web applications. It would be interesting to investigate how to securely integrate other classes of effects (e.g., exceptions which are historically leaky).

Another limitation apparent from our models API is that it takes some toil to extend STORM to support DB operations like `select` or `join`, which restricts the DB queries the developer can write. In future work, it would be valuable to see how to support more expressive queries by designing a way to systematically and automatically refine an ORM library that supports a large fragment of SQL.

Acknowledgments

Many thanks to the reviewers and Geoff Voelker for providing excellent feedback on early drafts of this work. We are especially grateful to our shepherd Jon Howell for spending hours to help illuminate murky passages in the exposition. This work was supported by the NSF under grant no. CNS-1514435, CCF-1943623, CCF-1918573, CCF-1911213, CNS-2048262, and by generous gifts from Microsoft Research and Cisco.

References

[1] T. Bar, "Notifying our developer ecosystem about a photo api bug," 2018, https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug/.

[2] The OWASP Foundation, "OWASP Top Ten," 2020, https://owasp.org/www-project-top-ten/.

[3] ——, "Top 10 2013," 2013, https://wiki.owasp.org/index.php/Top_10_2013-Top_10.

[4] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," *Journal of Computer Security*, vol. 25, 2017.

[5] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *PLDI*. New York, NY, USA: ACM, 2016, pp. 631–647.

[6] J. Parker, N. Vazou, and M. Hicks, "Lweb: information flow security for multi-tier web applications," *PACMPL*, vol. 3, no. POPL, pp. 75:1–75:30, 2019. [Online]. Available: https://doi.org/10.1145/3290388

[7] T. Armerding, "The IoT: Too big (and buggy) to patch?" 2018, https://www.synopsys.com/blogs/software-security/iot-big-buggy-patch/.

[8] D. Stefan, "LambdaChair policy," 2014, https://github.com/deian/lambdachair/blob/master/LambdaChair/Policy.hs.

[9] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones, "Refinement types for haskell," in *ICFP*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 269–282. [Online]. Available: https://doi.org/10.1145/2628136.2628161

[10] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *J. Funct. Program.*, vol. 27, p. e5, 2017. [Online]. Available: https://doi.org/10.1017/S0956796816000241

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web application via automatic partitioning," in *SOSP*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 31–44. [Online]. Available: https://doi.org/10.1145/1294261.1294265

[12] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *SIGMOD*, 2009, pp. 269–282.

[13] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," *SOSP*, 2009.

[14] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *OSDI*, 2010.

[15] D. A. Schultz and B. Liskov, "IFDB: decentralized information flow control for databases," in *Eurosys*, Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, Eds. ACM, 2013, pp. 43–56. [Online]. Available: https://doi.org/10.1145/2465351.2465357

[16] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 79–94.

[17] A. Sabelfeld and A. Myers, "Language-based information-flow security," 2003. [Online]. Available: citeseer.ist.psu.edu/article/sabelfeld03languagebased.html

[18] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2009, pp. 352–365.

[19] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, "Faceted execution of policy-agnostic programs," in *PLAS*, 2013.

[20] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical Fine-grained Decentralized Information Flow Control," in *PLDI*. ACM, 2009, pp. 63–74.

[21] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your ifcexception are belong to us," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 3–17.

[22] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," 2012.

[23] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A better facet of dynamic information flow control," in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 731–739.

[24] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 109–124.

[25] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[26] F. Pottier and V. Simonet, "Information flow inference for ml," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 319–330.

[27] N. Broberg, B. van Delft, and D. Sands, "Paragon–practical programming with information flow control," *Journal of Computer Security*, vol. 25, no. 4-5, pp. 323–365, 2017.

[28] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 369–383.

[29] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," vol. 20, no. 7, 1977.

[30] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL*, 1999.

[31] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: a platform for secure distributed computation and storage," in *SOSP*. ACM, 2009.

[32] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in haskell," in *ICFP*, 2015, pp. 289–301.

[33] V. Rajani and D. Garg, "On the expressiveness and semantics of information flow types," *Journal of Computer Security*, no. Preprint, pp. 1–28, 2019.

[34] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From fine-to coarse-grained dynamic information flow control and back," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[35] B. Montagu, B. C. Pierce, and R. Pollack, "A theory of information-flow labels," in *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, 2013, pp. 3–17.

[36] L. Lourenço and L. Caires, "Information flow analysis for valued-indexed data security compartments," in *Trustworthy Global Computing*. Springer, 2014, pp. 180–198.

[37] ——, "Dependent information flow types," in *Proceedings of the 42nd Symposium on Principles of Programming Languages*. ACM, 2015, pp. 317–328.

[38] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in Fine," in *ESOP*, 2010.

[39] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *ICFP*, 2011.

[40] S. Chong, K. Vikram, and A. C. Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *USENIX Security*, 2007.

[41] E. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. Schneider, "Logical attestation: an authorization architecture for trustworthy computing," in *SOSP*, 2011, pp. 249–264.

[42] P. Li and S. Zdancewic, "Encoding information flow in haskell," in *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, p. 16. [Online]. Available: https://doi.org/10.1109/CSFW.2006.13

[43] D. Schoepe, D. Hedin, and A. Sabelfeld, "Selinq: tracking information across application-database boundaries," in *ICFP*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 25–38. [Online]. Available: https://doi.org/10.1145/2628136.2628151

[44] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Liquid information flow control," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, pp. 105:1–105:30, 2020. [Online]. Available: https://doi.org/10.1145/3408987

[45] M. N. Krohn, "Building secure high-performance web services with OKWS." in *USENIX Annual Technical Conference (ATC), General Track*, Jun. 2004.

[46] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner, "Diesel: applying privilege separation to database access," in *Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 416–422.

[47] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, and T. Anderson, "Radiatus: Strong user isolation for scalable web applications," University of Washington, Tech. Rep., 2014.

[48] A. Blankstein and M. J. Freedman, "Automating isolation and least privilege in web services," in *Security and Privacy*. IEEE, 2014, pp. 133–148.

[49] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *NDSS*, 2018.

[50] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," in *USENIX Security Symposium*. USENIX, 2017, pp. 1463–1479.

[51] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building web applications on top of encrypted data using Mylar," in *NSDI*, 2014, pp. 157–172.

[52] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-end integrity protection for web applications," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 895–913.

[53] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "Shadowcrypt: Encrypted web applications for everyone," in *CCS*, 2014, pp. 1028–1039.

[54] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. M. Eyers, B. Shand, and P. R. Pietzuch, "Flowwatcher: Defending against data disclosure vulnerabilities in web applications," in *CCS*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 603–615. [Online]. Available: https://doi.org/10.1145/2810103.2813639

[55] F. Wang, R. Ko, and J. Mickens, "Riverbed: Enforcing user-defined privacy constraints in distributed web services," in *NSDI*. Boston, MA: USENIX, 2019, pp. 615–630.

[56] R. L. Constable and S. F. Smith, "Partial objects in constructive type theory," in *LICS*, 1987.

[57] J. Rushby, S. Owre, and N. Shankar, "Subtypes for specifications: Predicate subtyping in PVS," *IEEE TSE*, 1998.

[58] P. Rondon, M. Kawaguchi, and R. Jhala, "Liquid types," in *PLDI*, 2008.

[59] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis, "Refinement types for secure implementations," in *CSF*, 2008.

[60] J. Hamza, N. Voirol, and V. Kuncak, "System FR: formalized foundations for the stainless verifier," *PACMPL*, vol. 3, no. OOPSLA, pp. 166:1–166:30, 2019. [Online]. Available: https://doi.org/10.1145/3360592

[61] N. Vazou, A. Bakst, and R. Jhala, "Bounded refinement types," in *ICFP*, K. Fisher and J. H. Reppy, Eds. ACM, 2015, pp. 48–61. [Online]. Available: https://doi.org/10.1145/2784731.2784745

[62] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *POPL*. ACM, 1999, pp. 147–160.

[63] M. Snoyman and G. Weber, https://www.yesodweb.com/book/persistent.

[64] M. Torgersen, "Querying in c#: how language integrated query (LINQ) works," in *OOPSLA*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 852–853. [Online]. Available: https://doi.org/10.1145/1297846.1297922

[65] R. Jhala and N. Lehmann, github.com/storm-framework/core.

[66] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *OSDI*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 47–60. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin

[67] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build it, break it, fix it: Contesting secure development," in *CCS*, 2016, pp. 690–703.

[68] B. Cosman and R. Jhala, "Local refinement typing," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3110270

[69] "Vue.js: The progressive javascript framework," https://vuejs.org/.

[70] "Firebase realtime database," https://firebase.google.com/docs/database.

A Artifact Appendix

Abstract

Our artifact contains (a snapshot of) the source code for the implementation of STORM from § 5, the formally verified reference implementation λ_{STORM} described in § 6), the various policies, case-studies and applications used in our evaluation § 7.

Scope

The artifact provides a way to reproduce the results in the paper. First, we provide examples of how a programmer might write insecure code that fails to respect particular policies, as described in Section 3 and show how those mistakes are caught by refinement type checking. Next, the code shows how STORM is implemented on top of existing ORM and networking libraries as described in § 5. Further, the artifact contains the verified reference implementation of λ_{STORM} from Section 6 which shows how the API can be implemented on top of an LIO interface. Finally, addition, to the source code described above we include the various scripts used to compile the applications and measure the verification time and code annotation overheads that we report in Section 7.

Contents

The artifact comprises the following sub-directories and files: `storm-core`—the source for the verified reference implementation λ_{STORM} (§ 6); `models`—the ported policies from the expressiveness benchmarks (§ 7.1); `case-studies`—the source for the ported case-studies (§ 7.2); `disco` and `voltron`—the source for the end-to-end applications (§ 7.3); and `fig9.py`—the script used to generate Table 2. Each sub-directory contains a manifest file that links to the `github` commits for STORM and LIQUIDHASKELL that are needed to compile the application.

Hosting

You can obtain the artifact from `github` by running `git clone --recursive` on the repository `https://github.com/storm-framework/artifact` It suffices to use the `main` branch, specifically, commit `3eb138ab5145e688504eff71c669c6570701e10b`.

Requirements

You can run the artifact on any machine computer running Linux or MacOS after installing the following software. The artifact requires `python 3.7` and the following dependencies. (1) `stack v2.5.1` which can be installed by following these instructions[1]; (2) `z3 v4.8.8` which can be installed by downloading the binary [2]. You can ignore the shared libraries and bindings for Java and Python; just download and place a suitable `z3` binary somewhere in your `PATH`. (3) `tokei v12.1.2` which is used to count lines of code [3]. Familiarity with the `stack` build system for Haskell would be useful to evaluate the artifact but it is not necessary.

λ_{STORM} Implementation (§ 6)

Directory `storm-core` has the source for the verified reference implementation λ_{STORM} from § 6. To verify, run `cd storm-core && stack build`.

Policies (§ 7.1)

The code in `models/` contains the policies ported to evaluate expressiveness as described in § 7.1. This directory does not contain verifiable code, only the ported models files. The models files are grouped by the original tool they were taken from, e.g., the models file for the `Calendar` application in URFLOW is in `models/src/UrWeb/Calendar/Model.storm`.

[1] `https://docs.haskellstack.org/en/stable/README`
[2] `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.8`
[3] `https://github.com/XAMPPRocky/tokei#installation`

Case Studies (§ 7.2)

The case studies used to evaluate the burden STORM puts on programmers as described in § 7.2 are in `case-studies`. There is a `stack` project for each case study.

Verify the Code To verify one of the case studies go to the corresponding directory and build the project. For example, to verify the `WishList` application run `cd case-studies/wishlist && stack build`.

Breaking the Code To check how STORM catches leaks open `case-studies/wishlist/src/Controllers/Wish.hs`. The function `getWishData` at line 156 extracts the information out of a `Wish`. The query between lines 164 and 171 checks if the viewer is friends with the owner of the wish. Remove the check `frienshipStatus ==. "accepted"` from the query, i.e., the query should look like:

```
friends  <- selectFirst
  ( friendshipUser1' ==. owner &&:
    friendshipUser2' ==. viewerId )
```

Then verify by running `stack build`. Forgetting to check if the friendship is `"accepted"` causes a leak as the viewer may not be friends with the `Wish` owner, yielding an error:

```
    |
173 | level == "friends" →
    |   project wishDescription' wish
    |           ^^^^^^^^^^^^^^^^
```

Automation Evaluation (Fig 2)

To produce the count of lines of code in 2 run `python3 fig9.py`

Application: Disco (§ 7.3)

Verify the Code To verify `Disco`'s server code is leak free run `cd disco/server && stack build`

Break the Code Open the file `disco/server/src/Controllers/Room`. The function `updateTopic` on line 36 implements the functionality that allows a user to update a room's topic. If not done carefully, this operation may produce a subtle information flow bug as described in the discussion of § 7.3. `Line 42` checks that the user's visibility is set to `"public"` and only then allows them to update the topic. Update lines 42 to 50 to

```
Just roomId → do
  UpdateTopicReq {..} <- decodeBody
  validateTopic updateTopicReqTopic
  _ <- updateWhere
        (roomId' ==. roomId)
        (roomTopic' `assign` updateTopicReqTopic)
  room  <- selectFirstOr notFoundJSON
            (roomId' ==. roomId)
  roomData <- extractRoomData room
  respondJSON status200 roomData
Nothing → respondError status403 Nothing
```

and run `stack build`. Forgetting to check if the visibility is set to public produces an error when accessing the user's current room as the information may be leaked. You should see:

```
**** LIQUID: UNSAFE **********************
src/Controllers/Room.hs:39:23: error:
...
   |
39 |   userRoom <- project userRoom' viewer
   |                       ^^^^^^^^^
```

Application: Voltron (§ 7.3)

Verify the Code You can verify the code by `cd voltron/server && stack build`

Break the Code Open the file `voltron/server/src/Controllers/Class.hs` The function `addRoster` at line 102 implements the functionality to enroll a list of students to a class. This operation is restricted to instructors of the class which is checked by the query in lines 109 and 110. Removing the clause `classInstructor' ==. instrId` so the query reads:

```
cls <- selectFirstOr
        (errorResponse status403 Nothing)
        (className' ==. rosterClass)
```

produces an error as the user does not have enough permissions:

```
**** LIQUID: UNSAFE **********************
src/Controllers/Class.hs:113:19: error:
...
   |
113| mapT (addGroup clsId) (rosterGroups r)
   |                ^^^^^

src/Controllers/Class.hs:114:19: error:
...
   |
114| mapT (addEnroll clsId) (rosterEnrolls r)
   |                 ^^^^^
```

Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation

Shaghayegh Mardani[1], Ayush Goel[2], Ronny Ko[3], Harsha V. Madhyastha[2], Ravi Netravali[4]
[1]*UCLA*, [2]*University of Michigan*, [3]*Harvard University*, [4]*Princeton University*

Abstract

Web pages today commonly include large amounts of JavaScript code in order to offer users a dynamic experience. These scripts often make pages slow to load, partly due to a fundamental inefficiency in how browsers process JavaScript content: browsers make it easy for web developers to reason about page state by serially executing all scripts on any frame in a page, but as a result, fail to leverage the multiple CPU cores that are readily available even on low-end phones.

In this paper, we show how to address this inefficiency without requiring pages to be rewritten or browsers to be modified. The key to our solution, Horcrux, is to account for the non-determinism intrinsic to web page loads and the constraints placed by the browser's API for parallelism. Horcrux-compliant web servers perform offline analysis of all the JavaScript code on any frame they serve to conservatively identify, for every JavaScript function, the union of the page state that the function could access across all loads of that page. Horcrux's JavaScript scheduler then uses this information to judiciously parallelize JavaScript execution on the client-side so that the end-state is identical to that of a serial execution, while minimizing coordination and offloading overheads. Across a wide range of pages, phones, and mobile networks covering web workloads in both developed and emerging regions, Horcrux reduces median browser computation delays by 31-44% and page load times by 18-37%.

1 Introduction

Despite accounting for over half of all global web traffic [28, 30, 76], mobile browsing in the wild continues to operate far slower than what users can endure [17, 18, 34], with page loads often taking upwards of 10 seconds [14, 79]. Since users are more likely to abandon pages that are slow to load [39], the current sub-optimal state of mobile web performance negatively impacts not only user experience, but also the revenue of content providers [31].

A key contributor to slow page loads on mobile devices is the computation that browsers must perform to load a page [85, 62, 64, 22], most of which is accounted for by the execution of JavaScript code (§2). Numerous solutions have attempted to reduce the amount of necessary client-side computation, either by requiring developers to manually rewrite pages [37] or by having clients offload page load computations to more powerful servers [68, 67, 71, 13, 64, 32]. However, solutions of the former class come at the expense of manual effort and page functionality, while those in the latter class are largely unviable in practice (§7). Offloading to proxies [68, 67, 71] is infeasible in today's HTTPS-by-default web, while systems [64] in which origin servers return post-processed pages that elide computations risk compromising correctness since servers lack visibility into client-side state (e.g., localStorage) that can affect control flow in a page load.

We pursue an alternative and complementary approach: instead of attempting to *reduce* the amount of computation that web clients must perform, we seek to execute the necessary computation on client devices *more efficiently*. In particular, we observe that there exists a fundamental inefficiency in the computation model that browsers employ (§2). To simplify page development, JavaScript execution is single-threaded [65, 64], and worse, JavaScript and rendering tasks are forced to share a single "main" thread per frame in a page [38]. Consequently, browsers are unable to take advantage of the growing number of CPU cores available on popular phones in both developed and emerging regions [24, 25]. This inefficiency will only worsen as, due to energy constraints, increased core counts have become the main source of compute resource improvements on phones [77, 35].

A natural solution to this inefficiency is to parallelize JavaScript computations across a device's available cores. Browsers have included support for pages to spin up parallel JavaScript computation threads in the form of Web Workers [55, 2] for over 8 years now. Yet, only a handful of the top 1,000 sites use Workers on their landing pages, largely due to the challenges of writing efficient, concurrent code [15, 45]. These challenges manifest in two ways for the web.

- Determining *which* JavaScript executions on a page frame can be safely parallelized requires a precise understanding of the page state accessed by every script, due to the language's lack of synchronization mechanisms (e.g., locks). Placing the onus of this task on web developers [53, 6] is impractical, while reliance on browsers to speculatively make parallelism decisions [21, 56] is inefficient (§6.3).
- *How* to efficiently execute scripts in parallel is also not straightforward due to the restrictions that browsers impose on Workers. In particular, they cannot access a page's JavaScript heap or DOM tree, and coordinating with the main thread, which has these privileges, adds overheads.

Our goal is to automatically parallelize JavaScript computations for *legacy pages* on *unmodified browsers*, thereby addressing the cognitive and operational overheads involved in explicitly making parallelism decisions. Our solution, **Horcrux**, achieves these goals by employing a judicious split

between clients and servers, hence preserving HTTPS' end-to-end content integrity and privacy guarantees [67]. Servers perform the heavy lifting of finding parallelism opportunities and embed that information in their pages. Clients then run a JavaScript scheduler that efficiently manages parallelism using runtime information that servers lack, i.e., number of available threads, control flows taken in the current load. Three primary insights guide our design of Horcrux.

First, we ensure that any introduced parallelism preserves the final page state that developers expected when they wrote the page. For this, Horcrux forces computations that exhibit state dependencies (e.g., a read-write dependency on a global variable) to run serially in an order that matches the legacy load, while allowing other computations to run in parallel. Key to enabling this is Horcrux's offline use of concolic execution [48, 36, 80] on servers to explore all possible control flows on a page and identify all state that each JavaScript function *might* access, irrespective of how client-side nondeterminism could affect any particular load.

Second, Horcrux minimizes client-side coordination overheads by carefully partitioning responsibilities between the main browser thread and the Web Workers it spawns. Horcrux reserves the main thread for coordinating Worker offloads, managing global page state, and running DOM computations (which Workers cannot); all other computations are offloaded. This yields two benefits. First, scripts typically interleave computations that can and cannot be offloaded; Horcrux maximally parallelizes the former while carefully mediating the latter. Second, by keeping the main thread largely idle, Horcrux quickly adapts the parallelization schedule to the runtimes of JavaScript computations, offloading the next computation as soon as a Worker becomes available.

Third, the granularity at which Horcrux parallelizes JavaScript execution is crucial with respect to overheads and potential parallelism. A natural solution would be to offload the invocations of JavaScript functions, which account for 94% of JavaScript source code. However, the sheer number of invocations in a typical page load makes this too costly. Instead, we observe that functions are typically invoked hierarchically (i.e., nested functions), with significant state sharing within a hierarchy, but less across them. Therefore, Horcrux offloads at the granularity of root function invocations, or the root of each hierarchy along with its nested constituents. Compared to per-function offloading, this requires 4× fewer offloads while achieving 73% of the potential speedup.

We evaluated Horcrux using over 650 diverse pages, live mobile networks (LTE and WiFi), and three phones, that collectively represent browsing scenarios in both developed and emerging regions. Our experiments across these conditions reveal that Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index speedups of 18-29% and 24-37%, respectively. Further, Horcrux's median benefits are 1.3-2.1× larger than prior compute-focused web accelerators, and 1.4-2.1× more than (complementary) network optimizations.

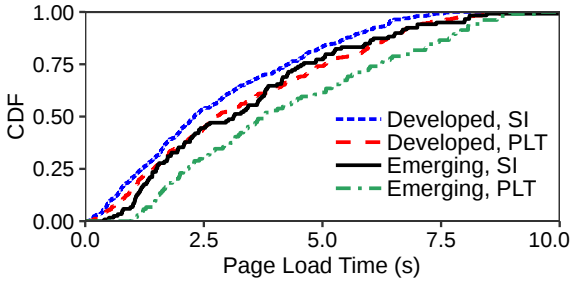


Figure 1: **Load times often exceed user tolerance levels (3 seconds) even when all network delays are removed.**

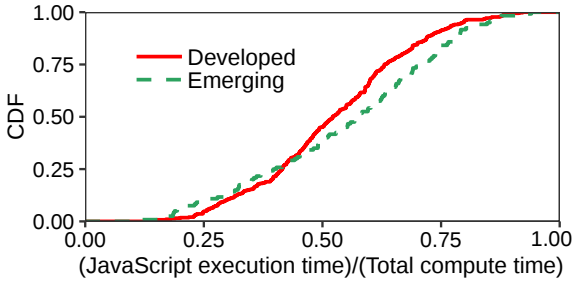


Figure 2: **JavaScript's role in browser computation delays.**

Taken together, our results highlight that, despite being written for a serial browser computation model, existing pages are surprisingly ripe with parallelization opportunities. Horcrux shows how such opportunities can be exploited under the hood, without having developers manually rewrite their pages. Source code and datasets for Horcrux are available at https://github.com/ShaghayeghMrdn/horcrux-osdi21.

2 Motivation and Background

Numerous studies have reported that client-side (browser) computation is a significant contributor to poor mobile web performance [85, 62, 79, 64]. We reproduce this finding below (§2.1), present measurements to elucidate why such delays are so pronounced (§2.1), and trace the origins for this poor performance to the computation model used by browsers today (§2.2). Our experimental setup (§6.1) covers web workloads in both developed and emerging markets by considering popular pages in the US and Pakistan and loading those pages on common phones in each region. Pages in the emerging region generally involve less JavaScript code, but are loaded on phones with fewer compute resources.

2.1 Web Computation Delays

Computation delays are significant. To quantify the computation delays in page loads, we replayed each page locally, without any network emulation, i.e., all object fetches took ≈ 0 ms. As shown in Figure 1, even without network delays, popular pages in developed and emerging markets have median load times of 2.7 and 3.8 seconds, respectively. Worse, 48% and 63% of pages require more than the 3 second load

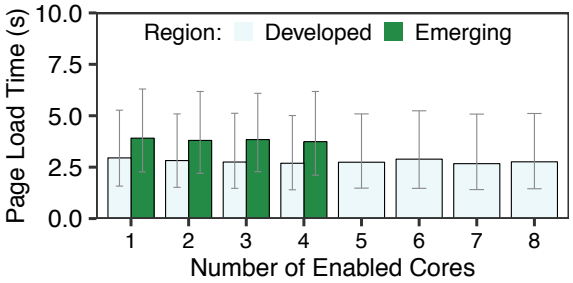


Figure 3: **Additional CPU cores have minimal impact on load times. The developed and emerging region phones have 8 and 4 cores. Bars list medians, with error bars for 25-75th percentiles.**

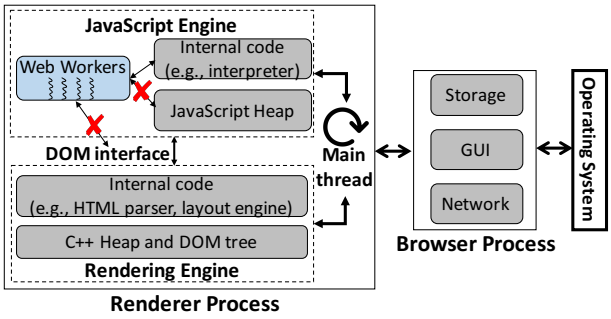


Figure 4: **Computation model for Chromium browsers.**

times that users tolerate [3]. These intolerable delays persist with metrics evaluating page visibility (i.e., Speed Index [40]), with 39% and 52% of pages in the developed and emerging regions taking more than 3 seconds to fully render.

JavaScript execution is the main culprit. To break down these high computation delays, we analyzed data from the browser's in-built profiler which lists the time spent performing various browser tasks including JavaScript execution, HTML parsing, rendering, and so on. Figure 2 illustrates our finding that JavaScript computation is the primary contributor, accounting for 52% and 58% of overall computation time for the median page in the two settings.

Browsers make poor use of CPU cores. Computation resources on mobile phones have globally increased in recent years, with improvements in both CPU clock speeds and total CPU cores. However, due to the energy constraints on mobile devices, increased core counts have been (and likely will continue to be) the primary source of improvements [77, 35]. For example, since their inception in 2016, Google's Pixel smartphones (our developed region phone) have improved clock speeds from 1.88 GHz to 2.15 GHz, while doubling the number of CPU cores (from 4 to 8). Similarly, the popular Redmi A series in India and Pakistan [4] (our emerging market phone) observed the same doubling in CPU cores (2 to 4) during that time period, while seeing only a modest clock speed improvement from 1.4 GHz to 1.75 GHz.

Unfortunately, although browsers can automatically benefit from clock speed improvements, we find that they fail to leverage available cores. To illustrate this, we iteratively disabled CPU cores on the phones in each setting and observed the impact on page load times. As shown in Figure 3, additional CPU cores yield minimal load time improvements, e.g., going from 1 to 8 cores on the Pixel 3 resulted in only a 8% speedup for the median page.

2.2 Browser Computation Model

To determine the origin of these computation inefficiencies, we must consider the computation model that browsers use today. Our discussion will be based on the Chromium framework [38], which powers the Chrome, Brave, Opera, and Edge browsers that account for 70% of the global market share [8, 5]. Figure 4 depicts Chromium's multi-process architecture. We focus on the *renderer process* which houses the Rendering and JavaScript engines, and thus embeds the core functionality for parsing and rendering pages.

The Rendering engine parses HTML code, issues fetches for referenced files (e.g., CSS, JavaScript, images), applies CSS styles, and renders content to the screen. During the HTML parse, the rendering engine builds a native representation of the HTML tree called the *DOM tree*, which contains a node per HTML tag. As the DOM tree is updated, the rendering engine recomputes a layout tree specifying on-screen positions for page content, and issues the corresponding paint updates to the browser process.

The JavaScript engine is responsible for parsing and interpreting JavaScript code specified in HTML `<script>` tags, either as inline code or referenced external files. During the page load, the JavaScript engine maintains a managed heap which stores both custom, page-defined JavaScript state and native JavaScript objects (e.g., `Dates` and `RegExps`). JavaScript code can initiate network fetches via the browser process (e.g., using `XMLHttpRequests`), and can also access the rendering engine's DOM tree (to update the UI) using the DOM interface. The DOM interface provides native methods for adding/removing nodes and altering node attributes; DOM nodes accessed via these methods are represented as native objects on the JavaScript heap.

The problem: single-threaded execution. JavaScript execution is single-threaded and non-preemptive [65, 64]. Worse, within a renderer process, all tasks across the two engines are coordinated to run on a single thread, called the *main thread*.[1] This lack of parallelism largely explains the poor use of CPU cores in §2.1. A primary reason for this suboptimal computation model is that the JavaScript language and DOM data structure (shared between the two engines) lack synchronization mechanisms (e.g., locks) to enable safe concurrency. Adding thread safety is feasible, but browsers have continually opted for a serial-access model to simplify page development. Browsers do create a separate renderer process per cross-domain `iframe` in a page (as per

[1] Some Chromium implementations move final-stage rendering tasks to raster/composite threads that create bitmaps of tiles to paint to the screen.

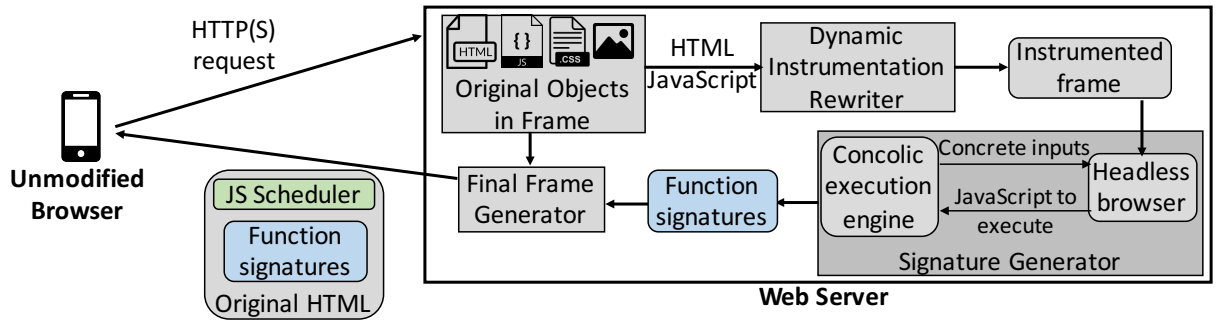


Figure 5: **Overview of the generation and fetch of each frame's top-level HTML file with Horcrux. Offline, servers collect a frame's files (both local and third-party), generate comprehensive state access signatures for each JavaScript function, and embed that information in their pages. Parallelism decisions are made online by unmodified client browsers using the Horcrux JavaScript scheduler, which manages computation offloads to Web Workers and maintains the page's JavaScript heap and DOM tree.**

# of Cores	% Speedup in Total JavaScript Runtime
2 cores	54%
4 cores	79%
8 cores	88%

Table 1: **Potential parallelism speedups with varying numbers of cores. Results list median speedups in total time to run all JavaScript computations per page in the developed region.**

the Same-origin content sharing policy [7]). However, for the median page in the Alexa top 10,000, the top-level frame accounts for 100% of JavaScript execution delays.

In summary, despite benefits regarding simplified page development, the single-threaded execution model that browsers impose results in significant underutilization of mobile device CPU cores, inflated computation delays, and degraded page load times. We expect this negative interaction to persist (and worsen) moving forward given the steady and unrelenting increase in the number of JavaScript bytes included in mobile web pages, e.g., there has been a 680% increase over the last 10 years [44].

3 Overview

Given the results in §2, a natural solution to alleviate client-side computation delays in mobile page loads is to *parallelize* JavaScript execution across a device's available CPU cores. However, not all workloads are amenable to parallel execution. In particular, we face the restriction that any introduced parallelism should preserve the page load behavior (and the final page state) that developers expected when writing their legacy pages—we call this property *safety*.

3.1 Potential Benefits

To estimate the potential benefits of parallelism with legacy pages, we analyzed the JavaScript code for each page in our corpus; in this section, we focus on page loads representative of those in developed regions, and we show later in §6 that similar benefits are achievable for page loads in emerging markets. Since JavaScript functions account for 94% of the JavaScript source code on the median page, our analysis operates at the granularity of functions, i.e., when splitting computations on a page across CPU cores, all code within a function runs sequentially on the same core. For completeness, we convert all code outside of functions into anonymous functions. For each function, we recorded both its runtime in a single load, as well as all accesses that it made to page state (in the JavaScript heap or DOM tree, as described below) in that load; §4.1.1 details the data collection process.

Using these logs, we estimated an *upper bound* on parallelism benefits by maximally packing function invocations to available cores and recording the resulting end-to-end computation times. To ensure safety (defined above), our analysis respects two constraints: 1) functions can run in parallel if they access disjoint subsets of page state or only read the same state, and 2) functions that exhibit state dependencies (i.e., access the same state and at least one writes to that state) execute in an order matching that in the legacy page load.

As shown in Table 1, legacy pages are highly amenable to safely reaping parallelism speedups. For example, distributing computation across 4 cores could bring a 75% reduction in the total time required to complete all JavaScript computations on the median page. These resources are now common in both developed and emerging markets [24].

3.2 Goals and Approach

To realize these benefits in practice, we seek an approach that minimizes the bar to adoption. As a result, requiring developers to rewrite pages [53] is a non-starter, given the complexities involved in manually reasoning about the impact of concurrently running portions of the JavaScript code on a page. Moreover, approaches that only require changes to browsers would either have to speculatively parallelize code [21, 56] or perform client-side analysis of JavaScript code (akin to the analysis that informed our estimated benefits above). As we show in §6.3, the overheads imposed by either strategy make them untenable, especially on energy-constrained phones.

Therefore, we pursue an approach which can safely parallelize JavaScript execution on *legacy* web pages with *unmodified* browsers. As shown in Figure 5, our solution, Hor-

Delay type	0.5 KB	1 KB	100 KB	1 MB
Startup	128 ms	155 ms	237 ms	317 ms
Value I/O	0 ms	1 ms	1 ms	7 ms

Table 2: **Web Worker overheads for different sizes of state transfers, i.e., source code size for startup delays and JavaScript object size for I/O delays.**

crux, only necessitates server-side changes that do not require developer participation to rewrite pages. Web servers perform the *expensive* task of tracking the state accessed (in the JavaScript heap or DOM tree) by every JavaScript function in a page frame, and include this information in that frame in the form of per-function *signatures*. Servers also embed a JavaScript (JS) scheduler library in the frames they serve, which enables unmodified client browsers to perform the *cheap* task of managing parallelism using function signatures obtained from servers. Dynamically determining the parallelization schedule at the client helps Horcrux account for information only available at runtime, e.g., the number of available threads and the control flows in the current load.

3.3 Challenges

The key building block in Horcrux is the widespread support in browsers for the Web Workers API [55], which allows the JavaScript engine to employ additional computation threads (Figure 4), as specified by a page's source code. Using Web Workers to parallelize JavaScript execution, however, presents numerous challenges that complicate achieving the idealistic parallelism benefits from above.

1. **Ensuring correctness.** Determining what JavaScript code can be *safely* offloaded requires a comprehensive understanding of how that code will access JavaScript heap and DOM state *in the current page load*. This, in turn, depends on the traversed control flows, which can vary due to both client-side (e.g., `Math.Random`) and server-side (e.g., cookies) sources of nondeterminism in JavaScript execution [59]. Missed state accesses can lead to dependency violations and broken pages.
2. **Constrained API.** Web Workers impose restricted computation models in two ways. First, due to the lack of synchronization mechanisms in JavaScript, Workers cannot access the JavaScript heap, and instead can only operate on values explicitly passed in by the browser's main thread (via `postMessages`). Thus, offloading computations to a Worker requires knowledge of precisely what state is required for those computations. Workers must then communicate computation results back to the main thread, which applies the results to the heap. Second, regardless of the state passed in, Workers cannot perform any DOM computations, including invocations of native DOM methods or operations on live DOM nodes referenced in the heap. We note that Workers can spawn and manage other Workers, but still must rely on the main thread for access to any global state.
3. **Offloading costs.** Lastly, operating Web Workers is not free. Instead, as shown in Table 2, spinning up a Web Worker can take over 100 ms, even for small amounts of source code being passed in. Pass-by-value I/O adds an additional several milliseconds, depending on the size of the transferred state. Moreover, JavaScript execution is non-preemptive; so, Workers that finish their tasks may go idle for long durations if the thread responsible for assigning them more tasks is busy.

We posit that, it is for these reasons that only a handful of the Alexa top 1,000 pages use Workers, despite support by commodity browsers for over 8 years. We next describe how Horcrux overcomes these issues to automatically parallelize JavaScript execution for legacy pages.

4 DESIGN

In designing Horcrux, we primarily need to answer two questions: 1) how to determine which JavaScript functions on a page can be executed in parallel without compromising correctness?, and 2) how to realize parallel execution at low overhead despite constraints placed by the browser's API? We present our solutions to these issues by separately describing server-side and client-side operation in Horcrux. Table 3 summarizes the main techniques underlying our design.

4.1 Server-side Operation

The goal of Horcrux's server-side component is to annotate page frames with per-function signatures that list the state that each function *might* access. Operating at a frame level, rather than at the granularity of entire pages, is in accordance with the browser's content sharing model [7, 64]. As in prior web optimizations that involve page alterations [63, 64, 54], Horcrux assumes access to a frame's source files. These files can be quickly collected either using a headless browser[2] or via integration into content management systems (for local files) [27, 89]. Source file collection and signature generation is retriggered based on hooks that many content management systems fire any time a (local) file-altering change is pushed [27, 89, 47, 54], e.g., for A/B testing; we discuss third-party content changes and personalization in §4.2.2.

4.1.1 Generating signatures

Since web servers cannot precisely predict the control flows that will arise in any particular page load (e.g., due to client-side nondeterminism), each function's signature must conservatively list *all possible* state accesses for that function. For this reason, we cannot directly apply recent web dependency tracking tools [63, 64] that rely purely on dynamic analysis to track data flows *in a given page load*. At the same time, pure static analysis approaches are ill-suited for JavaScript's dynamic typing, use of blackbox browser APIs,

[2] A headless browser performs all of the tasks that a normal browser performs during a page load except those that involve a GUI.

Goal	Techniques	Section
Ensure correctness	For each function, use concolic execution to identify union of the state it accesses across all control flows	§4.1.1
	Adapt offloading schedule during a page load to account for control flow in current load	§4.2.2
Account for API restrictions	Main browser thread centrally manages global page state, coordinates offloads, and performs unoffloadable (DOM) computations	§4.2.1, §4.2.3
	Intercept any Web Worker's DOM tree accesses and relay to the main thread	§4.2.3
	Use function signatures to determine what heap values to pass-by-value from main thread to workers and back	§4.2.3
Minimize overheads	Offline server-side generation of per-function signatures	§4.1.1
	Offload at the granularity of root functions	§4.1.2
	Dynamically determine offloading schedule based on function runtimes in current page load	§4.2.1

Table 3: **Overview of the main insights that Horcrux uses to address the challenges outlined in §3.**

and event-driven/asynchronous execution [75, 52]. For example, static analysis of variable name resolution is complicated by JavaScript statements that push objects to the front of the scope resolution chain (`with(obj){}`), or dynamically generate code (`eval()`). Similar issues arise from JavaScript's extensive use of variable aliasing for DOM objects, and the fact that property names are routinely accessed via dynamically-generated strings instead of static ones.

Thus, we turn to concolic execution [36, 80, 48], a variant of symbolic execution that executes programs concretely (rather than symbolically) while ensuring complete coverage of all control paths. A concolic execution engine loads a program with a concrete set of input values and observes its execution; §5 describes the inputs we consider, including browser state (e.g., cookies, screen size) and nondeterministic functions. Each input value and program-generated value is also given a symbolic expression constrained only by its type. For example, an input integer `a` may get a concrete value of 10 and an expression of $0 \leq \texttt{a} \leq 2^{32}-1$; symbolic expressions for a given variable are inherited by others via assignment statements. At each branch condition, the execution follows the appropriate path based on the current program state. In addition, the engine restricts the symbolic expressions for the values that influenced the chosen path according to the branching predicate. Once the program completes, the engine performs a backwards scan through the executed code, selects branching decisions to invert, and inverts the relevant symbolic expressions; an SMT solver [26] then generates concrete input values that satisfy the new constraints. This process repeats until all paths are explored.

Note that, for efficiency, many recent symbolic execution tools opt for a form of concolic execution, rather than a purely symoblic approach [20]. More specifically, concolic execution engines consult the expensive constraint solver only at the end of each path (rather than at intermediate branches), and eliminate the need to accurately model each input source to a program (and the ensuing traversal of paths that arise due to modeling errors).

To generate function signatures, in addition to the default output of a concolic execution engine – a list of potential control paths, with a concrete set of inputs to force each one – we must also log the state accessed by each path. To do this, prior to concolic execution, Horcrux instruments the JavaScript source code to log all accesses to state in both the JavaScript heap and DOM tree; our instrumentation matches recent dynamic analysis tools [63, 66, 64], but with the following differences based on our parallelism use case.

- First, we care not just about the state that remains at the end of the page load [64], but also any state accessible by multiple functions during a page load. Hence, in addition to global heap objects, Horcrux tracks all accesses to closure state: non-global state that is defined by a function *X* and is accessible by all nested functions that execute in *X*'s enclosed scope (anytime during the page load) [57].
- Since signatures will ultimately be used for pass-by-value offloading to Workers, only the finest granularity of accesses are logged. For instance, if object `a`'s "foo" property is read, Horcrux would log a read to `a.foo`, not `a`.
- For the DOM tree, Horcrux adopts a coarser approach than prior work. Instead of logging reads and writes to individual nodes in the DOM tree, Horcrux only logs whether a function accesses any live DOM nodes, either via DOM methods or references on the heap, and if so, whether they are reads or writes. Tracking at the coarse granularity of accesses to the entire DOM tree is conservative with respect to parallelism. However, finer-grained tracking is not beneficial because, as we explain later, our design has the browser's main thread serialize all DOM operations.

4.1.2 Signature granularity

Ideally, to limit client-side bookkeeping overheads, signatures should match the granularity at which computation is offloaded. However, determining the appropriate offloading granularity is challenging. On the one hand, fine-grained offloading reduces the chance that offloadable computations access shared state, thereby improving the potential parallelism and use of available Workers. On the other hand, finer granularities imply increased coordination overheads.

To address this tradeoff, Horcrux generates signatures (and offloads computation) at the granularity of *root function* invocations, i.e., invocations made directly from the global JavaScript scope. The signature for each root function invocation includes the state accessed not only by that function,

but also by any nested functions that are invoked in the call chain until the global scope is reached again.

Root function signatures are desirable for two reasons. First, they leverage our finding that functions already account for 94% of JavaScript code on the median page (§3) and thus provide a natural granularity for offloading; as in §3, Horcrux servers wrap all JavaScript code outside of any function into anonymous functions. Second, and more importantly, root functions impose far smaller offloading overheads compared to finer-grained function-level offloading, while enabling comparable parallelism benefits: the number of offloads drops by 4×, while the median potential benefits remain within 27% of those in Table 1. The reason is that there often exists significant state sharing within the invocations for a given root function (and its nested components), but less so across root functions, enabling parallelism.

4.2 Client-side Operation

Even with function signatures, a server cannot precompute a parallel execution schedule because the precise control flow, and hence, the set of functions executed and their runtimes, will vary across loads. Instead, Horcrux employs a client-side JavaScript computation scheduling library that unmodified browsers can run to dynamically make parallelism decisions based on signatures and the aforementioned runtime information. The key challenges are in efficiently ensuring correctness while offloading to multiple Workers, and handling the fact that signatures may be missing for certain functions. We next discuss how Horcrux addresses these challenges.

4.2.1 Dynamic scheduling

To load a page frame, any unmodified browser first downloads the top-level HTML, whose initial tag is an inline `<script>` housing Horcrux's scheduler library and all root function signatures. The scheduler runs on the browser's main thread and begins by asynchronously creating a pool of uninitialized Workers. This helps hide the primary overhead of spawning Workers amongst unavoidable delays for parsing initial HTML tags and fetching files they reference.

The scheduler then operates entirely in event-driven mode, whereby it waits for incoming postmessages specifying computations to perform or those that have completed, and makes subsequent offloading decisions. Importantly, to keep the main thread as idle as possible, the scheduler offloads *all* computations that Web Workers can support, and is primarily responsible for managing Workers and maintaining the page's global JavaScript heap and DOM state. This helps adapt the parallelization schedule within any page load to the runtimes of every root function in that load. The reason is that the main thread will be available to quickly assign a new function invocation (if one exists) to any worker that completes executing the function previously assigned to it.

Once the scheduler is defined, the browser operates normally, recursively fetching and evaluating referenced HTML, JavaScript, CSS, and image files. However, all JavaScript function invocations are modified to pass through the scheduler for offloading. More specifically, each root function is rewritten such that, upon invocation, the function sends a post message to the scheduler specifying its original source code and that of any nested functions. Special care is taken for asynchronous functions (e.g., timers) whose invocations are regulated by the browser's internal event queue which the scheduler does not have access to. To ensure visibility to such functions, the downloaded page includes shims around registration mechanisms for asynchronous functions, e.g., `setTimeout()`. Each shim modifies registered functions to send messages to the scheduler upon invocation.

Each time a root function is invoked, the scheduler uses its signature to determine whether or not it can be immediately offloaded. If not, the function is stored in an in-memory queue of *ordered*, to-be-invoked functions along with its signature. Functions are not offloadable if there are no available Workers, or if they might access state that is being modified by an already-offloaded or queued function. Note that functions that may access the DOM can be offloaded in parallel; we will discuss how to ensure safety in these cases shortly.

Regardless of the decision for a given invocation, the browser continues its execution. At first glance, it may appear that continuation after a queued invocation may generate errors since the queued function could alter the set of downstream invocations. However, recall that Horcrux offloads at the granularity of root functions—any nested invocations are already offloaded, and the ordering of root functions is mostly predefined by the page's source code. There are two exceptions. First, a function can alter downstream source code using `document.write()`; to handle this, the scheduler synchronously offloads such functions, thereby blocking downstream execution. Second, a root function can register an asynchronous function with a 0-ms timer—such functions are intended to run immediately after the current invocation. For this, the root signature includes state accesses for the 0-ms timeout functions they define. Once the root function is discovered, the scheduler adds a placeholder for the timeout function to its queue, thereby blocking downstream invocations that share state with the timeout function.

4.2.2 Handling Missing Signatures

We have assumed so far that the HTML file of every page frame includes accurate signatures for all JavaScript functions executed in that frame. This may not always hold.

- *Stale signatures.* A frame can include JavaScript content from multiple origins, and to preserve HTTPS content integrity and security [67], Horcrux has each origin serve its own files directly to clients. A third-party origin may update a script without explicitly informing the top-level origin to regenerate signatures. We expect this to be rare for two reasons. First, JavaScript files often have long cache lives (median of 1 day in our corpus), indicating infrequent changes. Second, scripts in a frame can share state [7].

Thus, even today, if a third-party origin significantly alters a script it serves, this should be communicated to other origins to avoid unexpected or broken behavior.

- *Dynamically-generated or personalized scripts.* JavaScript files may be created or personalized in response to user requests [54], e.g., based on Cookies. Unfortunately, generating signatures during client page loads would be far too slow. To handle this, for dynamically-generated or personalized first-party content, Horcrux could perform concolic execution on server-side content generation logic to determine the execution paths for all variants of a given response (§5). Third-party content of this type may result in missing signatures since the top-level origin does not have access to a user's third-party Cookies (or personalized content). However, many browsers preclude third-party Cookies in frames to prevent the tracking of users across sites [23].
- *Timeouts of concolic execution.* Given infinite time, concolic execution is guaranteed to explore all possible JavaScript execution paths in a page [36, 80]. However, the process may timeout, either during the execution of a given path (if the SMT solver cannot invert a branch condition), or less likely, due to a time bound placed on overall signature generation. Regardless of the reason, the effect is a potentially missing or incomplete signature. Such timeouts did not arise (i.e., concolic execution completed) for any pages in our experiments (§6.1). However, in the event that concolic execution does not complete, Horcrux could detect such timeouts *prior* to serving content to clients, and could thus address the corresponding missing signatures (described below) or revert to a normal page load.

Horcrux accounts for missing or inaccurate signatures in two ways. First, any underexplored function X is assigned a signature of $\star$, indicating that X may access all page state. This overconstrains the client's load, but ensures correctness: the client will execute X serially, and will also serialize downstream functions since X might alter the state they access. Second, signatures are keyed by a hash of the function's source code. Invocations without matching signatures are assigned signatures of $\star$ by Horcrux's client-side scheduler.

4.2.3 Function Offloading and Execution

Lastly, we discuss the mechanics of how every function invocation that is offloaded by the Horcrux scheduler is executed in a Web Worker. Figure 6 illustrates this process.

For each offloadable function, the scheduler uses its signature to generate a JSON package listing the information that the Worker will require for execution, i.e., the source code (including nested functions) and the current values for the function's read state. The source code is modified such that, upon completion, values in the write state are gathered and sent to the main thread (as execution results). In addition, closure values in the read state are embedded into the corresponding function's source code. Upon reception, the

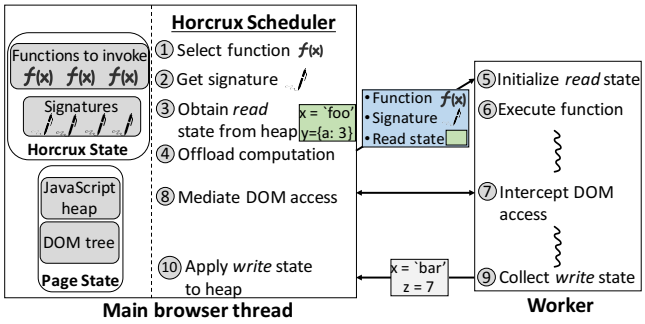


Figure 6: **A single function offload with Horcrux.**

post message handler inside the Worker sets up the read state in the Worker's global scope via assignment statements, and runs the code using the browser's `Function` constructor.

For the most part, functions execute normally, with JavaScript heap accesses hitting in the Worker's global scope, or for nested functions, in the scope of their parents–recall that Horcrux offloads entire root functions so nesting relationships are preserved. However, the key difference is with respect to DOM accesses: workers cannot call native DOM methods or operate on live DOM nodes referenced in the heap (§3.2). To handle this, all DOM computations are mediated by the scheduler and are serially applied to the live DOM tree. To intercept DOM computations, Horcrux includes shims around all DOM methods in each Worker environment; returned DOM nodes are replaced with proxy objects to interpose on direct accesses. Each intercepted DOM access blocks execution in the Worker, and is sent to the scheduler where it is queued; blocking is enforced using JavaScript generator functions [58].

The scheduler grants readers-writer locks to each Worker that may need to access the DOM tree (as per their signatures). Locks are granted in the order that the scheduler receives function invocations; note that this may not match the order in which functions are offloaded, but it preserves the relative ordering of DOM updates seen in a normal page load. As a concrete example, consider a function `a` that reads from the DOM, and a later function `b` that writes to the DOM. `b` may attempt to access the DOM first (e.g., if it is offloaded earlier or its DOM access occurs early in the function), but the scheduler will block it and wait to grant the lock to `a` first. Workers release DOM locks at the end of their execution. In essence, root functions that only read from the DOM tree can run in parallel, although their constituent DOM accesses are serialized on the browser's main thread. Root functions that write to the DOM are run serially with respect to other DOM-accessing root functions (to match the relative ordering of DOM updates in an unmodified load); for context, only 7.4% of the root functions on the median page in our corpus involve DOM writes, mitigating the effects of such serialization. Importantly, locking is done at the granularity of entire root invocations because the scheduler does not definitively know whether a given function will access

the DOM in the current load (and if so, how many times); signatures only list that a DOM access *may* occur.

Once a function completes execution, the Web Worker sends its computation results (i.e., its write state) to the scheduler. The Worker then clears any state in its scope in order to be ready for the next offload. Upon receiving computation results, the scheduler applies the writes to the global JavaScript heap; recall that DOM writes have already been made. One subtlety here is with respect to scope, and closure state in particular. The scheduler can access and apply computation results to the global scope's heap. However, root functions can also modify shared closure variables which are not accessible from the global scope (§4.1). For such writes, the scheduler maintains a global hash map listing the latest closure values. This map is updated as Workers complete computations, and is also queried to obtain read values when offloading; note that correctness is ensured because all offloads and computation results pass through the scheduler.

Finally, once the scheduler applies computation results to the JavaScript heap, it scans through its queue of ordered, to-be-executed functions and offloads the next one that can safely run. Given the serialization of DOM computations described above, if a function that writes to the DOM is queued, the scheduler prioritizes queued functions that do not access the DOM (and thus won't incur locking delays).

4.3 Discussion

Key to Horcrux's operation is the decision to maximally offload and parallelize JavaScript computations at the granularity of root functions. Recall that this decision was motivated by our analysis of the JavaScript computation on the pages in our experimental corpus (§3.1 and §4.2), which revealed that root function offloading favorably balances client-side overheads (e.g., pass-by-value I/O, main thread responsiveness) with the achievable speedups from parallelization.

However, these decisions may not deliver the largest speedups for certain pages. For example, root function-level offloading might be too restrictive and forego significant parallelism benefits, e.g., if a root function includes two nested functions that access entirely disjoint state but both involve significant runtime. Similarly, the root functions for certain pages could each embed only a single nested function, thereby inflating offloading costs relative to parallelization speedups, and potentially harming overall performance.

Although we did not observe these behaviors for any of the pages in our experiments (§6), we note that developers could perform analyses similar to the one presented in §3.1 to determine whether automatic parallelization of JavaScript code is desirable for (i.e., can speed up) their pages, and if so, what the best offloading granularity is. Importantly, these analyses do not require further instrumentation of web pages, and instead can directly leverage Horcrux's signatures, the per-function runtimes reported by in-built browser profilers, and the relatively stable offloading costs reported in Table 2.

5 Implementation

Horcrux instruments JavaScript source code to generate signatures and prepare frames using Beautiful Soup [78], Esprima [43], and Estraverse [84]. To employ concolic execution, we use a modified version of Oblique [48], which runs atop a headless version of Google Chrome (v85) and the ExpoSE JavaScript concolic execution engine [50]. Our Oblique implementation considers inputs specified by HTTP headers (e.g., Cookie, User-Agent, Origin, Host), the device (e.g., screen coordinates), and built-in browser APIs including nondeterministic functions [59] (e.g., `Math.Random`) and DOM methods. Input values suggested by the SMT solver are fed into the page load via either 1) rewritten HTTP headers, or 2) shims for browser APIs.

We grant Oblique a maximum of 10 mins to consider a given execution path, and 45 mins to explore all paths for a given page; we find that these time values are sufficient to ensure that concolic execution completes for all of the pages in our experimental corpora (§6.1). Signatures from each load are sent to a dedicated analysis server for aggregation. Since our current implementation operates directly on downloaded page source code and not live web backends (§6.1), Horcrux eschews Oblique's ability to perform concolic execution on server-side application logic. In total, Horcrux's implementation involves 5.6k LOC in addition to Oblique, including 4.5k for dynamic tracing (both static instrumentation and runtime tracking) and 1.1k for client-side scheduling.

Overheads. On the client-side, Horcrux inflates page sizes by 13 KB at the median (when using Brotli compression [41]). The scheduler accounts for 3 KB of that.

6 Evaluation

We empirically evaluate Horcrux across a wide range of real pages, live mobile networks, and phones from both developed and emerging markets. Our key findings are:

- Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index improvements of 18-29% and 24-37%. Improvements grow with warm browser caches (§6.2).
- Horcrux delivers larger speedups than prior web optimizations that 1) reduce required computations (by 1.7-2.1×), 2) speculatively parallelize computations (by 1.3-1.6×), and 3) mask network round trips (by 1.4-2.1×); Horcrux is complementary to network optimizations and running them together lowers load times by 31-45% (§6.3).
- Although the median page has 12 possible execution paths, Horcrux's reliance on conservative signatures (for correctness) only foregoes 7-10% of speedups compared to using signatures that target a specific load (§6.4).
- Horcrux is highly amenable to partial deployment: benefits are within 2% of total adoption when only a page's top-level origin runs Horcrux. Benefits persist for personalized pages and desktop settings (§6.5).

6.1 Methodology

We evaluated Horcrux in two different scenarios:

- **Developed regions.** We consider 700 US pages, randomly selected from and equally distributed amongst the following sources: popular landing pages from the Alexa [12] and Tranco [49] top 1000 lists, popular interior pages from the Hispar 100,000 list [16], and less popular pages (landing and interior) from the 0.5 million-site DMOZ directory [1]. Thus, our corpus involves diversity in terms of both page popularity and location within a website (i.e., landing vs. interior). From this set, we report results for the 582 pages that our current implementation could generate accurate signatures for. More specifically, we removed pages for the following reasons: 1) inaccurate signatures due to unsupported language features, which led to premature JavaScript termination (92) or rendering defects (22), and 2) unsupported features with Oblique (4). For all of the remaining pages, Horcrux's concolic execution and overall signature generation completes, the total JavaScript runtime with Horcrux falls within a standard deviation of that in default loads, and the final rendered page is unchanged. Our experiments consider two powerful phones, a Pixel 3 (Android Pie; 2.0 GHz octa-core processor; 4 GB RAM) and a Galaxy Note 8 (Android Oreo; 2.4 GHz octa-core; 6 GB RAM). For space, we only present results for the Pixel 3, but note comparable results with the Galaxy Note 8.
- **Emerging regions.** Web experiences in emerging regions often comprise different page compositions and devices than those considered above [24, 10, 11, 90]. To mimic such scenarios, we focus on a single emerging region: Pakistan. We consider a corpus of 100 landing and interior pages (50 each) selected from the Alexa Top 500 sites in Pakistan. Our evaluation uses the Redmi 6A phone (Android Oreo; 2.0 GHz quad-core processor; 2 GB RAM) that is popular in the region [4]. As per the same correctness checks as above, we report numbers on 91 pages.

Unless otherwise noted, page loads were run with Google Chrome for Android (v85). Mobile-optimized (including AMP [37]) pages were always used when available.

To create a reproducible test environment, and because Horcrux involves page rewriting, we use the Mahimahi web record-and-replay tool [68]. Emerging regions pages were recorded using a VPN to mimic a client in Pakistan. As described in §4, Horcrux's signature generation and page rewriting were performed offline. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Our phones were connected to the desktop via USB tethering and live Verizon LTE and WiFi networks with strong signal strength; LTE speeds for emerging regions experiments were throttled to Pakistan's 7 Mbps average [70]. We used Lighthouse [42] to initiate page loads via the USB connection, and all page load traffic traversed the wireless networks.

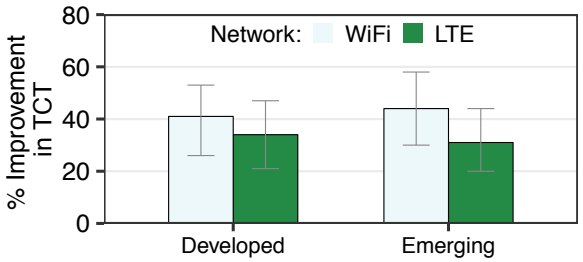
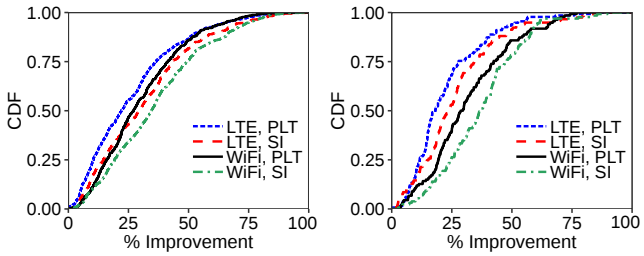


Figure 7: **Cold cache TCT improvements over default page loads. Bars list medians, with error bars for 25-75th percentiles.**



(a) **Developed region.** (b) **Emerging region.**

Figure 8: **Distributions of cold cache per-page improvements with Horcrux vs. default page loads.**

We evaluated Horcrux on multiple web performance metrics: 1) *Total Computation Time* (TCT), or the critical path of time spent parsing/executing source files and rendering the page, 2) *Page Load Time* (PLT) measured as the time between the `navigationStart` and `onload` JavaScript events, and 3) *Speed Index* (SI) [40] which captures the time required to progressively render the pixels in the initial viewport to their final form. TCT and PLT are measured using the browser profiler, while SI was reported by Lighthouse. In all experiments, we load each page three times with each system under test, rotating amongst them in a round robin; we report numbers per system from the load with the median TCT.

6.2 Page Load Speedups

Cold cache. Figure 7 illustrates Horcrux's ability to reduce browser computation delays compared to default page loads. TCT reductions were 41% (1.0 sec) and 44% (1.5 sec) for the median page in the developed and emerging region's WiFi settings, respectively. Improvements were 34% and 31% with LTE. Figure 8 shows how these computation speedups translate into faster end-to-end (i.e., including network delays) loads. For example, on WiFi, median improvements in the developed region setting were 27% for PLT and 35% for SI. Despite the lower CPU clock speeds, these numbers only marginally increase to 29% and 37% in the emerging region. Further improvements were hindered primarily by the lower number of available cores (and thus ability to parallelize). Benefits with Horcrux on LTE were comparable, but consistently lower than with WiFi. For example, in the developed region, PLT and SI speedups were 22% and 29%. The reason is that network delays (which Horcrux does not improve) account for larger fractions of end-to-end load times on LTE.

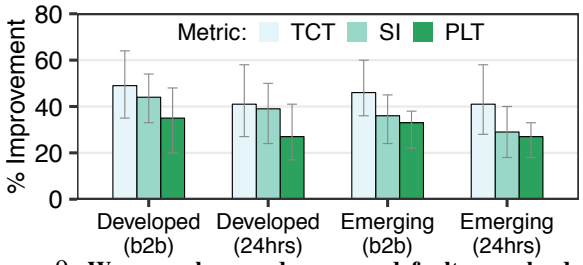


Figure 9: **Warm cache speedups over default page loads on LTE. Bars list medians, with error bars for 25-75th percentiles.**

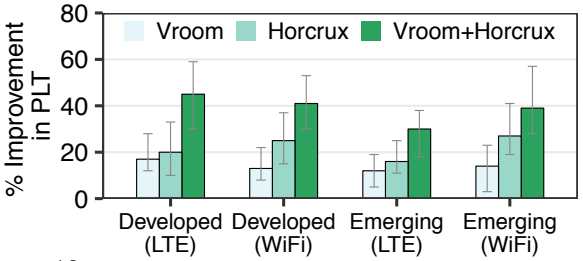


Figure 10: **Horcrux vs. Vroom [79] over LTE networks. Bars list medians, with error bars for 25-75th percentiles.**

Warm cache. Figure 9 shows Horcrux's speedups in different browser caching scenarios. We consider back-to-back page loads, as well as those separated by 12 and 24 hours. As shown, Horcrux's improvements grow as browsers house more objects in their caches. For example, in the back-to-back scenario, PLT and SI improvements in the developed region's LTE setting were 35% and 44%; for context, these improvements were 22% and 29% with cold caches. Improvements drop to 27% and 39% in the 24-hour warm cache scenario. The reason is that more cache hits lead to lower network delays and computation dominating end-to-end performance. In addition, cache hits enable browsers to begin processing JavaScript files earlier. This, in turn, provides Horcrux's scheduler with more invocation options at any time, thereby increasing the amount of potential parallelism.

6.3 Comparison to State-of-the-Art

Network optimizations. We first considered Vroom [79], a system in which web servers intelligently use HTTP/2's server push and preload features to aid clients in discovering (and downloading) required files ahead of time. Thus, Vroom is primarily a network-focused optimization. However, key to Vroom's benefits is the improved CPU utilization that results from eliminating blocking network fetches.

As shown in Figure 10, Horcrux delivers larger speedups than Vroom. For example, in the developed region, median PLT speedups with Horcrux are 2.1× and 1.3× higher than Vroom's on WiFi and LTE, respectively. In the LTE setting, Vroom delivers larger PLT speedups for 9% of pages. The reason is that network delays play a larger role in end-to-end load times for these pages, either due to less computation or more required network fetches. This drops to 1% and 3% when we move to the developed region's WiFi network or the emerging market's LTE network; in both cases, compute becomes more of a determinant of overall delays. Importantly, Figure 10 also confirms that Horcrux and Vroom are largely complementary to one another, with the combined systems outperforming each in isolation.

System	**Developed**	**Emerging**
Horcrux	1.63 (1.98)	2.15 (2.37)
Prepack [32]	2.19 (2.47)	2.82 (3.36)
Speculative parallelization	2.01 (2.28)	2.50 (3.07)

Table 4: **Comparing Horcrux with prior compute optimizations. Results are for WiFi networks and list median (75th) percentile TCTs in seconds.**

Reducing required computations. Prepack [32] is a server-side system that reduces the amount of JavaScript computation that clients must perform to load pages. To do this, Prepack performs static analysis on a page's JavaScript code, identifies expressions whose results are statically computable, and replaces those expressions with equivalent but simpler versions that remove intermediate computations. Importantly, computations involving client-side or nondeterministic state are unmodified; this helps Prepack preserve page behavior and correctness, unlike other computation reduction systems (§7). As shown in Table 4, Horcrux is more effective at reducing computation delays than Prepack: median TCTs are 26% and 24% lower with Horcrux in the developed and emerging regions, respectively.

Speculative parallelism. Prior efforts to increase parallelism in page loads (§7) primarily rely on speculative decisions about what can run in parallel, and runtime checks to detect (and revert from) dependency violations. Although these systems do not target all JavaScript execution, we considered a baseline that employs a similar parallelism strategy for JavaScript computation. Our baseline opportunistically parallelizes all root function invocations, and uses JavaScript proxy objects to track state accesses in each Worker. Any parallelized computations that share state are discarded, and the corresponding functions are rerun serially on the main browser thread. As shown in Table 4, Horcrux delivers superior median TCT values that are 14-19% lower across the two regions. The reason is twofold. First, proxy-based tracking to ensure correctness adds 10% overhead to JavaScript runtimes. Second, any speculation errors result in serial execution on the main thread and wasted computation (and thus, more overall computation). Using the setup in §6.5, we observe that this wasted computation inflates mobile device energy consumption by 9% for the median page on WiFi.

6.4 Understanding Horcrux's Benefits

Dissecting Horcrux's speedups. We analyzed Horcrux's behavior (and improvements) along three different axes. We focus on the developed region, but note that the trends hold for the emerging region setting. First, as expected, Horcrux's improvements are larger for pages that require more computation to load. For instance, with WiFi, median PLT improve-

ments with Horcrux were 35% for pages with more than 3 seconds of computation time, as compared to 23% for pages that did not meet that criteria. This divide carries over to different page types as well: improvements were 15% higher for interior pages than landing pages. The reason is that interior pages often involve more computation [16, 64].

Second, within each load, we investigated the degree of parallelism that Horcrux achieves for JavaScript computation. For the median page, when loaded over WiFi, Horcrux reaches a maximum of 6 concurrent Web Workers; this drops to 4 on LTE due to the aforementioned network delays limiting the scheduler's parallelism options.

Third, in addition to JavaScript parallelization, Horcrux reduces TCT by freeing the browser's main thread for rendering tasks. To understand the contribution of each source to Horcrux's speedups, we analyzed the browser's computation profiler. Overall, we find that both sources provide substantial benefits. For instance, on WiFi, Horcrux shrinks effective JavaScript computation times by a median of 42% (557 ms), and decreases end-to-end rendering delays by 36% (465 ms).

Server-side overheads. Signature generation took 33 minutes for the median page, and involved two primary overheads: the median page involved exploring 12 different execution paths via concolic execution, and our dynamic instrumentation (incurred in each load) inflated load times by 44%. These non-negligible delays are why Horcrux performs comprehensive signature generation offline, on servers. To understand how often servers have to incur these overheads, we recorded a random set of 50 pages from our emerging region's corpus every 12 hours for 1 week. The median page's signatures remained unchanged for the entire duration, in part due to Horcrux's coarse-grained DOM tracking which is unaffected by changes to HTML state (e.g., headlines).

Cost of conservative signatures. Horcrux relies on conservative signatures that list the state accesses across all possible control flows. While this ensures correctness, it may overconstrain a client load that traverses only a subset of those control flows. To understand the impact of this conservative strategy, we compared Horcrux with a variant that generates signatures for the precise control flows traversed in the target client load. Surprisingly, we observe that Horcrux's conservative behavior results in only mild performance degradations: improvements drop by 10% and 7% for PLT and SI, respectively, for the developed region WiFi setting. The reason is that conservative signatures typically either add only a few extra state accesses to a given root function, or many that are only accessed for short durations (i.e., within a root function)—neither significantly restricts parallelism.

6.5 Additional Results

Partial deployment. Our results thus far assumed that each frame in a page adopts Horcrux, i.e., embeds Horcrux's scheduler and signatures in the HTML. Figure 11 shows Horcrux's benefits when only the top-level origin for the page participates—this represents the simplest deployment scenario as the top-level origin is directly incentivized to accelerate loads of its pages. In this scenario, all JavaScript code in third-party-owned frames runs serially; JavaScript in the main frame can still be safely parallelized as browsers prevent cross-frame state sharing [7]. As shown, most of Horcrux's benefits persist, despite the lack of adoption by third-party frames. For example, in the developed region's WiFi setting, median SI benefits are within 2% of those with total adoption. The reason is that most JavaScript runtime (100% on the median page) resides in the page's main frame.

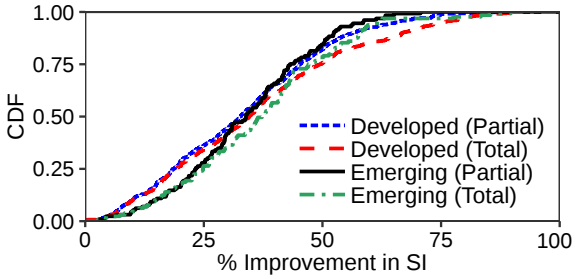


Figure 11: **Evaluating Horcrux when only a page's top-level origin participates. Results are for WiFi networks.**

Personalized pages. To evaluate Horcrux in settings where pages dynamically generate or personalize their content, we selected 20 pages from our developed region's corpus that supported user accounts. For each page, we created two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. For every file that does not appear in both loads, or whose content is different across the page versions, we assign its constituent functions signatures of $\star$ (§4.2.2). Overall, we observe that such personalization has minimal impact on Horcrux's speedups: in the WiFi setting, Horcrux's median load time benefits drop by only 4%. The reason is that only 6% of computation delays are accounted for by personalized scripts.

Energy savings. We connected our Pixel 3 phone to a Monsoon power monitor [60] and loaded the pages in our developed region corpus. With cold caches, Horcrux's speedups drop median per-page energy usage by 12% and 15% on WiFi and LTE. Savings are primarily from accelerating end-to-end computation (and load times), which results in lower active durations for WiFi or LTE radios.

Desktop page loads. Horcrux's acceleration techniques can also speed up desktop page loads. To evaluate this, we recorded desktop versions for the pages in our developed region corpus, and loaded them using a Dell G5 desktop and a wired network connection. We find that Horcrux reduces median TCT by 39% (0.52 secs). These speedups translate to PLT and SI improvements of 25% and 31%, respectively. At first glance, these improvements may appear surprising given the faster CPU clock speeds that desktops possess. However, desktops also possess more cores and load pages with more JavaScript computation [44], enabling more parallelism.

7 Related Work

Parallelization efforts. ParaScript [56] and others [61] leverage new runtimes and compiler information to speculatively parallelize iterations for hot loops in long-running JavaScript code (not page loads, where compilation overheads are too costly). In contrast, Horcrux operates with unmodified browsers, targets parallelism for general JavaScript code beyond loop iterations, and sidesteps the significant overheads of speculation errors and runtime checks (§6.3) by using conservative signatures. Zoomm [21] and Adrenaline [53] leave JavaScript execution unchanged, and instead parallelize tasks such as CSS rule parsing. These systems are orthogonal to Horcrux, which focuses entirely on JavaScript parallelization. Lastly, several libraries [6, 9] aid developers in writing parallel JavaScript code by abstracting inter-worker messaging. However, developers are responsible for identifying and enforcing (safe) parallelism decisions—Horcrux automates these tasks for legacy pages.

Reducing web computation overheads. Prior measurement studies have analyzed the performance of mobile web browsers [85, 62, 24, 73]. Like us (§2), they find that browser computations are a primary contributor to high page load times. In response to these studies, three separate lines of work have aimed to alleviate browser computation delays. First, certain sites have manually developed mobile-optimized versions of their pages using restricted forms of HTML, JavaScript, and CSS, e.g., according to the Google AMP standard [37, 46]. In contrast, Horcrux accelerates legacy pages without developer effort. Further, we find that Horcrux is able to accelerate the loading of AMP pages, which constitute 27% of our corpora.

Second, some systems [13, 87, 71, 22] offload computation tasks to well-provisioned proxy servers, which return computation *results* that are fast to apply. Though effective, such systems pose significant scalability challenges to support large numbers of mobile clients [82]. Worse, by relying on (often third-party) proxy servers, these systems violate HTTPS' end-to-end security guarantees [67]; clients must trust proxies to preserve the integrity of their HTTPS objects, and also must share private Cookies to accelerate personalized page content. In contrast, Horcrux is HTTPS-compliant.

Third, systems like Prophecy [64] enable servers to return post-processed page files that elide intermediate computations. However, content alterations with these systems may break page functionality [10], particularly for pages that adapt execution based on client-side state that servers are unaware of, e.g., localStorage. In contrast, Horcrux does not alter the set of computations required to load a page, and instead aims to execute those computations more efficiently.

Network optimizations for the web. Systems such as Alohamora [47], Vroom [79], and others [29, 86] leverage HTTP/2's server push and preload features to proactively serve files to clients in anticipation of future requests (thereby hiding download delays). Fawkes [54] develops static HTML templates that can be rendered while dynamic data is fetched. Polaris [63] and Klotski [19] reorder network requests to minimize the number of effective round trips while respecting inter-object dependencies. Cloud browsers [83, 67, 68] shift network round trips to wired proxy server links. Content delivery networks [69, 33] serve popular objects from proxy servers that are geographically close to clients, while compression proxies [10, 81, 72] selectively compress objects in-flight between servers and clients. Lastly, a handful of systems prefetch content according to predicted user browsing behavior [74, 51, 88]. As shown in §6.3, these efforts are complementary to Horcrux, which reduces browser computation delays by parallelizing JavaScript execution. Further, recall that computation delays often exceed user tolerance levels on their own (§2).

Concolic execution for web optimization. Like Horcrux, Oblique [48] uses concolic execution to accelerate web page loads. Indeed, Horcrux's server-side component builds atop Oblique's JavaScript concolic execution engine by adding dynamic instrumentation to capture per-function signatures (§5). However, despite this similarity, Oblique and Horcrux target different delays in the page load process: Oblique enables third-party servers to securely prefetch URLs that a client will need during a page load (hiding the associated network fetch delays), while Horcrux parallelizes the JavaScript execution required to load a page (reducing the associated computation delays). Consequently, as with other network-focused optimizations (§6.3), Oblique can run alongside Horcrux to provide complementary benefits.

8 Conclusion

Horcrux automatically parallelizes JavaScript computations in legacy pages to enable unmodified browsers to leverage the multiple CPU cores available on commodity phones. To account for the non-determinism in page loads and the constraints of the browser's API for parallelism, Horcrux employs a judicious split between clients and servers. Servers perform concolic execution of JavaScript code to conservatively identify parallelism opportunities based on potential state accesses, while clients use those insights along with runtime information to efficiently manage parallelism. Across browsing scenarios in developed and emerging regions, Horcrux reduced median browser computation delays and load times by 31-44% and 18-37%.

Acknowledgements: We thank James Mickens, Amit Levy, Anirudh Sivaraman, and Harry Xu for their valuable feedback on earlier drafts of the paper, as well as Blake Loring for useful discussions on the ExpoSE JavaScript concolic execution engine. We also thank our shepherd, Ryan Huang, and the anonymous OSDI reviewers for their constructive comments. This work was supported in part by NSF grants CNS-1943621 and CNS-2006437.

REFERENCES

[1] Directory of the web. https://dmoz-odp.org/.

[2] Web Workers. https://w3c.github.io/workers/, 2017.

[3] Find out how you stack up to new industry benchmarks for mobile page speed). https://www.thinkwithgoogle.com / marketing-strategies / app-and-mobile / mobile-page-speed-new-industry-benchmarks/, 2018.

[4] 10 Most Popular Phones in India in 2020 – Xiaomi and Samsung Rules. https://candytech.in/most-popular-phones-in-india/, 2020.

[5] Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share, 2020.

[6] Parallel.js. https://github.com/parallel-js/parallel.js, 2020.

[7] Same-origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2020.

[8] The 10 Best Chromium Browser Alternatives Better Than Chrome. https://www.makeuseof.com/tag/alternative-chromium-browsers/, 2020.

[9] Threads.js. https://github.com/andywer/threads.js, 2020.

[10] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

[11] S. Ahmad, A. L. Haamid, Z. A. Qazi, Z. Zhou, T. Benson, and I. A. Qazi. A View from the Other Side: Understanding Mobile Phone Characteristics in the Developing World. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 319–325. Association for Computing Machinery, 2016.

[12] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[13] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[14] D. An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com / marketing-resources / data-measurement / mobile-page-speed-new-industry-benchmarks/, 2018.

[15] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August. Perspective: A Sensible Approach to Speculative Automatic Parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 351–367. Association for Computing Machinery, 2020.

[16] W. Aqeel, B. Chandrasekaran, A. Feldmann, and B. Maggs. On Landing and Internal Web Pages. In *Proceedings of the 2020 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC. ACM, 2020.

[17] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[18] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[19] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[20] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[21] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A parallel web browser engine for multicore mobile devices. In *PPoPP*, 2013.

[22] M. Chaqfeh, Y. Zaki, J. Hu, and L. Subramanian. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of The Web Conference 2020*, WWW '20. ACM, 2020.

[23] Cookiebot. Google ending third-party cookies in Chrome. https://www.cookiebot.com/en/google-third-party-cookies/, 2020.

[24] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of Device Performance on Mobile Internet QoE. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, New York, NY, USA, 2018. ACM.

[25] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet QoE. In *IMC*, 2018.

[26] L. De Moura and N. Bjørner. 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, page 337–340. Springer-Verlag, 2008.

[27] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[28] E. Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[29] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[30] D. Etherington. Mobile internet use passes desktop for the first time, study

finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[31] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

[32] Facebook. Prepack. https://github.com/facebook/prepack, 2019.

[33] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. NSDI. USENIX, 2004.

[34] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[35] Y. Geng, Y. Yang, and G. Cao. Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices. In *IEEE Conference on Computer Communications*, INFOCOM, pages 46–54, 2018.

[36] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223. ACM, 2005.

[37] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[38] Google. Chromium. https://www.chromium.org/Home.

[39] Google. Why performance matters? https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[40] Google. Speed Index - WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2012.

[41] Google. Brotli compression format. https://github.com/google/brotli, 2019.

[42] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse/, 2019.

[43] A. Hidayat. Esprima. http://esprima.org.

[44] HTTP Archive. State of Javascript. https://httparchive.org/reports/state-of-javascript, 2020.

[45] J. Huang, P. Prabhu, T. B. Jablin, S. Ghosh, S. Apostolakis, J. W. Lee, and D. I. August. Speculatively Exploiting Cross-Invocation Parallelism. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 207–221, New York, NY, USA, 2016. Association for Computing Machinery.

[46] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[47] N. Kansal, M. Ramanujam, and R. Netravali. Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[48] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating Page Loads Using Symbolic Execution. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[49] V. Le Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczynski, and t. . T. s. . N. y. . . Joosen, Wouter.

[50] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017. ACM, 2017.

[51] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[52] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 499–509. Association for Computing Machinery, 2013.

[53] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar. USENIX Association, 2012.

[54] S. Mardani, M. Singh, and R. Netravali. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2020. USENIX Association.

[55] MDN. Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Worker, 2020.

[56] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA, 2011.

[57] J. Mickens. Rivet: Browser-Agnostic Remote Debugging for Web Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, USA, 2012. USENIX Association.

[58] J. Mickens. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. SP '14, page 261–275. IEEE Computer Society, 2014.

[59] J. Mickens, J. Elson, and J. Howell. Mugshot: De-

terministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10. USENIX Association, 2010.

[60] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[61] Y. Na, S. W. Kim, and Y. Han. JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications. *ACM Trans. Archit. Code Optim.*, 12(4):64:1–64:25, Jan. 2016.

[62] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[63] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[64] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[65] R. Netravali and J. Mickens. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[66] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[67] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

[68] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[69] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010.

[70] OpenSignal. Pakistan: Mobile Network Experience Report, February 2020. https://www.opensignal.com/reports/2020/02/pakistan/mobile-network-experience, 2020.

[71] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[72] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[73] A. Osmani. The Cost of JavaScript. https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4, 2018.

[74] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[75] J. Park, I. Lim, and S. Ryu. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 61–70. ACM, 2016.

[76] C. Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[77] G. Phillips. Smartphones vs. desktops: Why is my phone slower than my pc? https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[78] L. Richardson. Beautiful Soup Documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019.

[79] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM, 2017.

[80] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272. ACM, 2005.

[81] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[82] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[83] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM Inter-*

national on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[84] Y. Suzuki. Estraverse. https://github.com/estools/estraverse.

[85] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[86] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[87] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[88] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[89] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[90] Y. Zaki, J. Chen, T. Pötsch, T. Ahmad, and L. Subramanian. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 241–248. Association for Computing Machinery, 2014.

SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs

Jiang Zhang[1] Shuai Wang[2*] Manuel Rigger[3] Pingjia He[3] Zhendong Su[3]

University of Southern California[1] *HKUST*[2] *ETH Zurich*[3]

Abstract

Sanitizers detect unsafe actions such as invalid memory accesses by inserting checks that are validated during a program's execution. Despite their extensive use for debugging and vulnerability discovery, sanitizer checks often induce a high runtime cost. One important reason for the high cost is, as we observe in this paper, that many sanitizer checks are *redundant* — the same safety property is repeatedly checked — leading to unnecessarily wasted computing resources.

To help more profitably utilize sanitizers, we introduce SANRAZOR, a practical tool aiming to effectively detect and remove redundant sanitizer checks. SANRAZOR adopts a novel hybrid approach — it captures both dynamic code coverage and static data dependencies of checks, and uses the extracted information to perform a redundant check analysis. Our evaluation on the SPEC benchmarks shows that SANRAZOR can reduce the overhead of sanitizers significantly, from 73.8% to 28.0–62.0% for AddressSanitizer, and from 160.1% to 36.6–124.4% for UndefinedBehaviorSanitizer (depending on the applied reduction scheme). Our further evaluation on 38 CVEs from 10 commonly-used programs shows that SANRAZOR-reduced checks suffice to detect at least 33 out of the 38 CVEs. Furthermore, by combining SANRAZOR with an existing sanitizer reduction tool ASAP, we show synergistic effect by reducing the runtime cost to only 7.0% with a reasonable tradeoff of security.

1 Introduction

Software *sanitizers* are designed to detect software bugs and vulnerabilities in code written in unsafe languages like C/C++ [33]. A sanitizer typically inserts additional checks into the program during compilation; at run time, the sanitizer check terminates the program if it detects unsafe actions or states (e.g., a buffer overflow). To date, various sanitizers have been designed to help detect vulnerabilities in C/C++ programs [4, 8, 24, 32, 35].

*Corresponding author.

Sanitizers are commonly used by developers to find bugs before software deployment. In principle, they could also be used in deployed software, where they terminate program executions to prevent vulnerabilities from being exploited. In practice, however, the high runtime overhead of sanitizers inhibits their adoption in this application scenario [33, 40]. For example, our study on SPEC CPU2006 benchmarks [34] shows that the geometric mean overheads induced by AddressSanitizer (ASan) [32] and UndefinedBehaviorSanitizer (UBSan) [8] are, respectively, 73.8% and 160.1% (cf. Sec. 6).

It is difficult to reduce the overhead of sanitizer checks and therefore accelerate the execution of sanitization-enabled programs. To date, a number of approaches have been proposed aiming at finding unnecessary sanitizer checks with static analysis [6, 9, 11, 12, 15, 25, 37, 39, 44, 45]. For example, some approaches remove array bound checks by checking whether the value range of an index falls within the array size. They usually perform heavyweight, specialized program analyses to reduce specific sanitizer checks. In contrast, ASAP [40], the most closely related work to ours, elides sanitizer checks deemed the most costly based on a user-provided overhead budget. Despite being general and supporting sanitizers of different implementations, ASAP removes checks irrespective of their importance and may overoptimisitcally remove them, resulting in missed vulnerabilities. Thus, prior approaches for reducing sanitizer checks use either sanitizer-specific static analyses (e.g., [9, 37]) to remove only semantically redundant checks, or general heuristics [40] to remove costly sanitizer checks irrespective of their semantics.

This work explores a new, novel design point — it introduces a general framework, SANRAZOR, for effectively removing likely redundant checks. SANRAZOR is designed as a hybrid approach. First, it gathers coverage statistics during a profiling phase (e.g., based on a program's test suite). It then performs a correlation analysis, employing both the profiled coverage patterns as well as static data dependencies, to pinpoint and remove checks identified as likely redundant. Like ASAP [40], SANRAZOR is general and orthogonal to existing sanitizer reduction approaches that focus on specific

sanitizer check implementations [9, 37, 44]. Distinct from ASAP, SANRAZOR identifies and removes sanitizer checks that repeatedly check the same program property, while ASAP removes sanitizer checks of high cost and may miss vulnerabilities. Although, like ASAP, SANRAZOR is *unsound*, i.e., it may remove checks even when they are unique, in practice, our evaluation results show that it accurately maintains the sanitizer's effectiveness in discovering defects and provides significantly reduced runtime overhead.

We evaluate the performance gain of SANRAZOR on the SPEC CPU2006 benchmark. The results show that SANRAZOR reduces geometric mean runtime overhead caused by ASan from 73.8% to 28.0–62.0% (depending on the different reduction schemes in SANRAZOR). Similarly, geometric mean overhead incurred by UBSan on SPEC programs is reduced from 160.1% to 36.6–124.4%. To measure the accuracy of SANRAZOR, we evaluate 10 popular programs with a total of 38 known CVEs. Results show that after removing redundant sanitizer checks, at least 33 CVEs can still be discovered. Compared with ASAP, SANRAZOR significantly outperforms ASAP by discovering more CVEs when achieving the same amount of cost reduction. We also explored practical methods to combine SANRAZOR and ASAP and reduce runtime cost to only 7.0% with a reasonable tradeoff of security. These promising results suggest that SANRAZOR could help promote the adoption of sanitizers in production usage. In sum, we make the following main contributions:

- At the conceptual level, we introduce the novel approach to reducing performance overhead incurred by sanitizers by identifying and removing likely redundant checks. By reducing sanitizer cost, sanitization-enabled programs can be executed faster, making sanitizer adoption in production use more practical.
- At the technical level, we design and implement a practical tool, SANRAZOR, to reduce sanitizer checks. SANRAZOR performs a hybrid analysis by leveraging both coverage patterns and static data dependency features to identify sanitizer checks as likely redundant.
- At the empirical level, our evaluation on the SPEC benchmarks shows that SANRAZOR can significantly reduce runtime overhead caused by ASan and UBSan. Moreover, our evaluation on real-world software with known CVEs shows that after applying SANRAZOR to reduce sanitizer checks, almost all CVEs can still be discovered.

We have publicly released SANRAZOR on GitHub at https://github.com/SanRazor-repo/SanRazor.

2 Preliminaries

Sanitizers are dynamic tools for finding software defects [33]. Sanitizers insert *sanitizer checks*, which are statements for monitoring program behaviors and validating whether they violate certain properties. We now introduce two sanitizers provided by the LLVM framework, ASan and UBSan, which have helped to detect many vulnerabilities [33].

ASan. Memory access errors like buffer overflow and use-after-free are severe vulnerabilities in C/C++ programs. ASan is designed to detect memory errors [32], and consists of an instrumentation module and a runtime library. The instrumentation module allocates shadow memory regions for each memory address used by the program. It also instruments each memory load and store operation such that before a memory address a is used to access memory, a will be mapped to its corresponding shadow memory address sa; the value stored in sa is then loaded and checked to decide whether the access via a is safe. The instrumentation module also allocates a "bad" region for each shadow memory region; directly using a shadow memory address sa in the application code will be redirected to the "bad" region, which is inaccessible via page protection. The runtime library hooks the `malloc` function to create poisoned "redzones" next to allocated memories to detect memory access errors. Similarly, the `free` function is instrumented to put the entire deallocated memory region into "redzones." This ensures that the recently-freed region will not be used by `malloc` for reallocation.

UBSan. Undefined behaviors can incur severe software vulnerabilities [41]. UBSan [8] detects a large set of common undefined behaviors in C/C++ code, such as out-of-bounds access, divided by zero, and invalid shift. We briefly introduce one undefined behavior that UBSan can detect:

Out-of-bounds Array Access Unlike ASan, which relies on shadow memory, UBSan detects out-of-bounds array accesses by comparing each array index with the array size. Consider the sample code below:

```
1 UChar buf[32];  // buf size is 32
2 for(i = 0; i < nBuf; i++)
3   out[i] = buf[nBuf-i-1];
```

where `buf` has 32 elements. When `nBuf` is greater than 32, executing `buf[nBuf-i-1]` may trigger an out-of-bounds access (e.g. when `i` is 0). UBSan identifies this by placing an extra `if` condition to compare the array index `nBuf-i-1` with the array size 32 before executing the loop body.

3 Problem Formulation

Conceptually, a sanitizer check $c(v)$ (v is the input parameter) can be defined as follows:

```
if(P(v) does not hold) abort_or_alert();
```

where c checks whether a property P holds w.r.t. parameter v. Usually, v denotes critical program information (e.g., code pointers), and by violating property P, e.g., a null pointer dereference, c either aborts program execution or alerts the user. Considering a program p with N sanitizer checks inserted, we use $c_i.v$ and $c_i.P$ to denote the parameter of the ith check c_i and its checked property throughout this section.

As introduced in Sec. 2, computation overhead can be introduced by each c_i, since c_i performs complex safety property checking, and may require extra memory to store metadata.

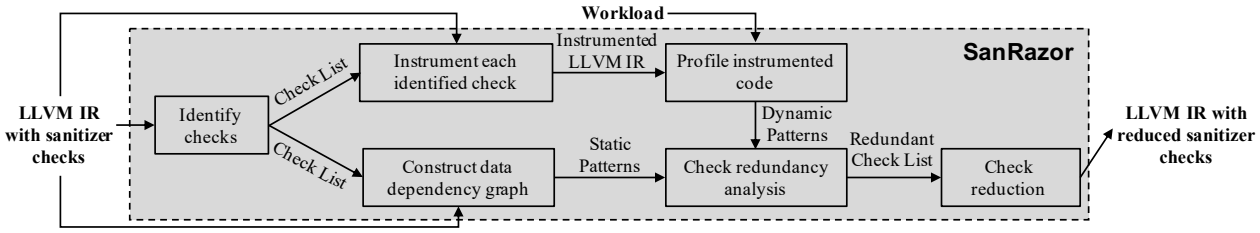


Figure 1: Workflow of SANRAZOR.

Nevertheless, a large portion of sanitizer checks repeatedly assert *identical* properties, thus wasting computing resources on properties deemed safe. We aim to remove redundant checks to reduce cost, thus making the production adoption of sanitizer checks more practically feasible. Next, we present a motivating example, and then formulate the notion of "redundant sanitizer checks" from a functionality perspective.

3.1 Sample Redundant Checks in `bzip2`

We show an example of checks in `bzip2` that repeatedly validate the same array index as follows:

```
for(i=0; i < nblock; i++) {
 j = eclass8[i]; //ASan1
 k = ftab[j] - 1; //ASan2;
 ftab[j] = k; //ASan3; ASan2 and ASan3 identical?
 fmap[k] = i; //ASan4; k < fmap_size always hold?
```

When ASan is enabled, four sanitizer checks are inserted to detect out-of-bound array access. Existing research could remove `ASan4`, by asserting `k` always falls within the size of `fmap`. In contrast, SANRAZOR advocates a new and orthogonal focus by deciding that `ASan2` and `ASan3` validate the same index, and therefore, `ASan3` can be removed without missing potential defects.[1] Indeed, our study shows that check redundancy is a general concern in real-world software (see Sec. 6), motivating a strong need for optimization. Also, to the best of our knowledge, standard compiler optimizations and previous research in this field (e.g., [6,9,11,12,15,28]) do not strive to use "similarity analysis" to reveal the equalivance of `ASan2` and `ASan3` and shave `ASan3` accordingly. In fact, our study shows that, when full optimizations (`-O3`) of `clang` are enabled, no sanitizers can be shaved for this case. This observation underlies the key novelty of SANRAZOR, whose design will be introduced in Sec. 4.

3.2 Redundant Sanitizer Checks

We start by giving a general definition of what a redundant check is, before refining the notion in an operational way:

Definition 1. *Assume that a sanitizer check c_i that could detect a hypothetical bug B in program p is removed. If B can still be detected, either by another sanitizer check c_j or by a user-defined check, then c_i is a redundant sanitizer check.*

More formally, given a nontrivial, single-threaded program p with a set of checks $c \in \mathbb{C}$, two checks c_i and c_j are deemed identical, when the following condition holds:

$$(c_i \in \text{dom}(c_j) \vee c_j \in \text{dom}(c_i)) \wedge [\![c_i.v]\!] = [\![c_j.v]\!] \wedge c_i.P = c_j.P$$

where $c_i \in \text{dom}(c_j)$ and $c_j \in \text{dom}(c_i)$ denote that c_i dominates c_j in the control flow graph or vice versa. Therefore, every execution from the program entry point to c_j goes through c_i or vice versa [5]. $[\![c_i.v]\!] = [\![c_j.v]\!]$ represents that $c_i.v$ and $c_j.v$ are semantically equivalent. $c_i.P = c_j.P$ means that c_i and c_j are the same kind of checks (e.g., they are both ASan checks, which can be recognized with pattern matching; cf. Sec. 4.1). When c_i and c_j satisfy the given condition and $c_i \in \text{dom}(c_j)$, c_j can be removed because if c_j is executed, c_i must be executed and they check the same property. The given condition specifies the functional equivalence of c_i and c_j. However, computability theory (e.g., Rice's theorem [29]) suggests that it could be very difficult, if possible at all, to assert $[\![c_i.v]\!] = [\![c_j.v]\!]$ for nontrivial programs. Moreover, performing control flow analysis to recover the dominator tree (e.g., $\text{dom}(c_j)$) information can be challenging and lead to false alarms, especially for cases where points-to analyses are extensively used in performing control flow analysis.

Given the theoretical challenge of identifying redundant sanitizer checks, we instead propose a practical approximation to identify *likely redundant checks*. Our approximation extracts both code coverage patterns and static input dependency patterns of checks (cf. Sec. 4); two checks are deemed "redundant" when they yield identical dynamic and static patterns. Specifically, we search for a pair of checks c_i and c_j and flag them as redundant if all the following conditions hold:

- c_i and c_j have correlated dynamic code coverage patterns, when executing the software with a nontrivial amount of workload inputs. Here, coverage patterns are checked regarding their "correlation", such that they can be identical, or one check's coverage pattern can subsume the other's pattern (see Sec. 4.4 for details).
- $c_i.P(c_i.v)$ and $c_j.P(c_j.v)$ are approximately equivalent w.r.t. static data dependency patterns deduced by our technique: $[\![c_i.P(c_i.v)]\!] \approx [\![c_j.P(c_j.v)]\!]$.

The first condition can be determined by instrumenting and profiling the program, and for the second, we assert

[1] We scope SANRAZOR to single threaded programs. See Sec. 4 for further discussion of application scope.

$[\![c_i.P(c_i.v)]\!] \approx [\![c_j.P(c_j.v)]\!]$ by checking the data dependency of two check inputs (see Sec. 4.4 for technical details).

4 Design

Fig. 1 depicts the workflow of SANRAZOR. SANRAZOR starts by identifying both user-defined checks and sanitizer checks (Sec. 4.1). It then instruments each check to record code coverage patterns (Sec. 4.2). We select a suitable workload (**Workload** in Fig. 1) and run the instrumented program with this workload to gather code coverage for each check. SANRAZOR then performs static analysis to construct data dependency graphs per check input and extract static patterns (Sec. 4.3). After obtaining static and dynamic characteristics for each check, SANRAZOR conducts an *unsound* check redundancy analysis (Sec. 4.4). The dynamic and static patterns are both analyzed, and checks with identical patterns will be marked as redundant and removed. The program with remaining checks will be compiled into an executable.

Application Scope. We implement SANRAZOR to analyze LLVM intermediate representation (IR) [17] and remove redundant ASan and UBSan checks. While the current implementation focuses on C/C++ programs, SANRAZOR does not rely on any specific features of C/C++. Therefore, programs written in any programming language can be analyzed, as long as they can be compiled to LLVM IR. Static and dynamic patterns leveraged by SANRAZOR are orthogonal to particular sanitizer implementations; hence, in principle SANRAZOR can reduce checks of different sanitizers. Contrarily, existing work often aims to flag useless checks with dedicated program analysis, while our approach generally circumvents this limitation. See Sec. 8 for comparisons with existing research.

Application Scenario. The focus and typical application scenario of SANRAZOR are to practically accelerate sanitization-enabled programs in production usage. When a production software cannot afford all the sanitizer checks, SANRAZOR can help effectively remove those checks that are least useful in terms of discovering unique problems. As will be shown in Sec. 6, SANRAZOR can reduce the overhead of ASan and UBSan significantly without primarily undermining vulnerability detectability. Moreover, SANRAZOR may be combined with complementary approaches to further reduce the overhead of sanitizer checks. For example, by combining SANRAZOR with ASAP, it is plausible to run these sanitizers in production (at 7% overhead) for their security benefits and vulnerability detectability. Thus, we believe users should generally incline to accept a low overhead (e.g., less than 10% when combining ASAP and SANRAZOR) for improved security and vulnerability detectability compared to running without sanitizer checks. In contrast, enabling full ASan can incur much higher cost (e.g., around 73.8%, as reported in Sec. 6) and is thus unrealistic in production. In practice, we would encourage users to explore combining SANRAZOR with other sanitization optimization tools [9, 37, 44] which share generally orthogonal focuses with SANRAZOR. We give further discussion and comparison with contemporary research works in Sec. 8.

4.1 Check Identification

We start by discussing how ASan and UBSan checks are identified. As aforementioned, each check can be represented as a comparison instruction followed by a control-flow transfer instruction in LLVM IR statements:

```
1 %o = icmp cond %a, %b
2 br i1 %o, label %bb1, label %bb2
```

where `%a` and `%b` are two LLVM IR identifiers, and `icmp` compares `%a` and `%b` w.r.t. the condition specified by `cond` (equal, greater than, etc.). A one-bit comparison output will be stored in `%o`, which is subsequently consumed by the control-flow instruction `br`. In case `%o` equals to one (i.e., the condition evaluates to "True"), the control flow will be transferred to the basic block pointed by `%bb1`; otherwise the basic block pointed by `%bb2` will be executed.

To distinguish sanitizer checks from user checks (i.e., branches in the source code), we search for calls to specific functions that are used by sanitizers. Specifically, a condition represents an ASan check, if a call to `_ASan_report` can be found in blocks pointed by label `%bb1` or `%bb2`. Similarly, a call to the `_UBSan_handle_XXX` function indicates the corresponding condition represents a UBSan check. Note that `XXX` denotes the name of an undefined behavior that this particular UBSan check detects. Overall, while ASan checks are designed to capture memory access errors, UBSan subsumes a much broader set of defects. The type of checked undefined behaviors can be seen from the handler name above, and indeed, the corresponding `icmp` statements can have different constant operands, characterizing the checked undefined behavior types.

4.2 Dynamic Check Pattern Capturing

SANRAZOR captures the dynamic patterns of checks by instrumenting the LLVM IR and inserting a counter statement before the `br` statement of identified checks (see the sample code in Sec. 4.1). We count how many times the control-flow statements are executed. We also record how many times the true and false branches are taken, by checking the one-bit operand of the control-flow statement. Sanitizer checks can be configured to abort the process or output an alert. To smoothly collect dynamic coverage patterns, we configure sanitizer checks to "alert users" instead of aborting the process. The collected coverage patterns will be used to identify redundant checks (cf. Sec. 4.4).

Workload Selection. Ideally, the more execution traces the selected workload can cover, the more comprehensive the dynamic patterns could become. SANRAZOR uses *default test cases* shipped with software to record dynamic patterns.

Our observation shows that the runtime overhead is primarily caused by sanitizer checks on *hot paths*. The shipped test cases typically suffice covering hot paths and derive nontrivial coverage patterns of most sanitizer checks. If one sanitizer check is never covered, it is *not* removed, since no dynamic pattern analysis is available. Sec. 7.4 presents further empirical evidences and discussions regarding workload selection.

4.3 Static Check Pattern Capturing

A user check or sanitizer check asserts certain program properties with a comparison statement (i.e., `icmp` in Sec. 4.1). For the static phase, we extract data-flow information from operands of each `icmp` statement. The extracted data flow facts constitute the static feature of a check.

To this end, SANRAZOR performs backward-dependency analysis to construct the data dependency graph for the branch condition operand. The analysis starts from the checked condition in the control-flow statement (the `br` instruction in Sec. 4.1; recall that `br` takes a LLVM identifier of one bit as its input). The checked condition register has a data dependency on two operands of the comparison statement. The constructed dependency tree will be used to constitute the static patterns of each check (see Sec. 4.3.1). The backward traversal will be stopped when we encounter terminal operands, including constants, `phi` [46] nodes, global variables, and function parameters. Also, when encountering function calls during the traversal (e.g., `malloc`), instead of performing heavyweight inter-procedural analysis, we take all function parameters as the dependency of the function return value.

4.3.1 Extracting Static Features with Three Schemes

After constructing the value dependency tree for the operand of the control-flow instruction `br`, the next step is to extract static features from the dependency tree. At this step, we design three schemes (*L0*, *L1*, and *L2*) by calibrating the extracted static features. Three schemes are designed as follows:

- *L0*, which gathers all the leaf nodes on the dependency tree into a set.
- *L1*, which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set except constant operands from the comparison statement (`icmp` instruction) associated with each sanitizer or user check.
- *L2*, which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set.

L0 collects all the leaf nodes into a set while *L1* and *L2* further canonicalize the constructed set by removing constant leaf nodes. In other words, we might treat checks for the following two pointer dereferences as "redundant", although they check different program properties:

```
1 int a = *ptr;   // ASan check on ptr
2 int b = *(ptr + 4);   // ASan check on (ptr+4)
```

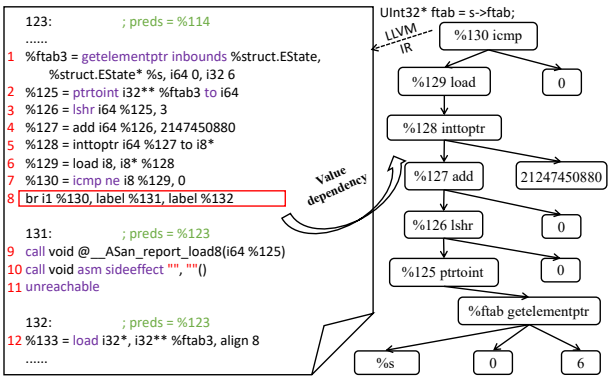


Figure 2: Tracing static data flow dependency.

With ASan enabled, the first and second ASan checks would take `ptr` and `ptr+4` as the inputs. Since both *L1* and *L2* schemes would eliminate constants (i.e., `4` for this case) from the leaf node set, these two checks are treated as redundant by *L1* and *L2*. Nonetheless, *L1* and *L2* schemes would unlikely miss discovering bugs derived from *pointer arithmetics*, in the sense that if an expression using pointer arithmetic (e.g., `ptr+4`) can provoke sanitizer check alerts, the pointer in the expression (i.e., `ptr`) is *presumably invalid* and provokes sanitizer check alerts as well. We present further discussion regarding security considerations in Sec. 4.3.2.

Overall, scheme *L1* and *L2* relax the notion of check "equivalence" by distilling pointer arithmetic expressions while still preserving rich information of the built dependency tree. Also, *L1* is designed to retain the constant operand of `icmp` statement associated with each check. As aforementioned, UBSan uses constant operands in the `icmp` statement to assert different program properties, keeping these specific constants can help to distinguish UBSan checks of different types. Sec. 6 further presents empirical results regarding each strategy; consistent with our intuition, evaluation results (Sec. 6.1) show that the relatively more aggressive scheme *L2* can help to identify more redundant checks and reduce runtime overhead. Moreover, Sec. 6.2 shows that even *L2* can still help to discover 33 (out of in total 38) CVEs from complex software. SANRAZOR provides all three schemes to extract static patterns, and we leave it to users to decide which one to use.

Fig. 2 illustrates feature extraction with an example. Once a path condition statement (line 8; the call statement on line 9 indicates this condition belongs to an ASan check) is identified, we trace the data dependency on the condition (`%130`) and construct a dependency tree. The dependency recovery forms a depth-first search, and we stop the search when encountering terminal nodes (e.g., the constants in Fig. 2). The recovered dependency tree will be used for comparison, following one of the schemes noted above.

4.3.2 Security Consideration

As previously mentioned, we deem that the *L1* and *L2* schemes would unlikely miss discovering bugs derived from

pointer arithmetics, given that if `ptr+4` is invalid, checking `ptr` is presumably sufficient to reveal the issue in practice. However, there are few corner cases, i.e., if `ptr` points to the end of an allocated memory, then `*ptr` is safe while `*(ptr+4)` corrupts. Similarly, `ptr` may point before the start of an allocated memory chunk: `*ptr` thus corrupts whereas `*(ptr+4)` is safe. As clarified in **Application Scenario** in Sec. 4, we focus on practically accelerating sanitization-enabled programs. Users concerned about "sophisticated attackers" can use *L0* or resort to full sanitization. When facing active attackers, another optimization opportunity is to first identify program attack interface, and then shave sanitizer checks out side those security sensitive code fragments. To do so, users can first employ information flow analysis techniques (e.g., taint analysis [31]). We leave it as future work to explore this direction.

4.3.3 Extension Using Static Analysis

As discussed in Sec. 3.2, computability theory suggests that it is difficult, and in general theoretically impossible, to rigorously establish the equivalence of two arbitrary nontrivial code fragments. Nevertheless, in practice, it is feasible to use static analysis to identify (likely) equivalent checks. In particular, we envision that symbolic techniques, e.g., (under-constrained) symbolic execution [27] and constraint solving, can be used to prove the *equivalence* of sanitizer checks.

We have observed a line of research seeking to perform code equivalence checking, by first collecting program input-output relations using symbolic execution [10, 19, 21]. Then, given symbolic constraints representing input-output relations of two code fragments, constraint solver can prove that these two code fragments are equivalent (suppose side effects are not considered). Moreover, constraint solvers can also be used to prove the inclusion of two symbolic constraints, i.e., deciding whether the satisfiability of one symbolic constraint will always induce the satisfiability of the other constraint. As a result, a potential extension of SANRAZOR is to decide whether check c_i validates a weaker property that can be inferred by a stronger property validated in another check c_j. If so, c_i could be redundant and removed. Overall, using such symbolic techniques could be the follow-up work of SANRAZOR to provide more principled guarantees; the tradeoff would be cost and scalability, given most symbolic execution-based code equivalent checking analyzes only basic blocks or execution traces [10, 20, 21].

In Sec. 7.2, we will show that the proposed technique can induce a number of false positive cases, i.e., treating distinct sanitizer checks as equivalent. However, most false positives discussed in Sec. 7.2 could be solved through intra-procedural static analysis, e.g., differentiating accesses to different fields in the same structure. We leave it as one future work to explore using field-sensitive point-to analysis (e.g., SVF [36, 38]) to alleviate false positives of SANRAZOR. Also, SANRAZOR currently omits to perform inter-procedural analysis, and as a result, sanitizer checks inside two procedures would be treated as different. This design decision may potentially lead to false negative cases (i.e., missing a pair of redundant checks). However, we find that in practice, false negative cases are primarily due to other reasons; see discussion in Sec. 7.3.

4.4 Sanitizer Check Reduction

For each pair of checks, SANRAZOR decides whether one check is redundant to the other, by comparing their static and dynamic patterns. In case two checks are identical w.r.t. *both* static and dynamic patterns, only one check will be retained.

Comparing Dynamic Coverage Patterns. Let the coverage pattern of sanitizer check sc_i be a tuple $\langle sb_i, stb_i, sfb_i \rangle$, where sb_i denotes the total coverage times of sc_i, stb_i and sfb_i represent that sc_i executes its true branch stb_i times and its false branch sfb_i times. Similarly, the dynamic pattern of a user-defined check uc_i can be denoted as a tuple $\langle ub_i, utb_i, ufb_i \rangle$, where ub_i denotes the total coverage times of uc_i, utb_i and ufb_i represent that uc_i executes its true branch utb_i times and its false branch ufb_i times. Then, for dynamic coverage patterns extracted from sanitizer check sc_i and user-defined check uc_i, if they satisfy one of the following conditions, we consider sc_i and uc_i having identical coverage patterns:

$$\begin{aligned} &\text{(a)} \quad (sb_i = ub_i) \wedge \big((stb_i = utb_i) \vee (stb_i = ufb_i)\big) \\ &\text{(b)} \quad (sb_i = utb_i) \wedge \big((stb_i = sb_i) \vee (sfb_i = sb_i)\big) \\ &\text{(c)} \quad (sb_i = ufb_i) \wedge \big((stb_i = sb_i) \vee (sfb_i = sb_i)\big) \end{aligned} \tag{1}$$

The first condition implies that two checks have the same dynamic pattern. This can happen when they reside on the same path and the sanitizer check sc_i's false branch has the same coverage times as the user check uc_i's true or false branch. As illustrated in Fig. 3(a), suppose sc_i and uc_i check the same program property, then the predicates of sc_i and uc_i shall be satisfied and failed for the same numbers of times. The latter two conditions are satisfied when sc_i is guarded by one branch of uc_i and one branch of sc_i is never executed. For instance, to understand the second condition (suppose sc_i and uc_i check the same property; see Fig. 3(b)), whenever uc_i is true, sc_i within its true branch (i.e., $sb_i = utb_i$) should always be evaluated to the same direction, as implied by $(stb_i = sb_i) \vee (sfb_i = sb_i)$.

Similarly, for two sanitizer checks sc_i and sc_j, if they satisfy the following condition, we assume that sc_i has identical dynamic patterns with sc_j (one case shown in Fig. 3(c)):

$$(sb_i = sb_j) \wedge \big((stb_i = stb_j) \vee (stb_i = sfb_j)\big) \tag{2}$$

As mentioned in Sec. 4.2, when collecting the dynamic coverage pattern, we configure sanitizer checks to "alert" users (not abort programs) and collect the coverage patterns for comparison. Depending on the implementation details, the program aborting/alerting routine could be found in either the true branch or the false branch of a sanitizer check. Hence, we

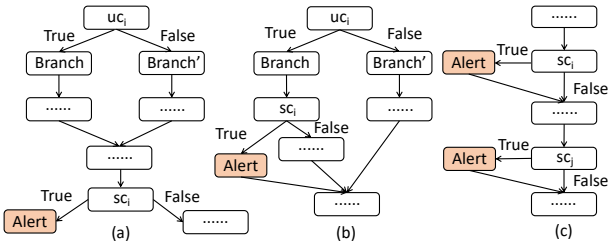


Figure 3: Coverage patterns. Sanitizer checks can be configured to abort programs or alert users. As noted in Sec. 4.2, we configure sanitizer checks to "alert" users (not abort programs) and collect the coverage patterns for comparison.

define a set of general conditions in Formulas 1 and 2, which shall take both cases (i.e., the "aborting" routine resides in true or false branches) into account.

We also note that we did not observe any alerts yielded by sanitizer checks in our experiments. In general, the alert branch of a sanitizer check is much less likely to execute, if at all, than the non-alert branch. The consequence is that a subset of the conditions in Formulas 1 and 2 are in fact used in our experiments. For instance, when the sanitizer check sc_i is within the true branch of uc_i, the second condition in Formula 1 must be satisfied. Nevertheless, in practice, this rarely affects the correctness of our redundancy judgement.

Comparing Static Dependency Patterns. As discussed in Sec. 4.3.1, we provide three schemes to extract static patterns (into sets) from dependency trees of operands in control transfer statements. Given two sets S_i and S_j formed by analyzing two checks c_i and c_j, c_i and c_j are considered to have identical static patterns, in case S_i and S_j are identical.

Removing Sanitizer Checks. SANRAZOR does *not* remove user-defined checks; we prune sanitizer checks in case they are redundant w.r.t. user or other sanitizer checks. Given a pair of likely redundant sanitizer checks c_i and c_j, we remove the check c_j if it was dominated by c_i. Users can also configure SANRAZOR to decide which one to remove. To remove a check, we set the condition of its control-flow statement (see Sec. 4.1) as `false` such that the branch for alerting/aborting will never be executed. This would let the dead-code-elimination of LLVM remove the redundant code.

Extension by Considering Dominating Cases. The aforementioned coverage pattern reasonably flags redundant checks and achieves high effectiveness of reducing overhead incurred by sanitizer checks. Nevertheless, we point that that Formula 2 only considers the equality cases; the dominating cases are not considered, which introduces false positives and the primary false negatives, as will be shown in Sec. 7.2 and Sec. 7.3. An improvement at this step is to maintain the potential dominating checks (denoted as $\mathcal{D}_i$) for sanitizer check c_i. Check c_i can be removed in case its dominating check $c_k \in \mathcal{D}_i$ manifests identical data dependency features with c_i. This extension primarily eliminates false negatives presented in Sec. 7.3.

5 Implementation

SANRAZOR [3] is written primarily in C++ with approximately 2,000 lines of code. We integrate SANRAZOR into the LLVM framework [17] by providing a wrapper of `clang`, namely `SanRazor-clang`. Users can replace `clang` in their building scripts with `SanRazor-clang`. `SanRazor-clang` inserts sanitizer checks to a C/C++ program, and then invokes our follow-up passes to reduce redundant checks. To use SANRAZOR, users need to prepare a reasonable amount of inputs. We note that standard test inputs would usually suffice removing a large amount of sanitizer checks; see our empirical study of workload selection in Sec. 7.4.

6 Evaluation

We give the cost evaluation of SANRAZOR in Sec. 6.1. We measure the reduction accuracy (in terms of vulnerability detectability) in Sec. 6.2 and compare it with ASAP in Sec. 6.3.

6.1 Cost Study

We start by measuring how well SANRAZOR can reduce the performance penalty of sanitizer checks. To this end, we leverage the industry-standard CPU-intensive benchmark suite, SPEC CPU2006, for the evaluation. SPEC CPU2006 contains 19 C/C++ programs. We are able to compile 11 SPEC benchmarks with the Clang compiler (version 9.0.0) and with ASan or UBSan enabled. These 11 test cases are `401.bzip2`, `429.mcf`, `445.gobmk`, `456.hmmer`, `458.sjeng`, `462.libquantum`, `433.milc`, `444.namd`, `470.lbm`, `482.sphinx3`, and `453.povray`. We encountered compatibility issues for the other benchmarks.[2]

Each SPEC benchmark is shipped with a training workload, a testing workload, and a reference workload. Following the convention, we use the training workload to profile these programs and obtain the dynamic patterns of sanitizer checks (see Sec. 4.2). After redundant sanitizer checks are removed by SANRAZOR, the reference workload is used to measure the performance of the optimized programs. We do not use test workload since it leads to much shorter execution time compared with the reference and training workload.

To evaluate the effectiveness of SANRAZOR, we measure the execution time reduction after eliminating redundant checks (referred to as M_0 metrics). We also count the number of removed sanitizer checks (referred to as M_1 metrics), and the execution cost (in terms of CPU cycles) saved by reducing sanitizer checks (referred to as M_2 metrics). M_0 is determined by measuring the execution CPU time. To calculate M_1, we record the total number of sanitizer checks inserted by the compiler and the number of sanitizer checks reduced by SANRAZOR. The calculation of M_2 is consistent with ASAP [40]

[2]Similar compatibility issues were also reported by ASAP [40]. We provide error messages in our artifact [3] for reference.

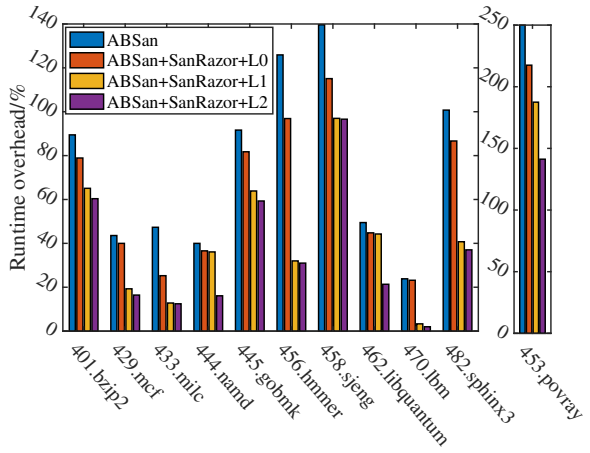


Figure 4: Comparison results w.r.t. M_0 metrics (execution time reduction) on ASan.

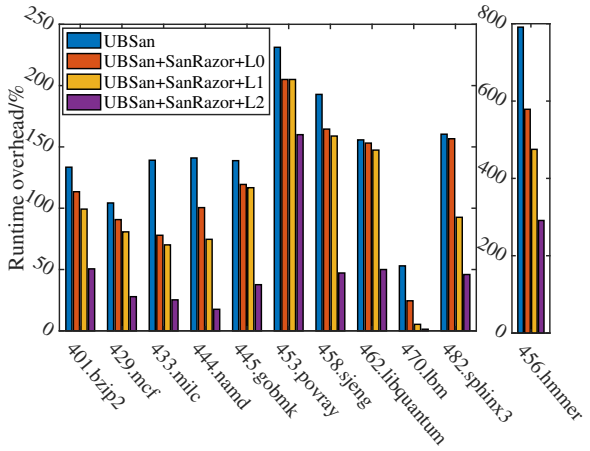


Figure 5: Comparison results w.r.t. M_0 metrics (execution time reduction) on UBSan.

(see comparison with ASAP in Sec. 6.3). In general, each sanitizer check performs a sequence of operations to assert a program property, and for each operation o_i (e.g., loading shadow memory), ASAP predefines a constant f_i denoting how many CPU cycles o_i takes. Suppose a sanitizer check is executed for c times and each execution requires in total F CPU cycles ($F = \sum_{i=1} f_i$), its execution cost is calculated as $c \times F$. We reuse f_i defined in ASAP to compute M_2.

Processing Time. All experiments are conducted on a workstation with an Intel i7-8700 CPU and 16GB memory. We use scripts provided by SPEC to profile programs and collect coverage patterns. It takes on average 323 CPU seconds to profile one SPEC program. The static dependency analysis phase of SANRAZOR takes on average 27 seconds per case.

Cost Evaluation Results

Fig. 4 reports the execution cost that is induced by ASan checks with, and without applying SANRAZOR. The geometric mean runtime overhead increase with full ASan enabled (the blue line) is 73.8%. After reducing redundant ASan checks, performance overhead is reduced by 12.1% (*L0*), 35.2% (*L1*), and 53.5% (*L2*), with the geometric mean remaining overhead being 62.0% (*L0*; the orange line), 35.8% (*L1*; the yellow line), and 28.0% (*L2*; the purple line). The reduced runtime overhead with the *L2* scheme can be up to 91.7% (for `470.lbm`), and the smallest reduction (`445.sjeng`, which exhibits the highest overhead with ASan enabled) still reduces the runtime cost by 30.8%.

Fig. 5 also illustrates the performance overhead for UBSan. In general, the runtime overhead caused by UBSan (geometric mean 154.3%; see the blue line) is much higher than ASan. By reducing redundant checks, the performance overhead can be reduced by 13.7% (*L0*), 35.5% (*L1*), and 75.5% (*L2*), with the geometric mean remaining overhead being 124.4% (*L0*; the orange line), 94.7% (*L1*; the yellow line), and 36.6% (*L2*; the purple line). The reduced runtime overhead of *L2* strategies is up to 97.5% (`470.lbm`), and at least 62.1% (`401.bzip2`).

We measure how many sanitizer checks are reduced by SANRAZOR and the saved CPU cycles. Table 1 reports the portion of reduced checks w.r.t. the total number of checks (M_1). Similarly, it also reports the portion of saved CPU cycles during run time w.r.t. the total CPU cycles taken by sanitizer checks (M_2). SANRAZOR can eliminate on geometric mean up to 29.5% of the ASan checks for the SPEC programs (with *L2* applied), which leads to 41.0% less CPU cycle cost of ASan checks during run time. We also observed a similarly promising trend for the UBSan evaluation. As shown in Table 1, SANRAZOR can eliminate up to 39.3% of UBSan checks for the SPEC programs on geometric mean (with *L2* applied), corresponding to 77.0% less cost during run time.

Sanitizer checks contribute differently to the total execution cost (i.e., some checks are executed far more often than others). For instance, SANRAZOR (with *L2* enabled; see Table 1) eliminates 13.1% of ASan checks in `456.hmmer`. However, the M_2 metrics is reduced by up to 70.4%, indicating that the removed checks are on the program's *hot paths*. Our manual investigation confirms this intuition; more than 99.3% of the runtime overhead caused by ASan stems from one function `P7Viterbi` in module `fast_algorithms.c`, which contains intensive memory access checks within a loop. SANRAZOR successfully identifies many redundant sanitizer checks within this loop, inducing effective check reduction for `456.hmmer`.

We also find some reduced sanitizer checks on the cold path of the benchmarks. For instance, while 22.6% of the UBSan checks are removed from `462.libquantum`, these checks have low runtime coverage and therefore removing them does not significantly improve performance. An aligned trend can be seen from the performance evaluation of `462.libquantum` in Fig. 5. Our manual study indicates that for `462.libquantum`, sanitizer checks in `gates.c` (on the hot path) contribute more than 99.2% of the total runtime cost. However, these checks assert different undefined behaviors and cannot be flagged as "redundant" by SANRAZOR.

Table 1: Evaluation results w.r.t. M_1 (number of removed sanitizer checks) and M_2 (saved CPU cycles by reducing sanitizer checks). Note that "empty cells" for `imageworsener`, `zziplib`, `libzip`, `graphicsmagick`, `jasper`, `potrace`, and `mp3gsin` are *not* due to setup errors; they indicate those CVEs are not discovered by the corresponding ASan (or UBSan) checks.

Benchmark	ASan-M_1			ASan-M_2			UBSan-M_1			UBSan-M_2		
	L0	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*
`401.bzip2`	22.4%	54.4%	58.1%	4.3%	30.3%	34.2%	38.7%	54.8%	66.0%	27.3%	37.9%	68.1%
`429.mcf`	10.2%	53.0%	60.9%	3.0%	46.6%	60.1%	35.0%	51.8%	76.2%	37.8%	47.6%	86.0%
`445.gobmk`	5.2%	23.4%	26.6%	7.2%	33.7%	41.0%	12.6%	21.6%	51.3%	21.4%	23.3%	73.9%
`456.hmmer`	5.9%	11.7%	13.1%	14.4%	70.3%	70.4%	8.2%	11.0%	14.8%	49.2%	60.7%	78.3%
`458.sjeng`	5.9%	12.6%	13.4%	4.4%	34.4%	36.7%	12.1%	18.3%	51.0%	20.7%	25.2%	79.2%
`462.libquantum`	7.4%	16.3%	22.6%	0.8%	1.4%	2.4%	12.7%	15.6%	26.9%	0.8%	0.8%	58.8%
`433.milc`	23.5%	32.5%	33.5%	35.8%	80.9%	82.7%	27.6%	42.2%	54.6%	51.0%	60.6%	83.6%
`444.namd`	6.4%	18.9%	24.0%	10.2%	29.8%	57.7%	8.7%	16.0%	26.2%	40.4%	54.1%	84.8%
`470.lbm`	1.6%	68.5%	72.1%	0.0%	88.7%	92.5%	17.7%	48.2%	51.3%	46.0%	92.5%	97.6%
`482.sphinx3`	10.7%	27.1%	32.5%	2.5%	56.9%	58.3%	18.2%	23.7%	40.0%	11.9%	45.3%	67.2%
`453.povray`	7.2%	9.5%	21.2%	2.3%	12.1%	69.1%	11.1%	11.9%	22.6%	22.6%	24.0%	75.5%
`autotrace`	12.2%	27.6%	35.7%	22.4%	65.4%	73.1%	20.6%	25.2%	39.0%	48.6%	57.5%	78.3%
`imageworsener`	-	-	-	-	-	-	26.8%	37.1%	53.3%	17.8%	21.6%	64.0%
`lame`	9.5%	38.5%	40.8%	11.0%	57.5%	74.9%	23.3%	34.1%	47.5%	17.0%	46.6%	71.4%
`zziplib`	3.8%	20.4%	23.9%	12.9%	80.2%	90.3%	-	-	-	-	-	-
`libzip`	6.2%	19.9%	27.8%	1.0%	3.9%	44.9%	-	-	-	-	-	-
`graphicsmagick`	1.2%	4.5%	5.8%	20.1%	49.4%	63.3%	-	-	-	-	-	-
`tiff`	7.8%	21.7%	29.8%	0.2%	2.1%	2.6%	12.3%	15.8%	21.7%	7.6%	10.5%	65.6%
`jasper`	-	-	-	-	-	-	12.8%	17.3%	25.9%	19.6%	20.6%	69.6%
`potrace`	13.0%	31.2%	38.8%	5.4%	41.9%	48.7%	-	-	-	-	-	-
`mp3gsin`	11.6%	43.6%	46.0%	4.8%	74.8%	78.4%	-	-	-	-	-	-

6.2 Vulnerability Detectability Study

This section explores whether sanitizer checks marked as redundant are true positive w.r.t. Definition 1 given in Sec. 3.2. This study reflects the accuracy of SANRAZOR. We select a number of programs with CVE vulnerabilities from an actively-maintained CVE list [22, 23], which documents procedures to compile each program and reproduce its CVEs. We select programs based on whether it can be successfully compiled and whether their documented CVEs can be triggered by the shipped inputs, and whether those CVEs can be detected by ASan/UBSan.

Ten programs (with in total 38 CVEs) are used for this evaluation. These ten programs are not cherry-picked; when selecting these ten programs, we checked each program in the CVE program list from the beginning [22, 23] and skipped only those that could not be properly set up for our study. For the evaluation setup, we start by compiling the provided source code with ASan or UBSan enabled. We report that those 38 CVEs can all be triggered by at least one input provided by the CVE list [22, 23], and after enabling ASan or UBSan, all the CVE-triggering inputs can be captured by either ASan or UBSan (i.e., all CVEs can be discovered). We then use SANRAZOR to perform check reduction with three schemes (*L0*, *L1*, and *L2*) and check whether after pruning, the CVE-triggering inputs can still be captured. Table 2 reports the evaluation results in terms of which CVE vulnerabilities can still be discovered by the pruned checks.

The static analysis phase of SANRAZOR takes on average 17.5 CPU seconds to process one program. Programs in the CVE list are typically shipped with a small number of inputs, including both regular and bug-triggering inputs. At this step, we use regular inputs to generate dynamic coverage patterns and shave sanitizer checks. We then test if bugger-triggering inputs can still be captured by the remaining sanitizer checks. The execution of most programs takes negligible amount of time (on average 1.5 CPU seconds). Overall, their shipped inputs are used for asserting *functionality*, not for *benchmarking*. Regarding M_0 metrics, we report that SANRAZOR reduces geometric mean runtime overhead caused by ASan from 24.9% to 15.8–22.4% (depending on the different reduction schemes). Similarly, geometric mean overhead incurred by UBSan on these CVE programs is reduced from 7.0% to 1.5–5.0%. We also report that we observed significant hot-path vs. cold-path distinction of these CVE programs, given their inputs of relatively low comprehensiveness. Nevertheless, we note that in case a sanitizer check is not covered by a shipped regular input, we will not even have its dynamic coverage pattern, and thus will not remove it. This way, vulnerabilities relevant to this check can be protected.

We then evaluate these programs w.r.t. the M_1 and M_2 metrics. As shown in Table 1, there is no significant gap comparing the number of checks removed from the CVE and SPEC programs, e.g., 9.9% vs. 8.2% with ASan-M_1&*L0* and 19.1% vs. 19.2% with UBSan-M_2&*L0*. Therefore, Table 2 shows, even if approximate reduction is achieved, almost all CVEs can still be discovered. The rest of this section elaborates on each case. We discuss all *false positive* cases (i.e., missed CVEs due to incorrectly removed checks) exposed in Table 2 in Sec. 7.

`autotrace` is an open-source software written in C, transforming bitmap images into vector images. We reproduce

19 CVEs in six modules of `autotrace-0.31.1`. The UBSan checks avert nine CVEs, including eight signed integer overflows (CVE-2017-9161∼9163, CVE-2017-9183∼9187), and one left shift of negative value (CVE-2017-9188). The rest are heap buffer overflows detected by ASan checks (CVE-2017-9167∼9173, CVE-2017-9164∼9166). As reported in Table 2, all of these CVEs can still be detected, after eliminating redundant sanitizer checks with the *L0* and *L1* schemes. Nevertheless, *L2* generates two false positives for the UBSan cases (see Sec. 7.2).

`imageworsener` is a C/C++ library supporting scaling and processing images with multiple formats. We evaluate the performance of SANRAZOR with five CVEs found in `imageworsener-1.3.1`, including two divide-by-zero CVEs (CVE-2017-9201∼9202) in `imagew-cmd.c`, two null pointer dereferences (CVE-2017-9204∼9205) in `imagew-util.c`, and one out-of-bounds access (CVE-2017-9203) in `imagew-main.c`. As reported in Table 2, both *L0* and *L1* strategies in SANRAZOR can detect all these CVEs, while the *L2* strategy generates one false positive in CVE-2017-9203. As for the performance gain, Table 1 shows encouraging results by saving up to 64.0% w.r.t. M_2 metrics.

`lame` is a MP3 encoder written in C. We reproduce two CVEs in two C files of `lame-3.99.5`, where one is a division by zero vulnerability detected by UBSan in `get_audio.c` (CVE-2017-11720), and the other is a heap buffer overflow detected by ASan in `util.c` (CVE-2015-9101). Table 2 shows that both UBSan and ASan can still discover these two CVEs for all three settings. For the inserted ASan checks, Table 1 reports that up to 74.9% cost can be saved. Similar trends (up to 71.4%) can be observed for UBSan.

`zziplib` is a lightweight C library for extracting data from a zip file. Two CVEs (CVE-2017-5976∼5977) in `zziplib-0.13.62` are evaluated, and both of them are caused by heap buffer overflow in `memdisk.c`. Our evaluation shows that after check reduction with all three settings, both CVEs can still be discovered by ASan.

`libzip` is a C library for processing zip files. There is a use-after-free CVE (CVE-2017-12858) in `libzip-1.2.0`, whose triggering-inputs can be captured by ASan. As shown in Table 2, only the *L2* scheme mistakenly eliminates the corresponding ASan check and missed one CVE.

`graphicsmagick` is a tool for viewing and editing commonly-used file formats including PDF, PNG, and JPEG. We evaluate SANRAZOR on CVE-2017-12937, a heap use-after-free vulnerability in `sun.c`. Table 2 shows that this CVE can be detected by ASan checks for all three settings.

`libtiff` is a library for viewing and editing tiff images. Four CVEs, including two heap buffer overflows (CVE-2016-10270, CVE-2016-10271), one stack buffer overflow (CVE-2016-10095) and one division-by-zero (CVE-2017-7598), are used to evaluate SANRAZOR. Experimental results show that SANRAZOR can detect all CVEs in all three settings.

`jasper` is also a complex image processing tool (with over 40K LOC). We evaluate SANRAZOR on CVE-2017-5502, a left shift of a value less than zero that can be detected when UBSan checks are fully enabled. Our evaluation shows that after check reduction, this CVE can still be discovered by the remaining UBSan checks.

`potrace` is a commonly-used C tool for converting bitmaps into smooth and scalable images. We evaluate SANRAZOR on CVE-2017-7263, a heap buffer overflow in `bitmap.c` of `potrace-1.2`. Table 2 shows that ASan can still discover this CVE after redundant checks are removed in all three settings.

`mp3gain` is a C library for analyzing MP3 files. We reproduce five CVEs in `mp3gain-1.5.2`, including one null pointer dereference (CVE-2017-14406) in `interface.c`, one buffer overflow (CVE-2017-14407) in `gain_analysis.c`, and two buffer overflows (CVE-2017-14408∼14409) in `layer3.c`. As reported in Table 2, one false positive is found, where both the *L1* and *L2* schemes over-aggressively remove the sanitizer check for detecting CVE-2017-14406.

The evaluation has demonstrated the promising and practical accuracy of SANRAZOR: vulnerability detectability is unlikely impacted even if we reduce sanitizer cost. Also, readers may suspect that if during the profiling phase software is not "stressed enough", SANRAZOR will elide a small set of checks and, as expected, catch the respective CVEs. However, we again note that *all* sanitizer checks detecting the 38 CVEs are covered during the profiling phase. That is, our observation — at least 33 CVEs are discovered by the remaining checks — is not due to "under-stressed profiling", rather, the corresponding checks are not incorrectly deemed redundant.

6.3 Comparison Study

We compare SANRAZOR with the closely related work, ASAP [40]. ASAP reduces sanitizer checks with high cost in order to satisfy an overhead budget specified by users. In contrast to SANRAZOR, ASAP does not identify "likely identical checks". Given an overhead budget T, ASAP first estimates the performance cost that a sanitizer check c can incur. Sanitizer checks will then be ranked by cost and iteratively removed starting from the most expensive one until the estimated cost is lower than the budget T. We compare SANRAZOR with ASAP by running ASAP on our CVE cases with different overhead budgets, and reports whether the known CVEs can still be discovered (i.e., evaluation conducted in Sec. 6.2). Contrarily, we do *not* evaluate ASAP on the sanitizer overhead it can save. ASAP reduces sanitizer checks to meet a fixed cost budget; it is easy to see that comparing ASAP and SANRAZOR on this matter (i.e., SPEC evaluation in Sec. 6.1) is not reasonable.

In this evaluation, we start by using the default budget of ASAP, 5%, to measure the check reduction. As shown in Table 2, setting the overhead budget to 5% (i.e., Budget_0) prunes 23 out of 38 checks that can detect CVEs. Furthermore, to

Table 2: CVE case study. N denotes the number of CVEs. The "SANRAZOR" and "ASAP" columns report the number of remaining CVEs that can be discovered by sanitizer checks after reduction. ASAP allows to configure arbitrary overhead budget. We evaluate ASAP by using its default budget ($Budget_0$), and with the same overhead as SANRAZOR (i.e., $Budget_1$, $Budget_2$, $Budget_3$ correspond to *L0*, *L1*, *L2*, respectively). See Sec. 6.3 for the setup.

Software	CVE			SANRAZOR			ASAP			
	Type	Sanitizer	N	*L0*	*L1*	*L2*	$Budget_0$	$Budget_1$	$Budget_2$	$Budget_3$
autotrace	signed integer overflow	UBSan	8	8	8	6	6	8	8	8
	left shift of 128 by 24	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	10	10	10	10	0	8	2	2
imageworsener	divide-by-zero	UBSan	2	2	2	2	2	2	2	2
	index out of bounds	UBSan	1	1	1	0	1	1	1	1
lame	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	1	1	1	1	0	1	0	0
zziplib	heap buffer overflow	ASan	2	2	2	2	0	0	0	0
libzip	user after free	ASan	1	1	1	0	0	1	1	1
graphicsmagick	heap use after free	ASan	1	1	1	1	0	1	1	1
libtiff	heap buffer overflow	ASan	2	2	2	2	0	2	2	2
	stack buffer overflow	ASan	1	1	1	1	1	1	1	1
	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
jasper	left shift of negative value	UBSan	1	1	1	1	1	1	1	1
potrace	heap buffer overflow	ASan	1	1	1	1	0	1	1	0
mp3gain	stack buffer overflow	ASan	2	2	2	2	0	2	0	0
	global buffer overflow	ASan	1	1	1	1	0	0	0	0
	null pointer dereference	ASan	1	1	0	0	1	1	1	1
In total			38	38	37	33	15	33	24	23

present a fair comparison with SANRAZOR, we iterate each tested program and put their CVEs into different types (the second column of Table 2). We then use the M_2 metrics of SANRAZOR in terms of each CVE type as three different overhead budgets of ASAP (i.e., $budget_1$, $budget_2$, $budget_3$). For instance, the "divide-by-zero" CVE of `imageworsener` can be captured by UBSan checks. After applying SANRAZOR (with *L0*, *L1*, and *L2* schemes) on `imageworsener` with full UBSan checks enabled, we report that the remaining M_2 overhead is 82.2%, 78.4%, and 36.0%, respectively. Then, ASAP is configured to take these three remaining M_2 overhead as its overhead budget and performs sanitizer check reduction. As shown in Table 2, two CVEs of `imageworsener` (the "divide-by-zero" row) can still be discovered for all three budgets. Overall, ASAP is configured to achieve the same amount of performance cost as SANRAZOR, and we record how many CVEs can still be discovered in this "apple-to-apple" setting.

We record the remaining M_2 overhead (geometric mean 89.2%) for each CVE type after using SANRAZOR with the *L0* scheme enabled. As shown in Table 2, five CVEs cannot be detected when assuming this budget for ASAP. The *L1* and *L2* schemes perform a relatively more tolerant reduction (73.5% and 45.7% geometric mean remaining M_2 overhead), and accordingly, ASAP removes 14 and 15 checks, respectively. We find that considerable critical CVEs are not discovered after using ASAP, since the corresponding checks are in the "hot paths" of test cases, incurring high cost, and therefore are removed. Overall, we interpret the comparison as encouraging, showing that ASAP neglects the important observation of sanitizer redundancy, causing it to fail discovering CVEs on hot paths. Contrarily, Table 2 shows that SANRAZOR can help discover more CVEs after reducing identical cost.

6.4 Combining SANRAZOR with ASAP

We also conduct a case study on `autotrace` to explore how we could achieve potential synergistic effects by combining SANRAZOR with ASAP and reduce the M_2 overhead for production usage. To this end, we explore whether, after applying ASAP, SANRAZOR can find further opportunities for eliminating redundancy that ASAP may have missed.

Specifically, we first set the overhead budget of ASAP to the reasonable, but arbitrary threshold of 30% and run it on `autotrace` with full ASan enabled to remove high-cost checks. ASAP aggressively reduces ASan checks; after reducing the M_2 overhead to 30%, six out of in total 10 CVEs are missed. We then leverage SANRAZOR to identify redundant checks. We report that when applying SANRAZOR with the *L*0 scheme, we observe that the M_2 overhead can be further reduced to 7.0%, *without missing any additional CVEs*. In contrast, using ASAP with this aggressive budget (7.0%), would reduce too many ASan checks and miss all 10 CVEs (cf. Table 2).

SANRAZOR Extension and Future Directions. Note that ASAP primarily focuses on shaving costly checks on the hot paths, which indicates promising potential of fine-tuning SANRAZOR's schemes to be more adaptive by taking cost into consideration. SANRAZOR can thus be extended to apply L0/L1 schemes to shave checks with low costs and L1/L2 schemes to shave checks with high costs. This should better balance performance and safety, rather than using the same scheme to all checks.

Our study in this section sheds light on the significant potential in combining SANRAZOR with previous works (e.g., [14,37,44]) and exploring their synergistic effects, since the strategies for which checks to remove are generally orthog-

Table 3: Quantitative analysis of the removed sanitizer checks.

Software	Sanitizer Type	#Reduced Checks	#Identical Checks	#Correlated Checks
401.bzip2	UBSan	11,406	6,562	4,844
autotrace	ASan	2,434	460	1,974

onal. Also, in addition to the combination strategy demonstrated above, we envision using other feasible schemes to combine SANRAZOR and ASAP. For instance, given a user-specified budget, we first use SANRAZOR to remove all the likely redundant checks, and if the budget is still not met, we use ASAP to remove further checks. We leave it as future work to explore other practical methods to combine SANRAZOR with existing sanitizer reduction tools.

This section demonstrates combining SANRAZOR with ASAP to reveal more debloating opportunities. In addition, we also expect that by using SANRAZOR to shave likely equivalent checks, sanitizer-guided security applications can be boosted. For instance, by shaving redundant checks, sanitizer-guided fuzz testing tool, ParmeSan [26], may have a higher throughput and likely find more bugs within a given time budget. We leave this as one future work to explore using SANRAZOR to boost ParmeSan.

7 Discussion

7.1 Characteristics of Removed Checks

This section further explains the characteristics of the checks that are removed. Given the observation that thousands of sanitizer checks are inserted into each program, we deem investigating all test cases infeasible. Rather, we manually checked SPEC program `401.bzip2` (with UBSan) and CVE program `autotrace` (with ASan) and analyzed check reduction patterns (Table 3). The two programs contain in total 24,132 checks (17,272 in `401.bzip2` and 6,860 in `autotrace`). SANRAZOR with the *L2* scheme enabled removes 66.0% sanitizer checks from `bzip2` and 35.7% checks from `autotrace`. We studied each removed check to identify two common patterns that we refer to as "identical checks" and "correlated checks". Below, we present typical cases for each category.

Checks that are identical with other checks. Sanitizer checks of this class have the same functionality as other checks. Consider the code snippet in `bzip2.c` as follows:

```
1 void BZ_blockSort(EState* s){
2  UInt32* ptr   = s→ptr;
3  UChar* block = s→block;
```

where two UBSan checks are inserted to check whether `s` is a null pointer. However, these two checks indeed assert the same property and are therefore identical with each other. Removing one of them can still ensure that the null pointer is detected. SANRAZOR will remove one of them since their coverage patterns are exactly the same and their control-flow statements (i.e., the `br` statement in LLVM IR) have condition operands of identical dependency trees.

Checks that are correlated with other checks. SANRAZOR removes this class of checks since they have the same dynamic and static patterns with other checks, indicating strong correlation with each other, as, for instance, for different pointer arithmetic expressions over the same pointer (as discussed in Sec. 4.3.1). Consider CVE-2017-9169 as an example:

```
1  *(temp++)= buffer[xpos * 3 + 2];  //line 353
2  *(temp++)= buffer[xpos * 3 + 1];  //line 354
3  *(temp++)= buffer[xpos * 3];  //line 355
```

which is a heap buffer overflow in line 353 of file `input_bmp.c` (`temp` in above code). When enabling ASan, three sanitizer checks (sc_1,sc_2,sc_3) are inserted for this case to check the shadow memory of pointer `temp` (in line 1-3 above). When using SANRAZOR (with *L1* or *L2* enabled) to analyze this case, all three checks exhibit identical dynamic and static patterns. Thus, two checks will be removed. Although the heap buffer overflow in CVE-2017-9169 roots in the invalid memory access of pointer `temp` in line 353, ASan can presumably detect the vulnerability when using any of the other two checks.

7.2 False Positive Analysis

SANRAZOR can also induce false positives (i.e., unique checks that are removed), since the captured dynamic patterns only provide statistical information of sanitizer checks and the static pattern sets used for the redundancy analysis could be optimistic as well. Below, we provide detailed analysis of all five false positive cases caused by the *L2* scheme of SANRAZOR (the false positive case of using *L1* scheme is also subsumed).

`CVE-2017-9203` is an index out of bounds vulnerability in `imagew-main.c` of `imageworsener-1.3.0` as follows:

```
1 int_ci = &ctx→intermed_ci[intermed_channel];
2 output_channel = int_ci→
      corresponding_output_channel;  // CVE
3 out_ci = &ctx→img2_ci[output_channel];
```

Specifically, `ctx` is an input argument of the enclosing function (line 2), which has a `struct iw_context*` type. When compiling this module with UBSan, three sanitizer checks will be inserted to detect index out of bounds, type check, and pointer overflow vulnerabilities on line 2. Recall as introduced in Sec. 4.1, these three UBSan checks can be differentiated by the constant operands of their associated `icmp` statement. Nevertheless, since all these checks take the memory address of `ctx` as its inputs, they have the same value dependency on `ctx` when using the *L2* scheme (recall *L2* eliminates all constants). Therefore, SANRAZOR will identify two of them as redundant sanitizer checks and remove them, causing UBSan to fail reporting the index out of bounds vulnerability in this CVE. However, if SANRAZOR is configured with *L1*, these checks can be kept since the constant parameter used to differentiate these three UBSan checks are preserved.

CVE-2017-12858 is a use-after-free vulnerability detected by ASan in `zip_buffer.c` of `libzip-1.2.0`. As shown below, variable `buffer` of `zip_buffer_t*` type attempts to access its element `free_data` in the `if` condition (line 5):

```
1 void _zip_buffer_free(zip_buffer_t *buffer){
2  if (buffer == NULL) return;
3  if(buffer→free_data){  // CVE
4   free(buffer→data);
```

When ASan is enabled, a check is inserted to assert whether the memory pointed by `buffer` has been freed before accessing its element `free_data`. SANRAZOR with the *L2* scheme enabled eliminates this check since it has the same dynamic coverage pattern with two user checks (two `if` conditions on line 2 and line 3), which also share the same data dependency pattern (since variable `buffer` is used for all three user and sanitizer checks). However, the check inserted by ASan performs shadow memory calculation, which indeed depends on different constant values with two user checks. Therefore, SANRAZOR with *L1* scheme can retain this check.

CVE-2017-9184 is a signed integer overflow in `autotrace=0.31.1` reported by UBSan. It is derived from a heap memory allocation in `input-bmp.c` as follows:

```
1 XMALLOC(image, width * height
2         * 1 * sizeof(unsigned char));  // CVE
3 ypos = height - 1;
4 switch (...) {
5   case 1: {
6     while (ypos >= 0 && xpos <= width) { ...
```

`XMALLOC` allocates memory buffers on the heap by taking the second parameter as the buffer length, where a UBSan check is inserted in line 1 to detect the signed integer overflow when multiplying `height` with `width`. Moreover, we find a user check in the `while` loop condition (line 6), which shares the same value dependency with this critical sanitizer check (`ypos` derives from `height` and `xpos` is a constant). Therefore, SANRAZOR with *L2* enabled removes this check. Nevertheless, the inserted UBsan check and `while` condition assert different program properties, exposing a false positive. In contrast, this false positive can be avoided when using the *L1* scheme, since the user check also depends on constant value `0`, exhibiting different value dependency patterns with the inserted sanitizer check by UBSan.

CVE-2017-9187 is a signed integer overflow vulnerability found from `autotrace=0.31.1`. Consider the following code snippet showing the CVE in `input-bmp.c`:

```
1 unsigned char *temp2, *temp3;
2 XMALLOC(image, width * height
3         * 3 * sizeof(unsigned char));  // CVE
4 temp3 = image;  //another UBSan check
```

When compiling this code snippet with UBSan enabled, a sanitizer check is inserted to check whether the second parameter of `XMALLOC` can incur an integer overflow. Also, another check is added to detect whether the pointer `temp3` is null. Since `temp3` points to `image` and the value of `image` is assigned by `XMALLOC`, the parameter of the second UBSan check depends on variable `width` and `height` (recall as mentioned in Sec. 4.3, for interprocedural analysis the function call output, `image` for this case, conservatively depends on all function parameters). Therefore, SANRAZOR with *L2* enabled will consider this sanitizer check to be redundant with the assertion, while the *L1* scheme would not, as these two checks can be differentiated by their constant parameters.

CVE-2017-14406 is a null pointer dereference found in `interface.c` of `mp3gain-1.5.2`. Consider the code below:

```
1 int sync_buffer(PMPSTR mp,int free_match) {
2  for (i=0; i<mp→bsize; i++) // CVE
3  { ... }
4  struct frame *fr = &mp→fr;
5  h = head_check(head,fr→lay);
```

Two ASan checks are used to check `mp` and `fr` when accessing their struct elements `bsize` (line 2) and `lay` (line 5), respectively. `fr` is initialized with `mp->fr` (line 4), which depends on the function parameter `mp` (line 1). That is, the two ASan checks have identical data dependencies w.r.t. the *L1* and *L2* schemes. SANRAZOR eliminates the first ASan check and becomes incapable of detecting the CVE vulnerability on line 2. However, SANRAZOR with *L0* enabled can differentiate these two checks, since they depend on different constant offsets when accessing the struct elements.

7.3 False Negative Analysis

SANRAZOR could also have false negatives (i.e., redundant checks are not removed). Take the following piece of code in `462.libquantum` for example, where ASan inserts two checks to detect a buffer overflow in `reg->node[i]`. Let the inserted check in line 2 and line 3 as sc_1 and sc_2, respectively. Although sc_2 is redundant with sc_1 for this specific case, SANRAZOR does not recognize sc_2 as a redundant sanitizer check during our experiment, because the dynamic pattern of sc_2 differs from that of sc_1. Such cases are the primary cause for generating false negatives, according to our observations. Nevertheless, Sec. 4.4 has discussed that these false negative cases can be primarily eliminated by considering dominating relations in comparing dynamic coverage patterns.

```
1 for(i=0; i<reg→size; i++) {
2   if(reg→node[i].state & ...) {
3     if(reg→node[i].state & ...) {
```

7.4 Effects of Workload Selection

As mentioned in Sec. 4.2, SANRAZOR relies on dynamic coverage patterns to pinpoint potentially redundant checks. Therefore, in this section, we present study and discussion on the efficiency of check reduction w.r.t. the size of workload. To do so, we incrementally enlarge the workload for profiling `bzip2` record both the number of reduced sanitizer checks

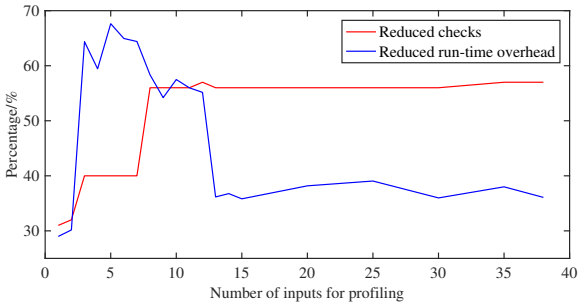


Figure 6: Effects of workload selection evaluation on ASan.

and the runtime overhead caused by ASan. For this study, we configure SANRAZOR with the *L2* scheme for check reduction. We download a `bzip2` testsuite with 38 different inputs from [2] and conduct experiments on `bzip2` (a single-file version from [1]).

As illustrated in Fig. 6, the percentage of reduced sanitizer checks will increase when more inputs are fed to the test case. However, when the number of test inputs are more than eight, the number of reduced checks reach the saturation point at 58.0%. Similarly, the percentage of reduced runtime cost can also become stable when the number of inputs is larger than 12. Also, notice that the blue line will first increase (when the input is less than five), and then decrease until reaching the saturation point (when input is 13). The reason is that with insufficient amount of inputs in the workload, irrelevant checks may exhibit identical dynamic coverage patterns and be treated as "redundant." In other words, SANRAZOR may report false positives when the available inputs are insufficient, and aggressively flag too many "redundant" checks (Sec. 7 discusses false positives and false negatives of this research). In general, when adopting SANRAZOR in real-world scenarios, sufficient inputs are needed for achieving good reduction results and reducing false alarms, but they may not need to be too many.

8 Related Work

Static Check Reduction. Existing research has proposed heavyweight program analyses to elide redundant bounds check by inferring the value ranges of certain variables. For example, some approaches deem checks unnecessary if the value range of an index is below the size of its accessed array [6, 11, 12, 15, 25, 37, 39, 44, 45]. SIMBER [7, 45] uses statistical inference to identify redundant bounds checks from past executions. RedCard [9] flags unnecessary race condition checks by scoping specific "release-free" code region where it is proved that only one race check is needed for each region. BigFoot [28] coalesces race checks on arrays and C structs. Overall, the extensive existing work on this topic focuses on specific types of checks, and cannot be easily generalized to other checks. For instance, [9] identifies a "release-free" region by checking if no lock release synchronization operations (e.g., wait, fork) can be found in that region. Hence, race checks only need to be done once within each region. Scoping such a "safe region" could be very difficult for other checks: to decide such a safe region for ASan, we anticipate to perform expensive alias analysis for every pointer within that region to confirm a checked pointer is never modified. In contrast, SANRAZOR analyze the equivalence of checks in a general and practical way to eliminate duplications.

The most closely related work is ASAP [40], which, like SANRAZOR, is unsound but general. ASAP is designed based on the observation that a few "hot" sanitizer checks account for most of the overhead and that most CVEs are located in the "cold" parts of a program. ASAP removes sanitizer checks with high runtime overhead until the overall overhead meets a user-provided cost budget. Different from SANRAZOR, ASAP does not consider check redundancy and is prone to removing critical checks on a program's hot paths as we have shown.

Run-time Check Reduction. Safe Sulong [30] is a sanitizer that relies on the dynamic compiler of the Java Virtual Machine to reduce checks. Java compilers are capable of eliding certain unneeded checks [42]. However, Safe Sulong can be overly conservative since it eliminates only those checks that are identified as redundant by the compiler.

Some approaches reduce sanitizer checks by runtime partitioning. Kurmus et al. split the kernel into an unprotected and a protected partition to reduce overhead caused by kernel hardening [16]. Bunshin [43] distributes sanitizer checks into different program variants and executes them in parallel to reduce the overall overhead. PartiSan [18] is a runtime sanitizer partitioning tool using control-flow diversity to improve sanitizer efficiency. Varan [13] is a multi-version monitor that uses selective binary rewriting to execute multiple versions of a system (e.g., instrumented by multiple sanitizers). However, these approaches reduce sanitization cost with parallel execution, rather than analyzing redundancy offline.

9 Conclusion

We have presented SANRAZOR, a novel, practical tool for sanitizer check reduction. SANRAZOR identifies redundant checks by analyzing their dynamic coverage patterns and static data dependency patterns. Evaluation on CPU benchmarks and programs with CVEs shows that SANRAZOR can effectively lower the overhead caused by ASan and UBSan, while still retaining high vulnerability detection capability.

Acknowledgments

We thank anonymous reviewers and our shepherd, Shan Lu, for their valuable feedback. We also thank Fuqiang Fan, who proofread an early version of the paper and pointed out an error in Definition 1.

References

[1] Large single compilation-unit C programs. https://people.csail.mit.edu/smcc/projects/single-file-programs/, 2006.

[2] Bzip2 testsuite. https://sourceware.org/git/?p=bzip2-tests.git, 2019.

[3] SanRazor. https://github.com/SanRazor-repo/SanRazor, 2021.

[4] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy*, pages 263–277. IEEE, 2008.

[5] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[6] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–333, 2000.

[7] Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. CHOP: Bypassing runtime bounds checking through convex hull optimization. *Computers & Security*, 90:101708, 2020.

[8] LLVM Developers. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017.

[9] Cormac Flanagan and Stephen N Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming*, pages 255–280. Springer, 2013.

[10] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically finding semantic differences in binary programs. ICICS, 2008.

[11] Rigel Gjomemo, Phu H Phung, Edmund Ballou, Kedar S Namjoshi, VN Venkatakrishnan, and Lenore Zuck. Leveraging static analysis tools for improving usability of memory error sanitization compilers. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 323–334, 2016.

[12] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[13] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient N-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 339–353, 2015.

[14] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 249–263, 2020.

[15] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 270–278, 1995.

[16] Anil Kurmus and Robby Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *ACM Conference on Computer & Communications Security (CCS)*, 2014.

[17] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–, 2004.

[18] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. PartiSan: fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 403–422, 2018.

[19] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *International Conference on Software Engineering Companion (ICSE-C)*, pages 501–510, 2016.

[20] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. Automatic grading of programming assignments: an approach based on formal semantics. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 126–137, 2019.

[21] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[22] Dongliang Mu. CVE list. https://github.com/VulnReproduction/VulnReproduction.github.io, 2019.

[23] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, 2018.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 245–258, 2009.

[25] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 333–344, 1998.

[26] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[27] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[28] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. Bigfoot: Static check placement for dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 141–156, 2017.

[29] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[30] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and thanks for all the bugs: Finding errors in C programs by abstracting from the native execution model. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 377–391, 2018.

[31] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, 2010.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, 2012.

[33] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. 2019.

[34] Cloyce D Spradling. SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130–134, 2007.

[35] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.

[36] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *International Conference on Compiler Construction (CC)*, pages 265–266, 2016.

[37] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[38] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE TSE*, 40(2):107–122, 2014.

[39] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 132–143, 1977.

[40] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy*, pages 866–879, 2015.

[41] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, 2013.

[42] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *International Symposium on Principles and Practice of Programming in Java (PPPJ)*, pages 125–133, 2007.

[43] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.

[44] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 70–82, 2000.

[45] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426, 2017.

[46] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. pages 175–186, 2013.

Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads

John Thorpe[†♣] Yifan Qiao[†♣] Jonathan Eyolfson[†] Shen Teng[†] Guanzhou Hu[†‡] Zhihao Jia[§]
Jinliang Wei[*] Keval Vora[♭] Ravi Netravali[♯] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *University of Wisconsin*[‡] *CMU*[§] *Google Brain*[*] *Simon Fraser*[♭] *Princeton University*[♯]

Abstract

A graph neural network (GNN) enables deep learning on structured graph data. There are two major GNN training obstacles: 1) it relies on high-end servers with many GPUs which are expensive to purchase and maintain, and 2) limited memory on GPUs cannot scale to today's billion-edge graphs. This paper presents Dorylus: a distributed system for training GNNs. Uniquely, Dorylus can take advantage of *serverless computing* to increase scalability at a low cost.

The key insight guiding our design is *computation separation*. Computation separation makes it possible to construct a *deep, bounded-asynchronous* pipeline where graph and tensor parallel tasks can fully overlap, effectively hiding the network latency incurred by Lambdas. With the help of thousands of Lambda threads, Dorylus scales GNN training to billion-edge graphs. Currently, for large graphs, CPU servers offer the best performance per dollar over GPU servers. Just using Lambdas on top of Dorylus offers up to 2.75× more performance-per-dollar than CPU-only servers. Concretely, Dorylus is 1.22× faster and 4.83× cheaper than GPU servers for massive sparse graphs. Dorylus is up to 3.8× faster and 10.7× cheaper compared to existing sampling-based systems.

1 Introduction

Graph Neural Networks (GNN) [40, 55, 71, 50, 53, 35] are a family of NNs designed for deep learning on graph structured data [103, 92]. The most well-known model in this family is the graph convolutional network (GCN) [40], which uses the connectivity structure of the graph as the filter to perform neighborhood mixing. Other models include graph recursive network (GRN) [51, 67], graph attention network (GAT) [6, 78, 100], and graph transformer network (GTN) [96]. Due to the prevalence of graph datasets, GNNs have gained increasing popularity across diverse domains such as drug discovery [85], chemistry [19], program analysis [2, 5], and recommendation systems [91, 95]. In fact, GNN is one of the most popular topics in recent AI/ML conferences [32, 45].

♣ Contributed equally.

GPUs are the de facto platform to train a GNN due to their ability to provide highly-parallel computations. While GPUs offer great efficiency for training, they (and their host machines) are expensive to use. To train a (small) million-edge graph, recent works such as NeuGraph [55] and Roc [34] need at least four such machines. A public cloud offers flexible pricing options, but cloud GPU instances still incur a non-trivial cost — the lowest-configured p3 instance type on AWS has a price of $3.06/h; training realistic models requires dozens/hundreds of such machines to work 24/7. While cost is not a concern for big tech firms, it can place a heavy financial burden on small businesses and organizations.

In addition to being expensive, GPUs have limited memory, hindering *scalability*. For context, real-world graphs are routinely *billion-edge* scale [69] and continue to grow [95]. NeuGraph and Roc enable coordinated use of multiple GPUs to improve scalability (at higher costs), but they remain unable to handle the billion-edge graphs that are commonplace today. Two main approaches exist for reducing the costs and improving the scalability of GNN training, but they each introduce new drawbacks:

- CPUs face far looser memory restrictions than GPUs, and operate at significantly lower costs. However, CPUs are unable to provide the parallelism in computations that GPUs can, and thus deliver far inferior *efficiency* (or speed).
- Graph sampling techniques select certain vertices and sample their neighbors when gathering data [25, 95]. Sampling techniques improve scalability by considering less graph data, and it is a generic technique that can be used on either GPU or CPU platforms. However, our experiments (§7.5) and prior work [34] highlight two limitations with graph sampling: (1) sampling must be done repeatedly per epoch, incurring time overheads and (2) sampling typically reduces *accuracy* of the trained GNNs. Furthermore, although sampling-based training converges often in practice, there is no convergence guarantee for trivial sampling methods [9].

Affordable, Scalable, and Accurate GNN Training. This

paper devises a *low-cost* training framework for GNNs on *billion-edge graphs*. Our goal is to simultaneously deliver high efficiency (*e.g.*, close to GPUs) and high accuracy (*e.g.*, higher than sampling). Scaling to billion-edge graphs is crucial for applicability to real-world use cases. Ensuring low costs and practical performance improves the accessibility for small organizations and domain experts to make the most out of their rich graph data.

To achieve these goals, we turn to the *serverless computing* paradigm, which has gained increasing traction [20, 43, 37] in recent years through platforms such as AWS Lambda, Google Cloud Functions, or Azure Functions. Serverless computing provides large numbers of parallel "cloud function" threads, or *Lambdas*, at an extremely low price (*i.e.*, $0.20 for launching one million threads on AWS [3]). Furthermore, Lambda presents a pay-only-for-what-you-use model, which is much more appealing than dedicated servers for applications that need only massive parallelism.

Although it appears that serverless threads could be used to complement CPU servers without significantly increasing costs, they were built to execute light asynchronous tasks, presenting two challenges for NN training:

- *Limited compute resources* (*e.g.*, 2 weak vCPUs)
- *Restricted network resources* (*e.g.*, 200 Mbps between Lambda servers and standard EC2 servers [42])

A neural network makes heavy use of (linear algebra based) tensor kernels. A Lambda[1] thread is often too weak to execute a tensor kernel on large data; breaking the data to tiny minibatches mitigates the compute problem at the cost of higher data-transfer overheads. Consequently, using Lambdas naïvely for training an NN could result in significant slowdowns (*e.g.*, 21× slowdowns for training of multi-layer perceptron NNs [29], even compared to CPUs).

Dorylus. To overcome these weaknesses, we developed Dorylus[2], a distributed system that uses cheap CPU servers and serverless threads to achieve the aforementioned goals for GNN training. Dorylus leverages GNN's special computation model to overcome the two challenges associated with the use of Lambdas. Details are elaborated below:

The *first* challenge is *how to make computation fit into Lambda's weak compute profile?* We observed: *not all operations in GNN training need Lambda's parallelism.* GNN training comprises of two classes of tasks [55] – neighbor propagations (*e.g.*, `Gather` and `Scatter`) over the input graph and per-vertex/edge NN operations (such as `Apply`) over the tensor data (*e.g.*, features and parameters). Training a GNN over a large graph is dominated by *graph computation* (see §7.6), not *tensor computation* that exhibits strong SIMD behaviors and benefits the most from massive parallelism.

Based on this observation, we divide a training pipeline into a set of *fine-grained tasks* (Figure 3, §4) based on the type of data they process. Tasks that operate over the graph structure belong to a *graph-parallel path*, executed by CPU instances, while those that process tensor data are in a *tensor-parallel path*, executed by Lambdas. Since the graph structure is taken out of tensors (*i.e.*, it is no longer represented as a matrix), the amount of tensor data and computation can be significantly reduced, providing an opportunity for each tensor-parallel task to run a *lightweight linear algebra operation* on *a data chunk of a small size* — a granularity that a Lambda is capable of executing quickly.

Note that Lambdas are a perfect fit to GNNs' tensor computations. While one could also employ regular CPU instances for compute, using such instances would incur a much higher monetary cost to provide the same level of burst parallelism (*e.g.*, **2.2**× in our experiments) since users not only pay for the compute but also other unneeded resources (*e.g.*, storage).

The *second* challenge is *how to minimize the negative impact of Lambda's network latency?* Our experiments show that Lambdas can spend one-third of their time on communication. To not let communication bottleneck training, Dorylus employs a novel parallel computation model, referred to as *bounded pipeline asynchronous computation* (BPAC). BPAC makes full use of *pipelining* where different fine-grained tasks overlap with each other, *e.g.*, when graph-parallel tasks process graph data on CPUs, tensor-parallel tasks process tensor data, simultaneously, with Lambdas. Although pipelining has been used in prior work [34, 63], in the setting of GNN training, pipelining would be impossible without fine-grained tasks, which are, in turn, enabled by computation separation.

To further reduce the wait time between tasks, BPAC incorporates *asynchrony* into the pipeline so that a fast task does not have to wait until a slow task finishes even if data dependencies exist between them. Although asynchronous processing has been widely used in the past, Dorylus faces a unique technical difficulty that no other systems have dealt with: as Dorylus has two computation paths, where exactly should asynchrony be introduced?

Dorylus uses asynchrony in a novel way at two distinct locations where staleness can be tolerated: *parameter updates* (in the tensor-parallel path) and *data gathering from neighbor vertices* (in the graph-parallel path). To not let asynchrony slow down the convergence, Dorylus *bounds* the degree of asynchrony at each location using different approaches (§5): *weight stashing* [63] at parameter updates and *bounded staleness* at data gathering. We have formally proved the convergence of our asynchronous model in §5.

Results. We have implemented two popular GNNs – GCN and GAT – on Dorylus and trained them over four real-world graphs: `Friendster` (3.6B edges), `Reddit-full` (1.3B), `Amazon` (313.9M), and `Reddit-small` (114.8M). With the help of 32 graph servers and thousands of Lambda threads, Dorylus was able to train a GCN, *for the first time without sampling*, over billion-edge graphs such as `Friendster`.

[1] We use "Lambda" in this paper due to our AWS-based implementation while our idea is generally applicable to all types of serverless threads.

[2] Dorylus is a genus of army ants that form large marching columns.

To enable direct comparisons among different platforms, we built new GPU- and CPU-based training backends based on Dorylus' distributed architecture (with computation separation). Across our graphs, Dorylus's performance is **2.05**× and **1.83**× higher than that of GPU-only and CPU-only servers *under the same monetary budget*. Sampling is surprisingly slow — to reach the same accuracy target, it is **2.62**× slower than Dorylus due to its slow accuracy climbing. In terms of accuracy, Dorylus can train a model with an accuracy **1.05**× higher than sampling-based techniques.

Key Takeaway. Prior work has demonstrated that Lambdas can only achieve suboptimal performance for DNN training due to the limited compute resources on a Lambda and the extra overheads to transfer model parameters/gradients between Lambdas. Through computation separation, Dorylus makes it possible, *for the first time*, for Lambdas to provide a scalable, efficient, and low-cost distributed computing scheme for GNN training.

Dorylus is useful in two scenarios. First, for small organizations that have tight cost constraints, Dorylus provides an affordable solution by exploiting Lambdas at an extremely low price. Second, for those who need to train GNNs on very large graphs, Dorylus provides a scalable solution that supports fast and accurate GNN training on billion-edge graphs.

2 Background

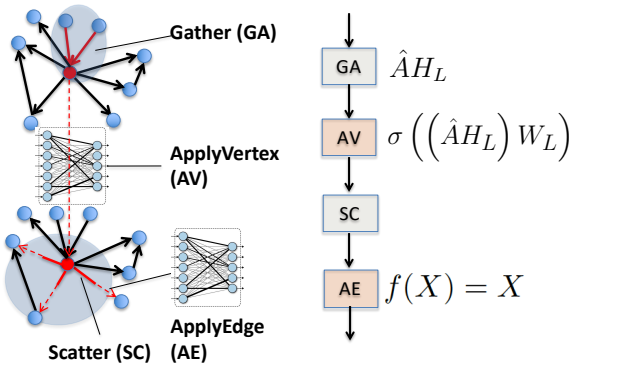
A GNN takes graph-structured data as input, where each vertex is associated with a feature vector, and outputs a feature vector for each individual vertex or the whole graph. The output feature vectors can then be used by various downstream tasks, such as, graph or vertex classification. By combining the feature vectors and the graph structure, GNNs are able to learn the patterns and relationships among the data, rather than relying solely on the features of a single data point.

GNN training combines graph propagation (*e.g.*, `Gather` and `Scatter`) and NN computations. Prior work [17, 89] discovered that GNN development can be made much easier with a programming model that provides a *graph-parallel* interface, which allows programmers to develop the NN with familiar graph operations. A typical example is the deep graph library (DGL) [17], which unifies a variety of GNN models with a common GAS-like interface.

Forward Pass. To illustrate, consider graph convolutional network (GCN) as an example. GCN is the simplest and yet most popular model in the GNN family, with the following forward propagation rule for the L-th layer [40]:

$$(R1) \qquad H_{L+1} = \sigma(\hat{A}H_L W_L)$$

A is the adjacency matrix of the input graph, and $\tilde{A} = A + I_N$ is the adjacency matrix with self-loops constructed by adding A with I_N, the identity matrix. $\tilde{D}$ is a diagonal matrix such that $\tilde{D}_{ii} = \Sigma_j \tilde{A}_{ij}$. With $\tilde{D}$, we can construct a *normalized adjacency matrix*, represented by $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. W_L is a layer-specific trainable weight matrix. $\sigma(.)$ denotes a non-linear activation function, such as



(a) Computation model (b) Annotated dataflow graph

Figure 1: A graphical illustration of GCN's computation model and dataflow graph in each forward layer. In (a), edges in red represent those along which information is being propagated; solid edges represent standard `Gather`/`Scatter` operations while dashed edges represent NN operations. (b) shows a mapping between the SAGA-NN programming model and the rule R1.

$ReLU$. H_L is the *activations matrix* of the L-th layer; $H_0 = X$ is the input feature matrix for all vertices.

Mapping R1 to the vertex-centric computation model is familiar to the systems community [55] — each forward layer has four components: `Gather` (GA), `ApplyVertex` (AV), `Scatter` (SC), and `ApplyEdge` (AE), as shown in Figure 1(a). One can think of layer L's activations matrix H_L as a group of *activations vectors*, each associated with a vertex (as analogous to *vertex value* in the graph system's terminology). The goal of each forward layer is to compute a new activations vector for each vertex based on the vertex's previous activations vector (which, initially, is its feature vector) and the information received from its in-neighbors. Different from traditional graph processing, the computation of the new activations matrix H_{L+1} is based on computationally intensive NN operations rather than a numerical function.

Figure 1(b) illustrates how these vertex-centric graph operations correspond to various components in R1. First, GA retrieves a vector from each in-edge of a vertex and aggregates these vectors into a new vector v. In essence, applying GA on all vertices can be implemented as a matrix multiplication $\hat{A}H_L$, where $\hat{A}$ is the normalized adjacency matrix and H_L is the input activations matrix. Second, $(\hat{A}H_L)$ is fed to AV, which performs neural network operations to obtain a new activations matrix H_{L+1}. For GCN, AV multiplies $(\hat{A}H_L)$ with a trainable weight matrix W_L and applies a non-linear activation function σ. Third, the output of AV goes to SC, which propagates the new activations vector of each vertex along all out-edges of the vertex. Finally, the new activations vector of each vertex goes into an edge-level NN architecture to compute an activations vector for each edge. For GCN, the edge-level NN is not needed, and hence, AE is an identity

function. We leave AE in the figure for generality as it is needed by other GNN models.

The output of AE is fed to GA in the next layer. Repeating this process k times (*i.e.*, k layers) allows the vertex to consider features of vertices k hops away. Other GNNs such as GGNNs and GATs have similar computation models, but each varies the method used for aggregation and the NN.

Backward Pass. A GNN's backward pass computes the gradients for all trainable weights in the vertex- and edge-level NN architectures (*i.e.*, AV and AE). The backward pass is performed following the chain rule of back propagation. For example, the following rule specifies how to compute the gradients in the first layer for a 2-layer GCN:

$$(R2) \quad \nabla_{W_0}\mathcal{L} = \left(\hat{A}X\right)^T \left[\sigma'\left(in_1\right) \odot \hat{A}^T(Z-Y)W_1^T\right]$$

Here Z is the output of the GCN, Y is the label matrix (*i.e.*, ground truth), X is the input features matrix, W_i is the weight matrix for layer i, and $in_1 = \hat{A}XW_0$. $\hat{A}^T$ and W_i^T are the transpose of $\hat{A}$ and W_i, respectively.

A training *epoch* consists of a forward and a backward pass, followed by *weights update*, which uses the gradients computed in the backward pass to update the trainable weights in the vertex- and edge-level NN architectures in a GNN. The training process runs epochs repeatedly until reaching acceptable accuracy.

3 Design Overview

This section provides an overview of the Dorylus architecture. The next three sections discuss technical details including how to split training into fine-grained tasks and connect them in a deep pipeline (§4), and how Dorylus bounds the degree of asynchrony (§5), manages and autotunes Lambdas (§6).

Figure 2 depicts Dorylus's architecture, which is comprised of three major components: EC2 graph servers, Lambda threads for tensor computation, and EC2 parameter servers. An input graph is first partitioned using an edge-cut algorithm [104] that takes care of load balancing across partitions. Each partition is hosted by a graph server (GS).

GSes communicate with each other to execute graph computations by sending/receiving data along cross-partition edges. GSes also communicate with Lambda threads to execute tensor computations. Graph computation is done in a conventional way, breaking a vertex program into vertex-parallel (*e.g.*, `Gather`) and edge-parallel stages (*e.g.*, `Scatter`).

Each vertex carries a vector of float values and each edge carries a value of a user-defined type specific to the model. For example, for a GCN, edges do not carry values and `ApplyEdge` is an identity function; for a GGNN, each edge has an integer-represented type, with different weights for different edge types. After partitioning, each GS hosts a graph partition where vertex data are represented as a two-dimension array and edge data are represented as a single array. Edges are stored in the *compressed sparse rows* (CSR) format; inverse edges are also maintained for the backpropagation.

Each GS maintains a *ghost buffer*, storing data that are scattered in from remote servers. Communication between GSes is needed only during `Scatter` in both (1) forward pass where activation values are propagated along cross-partition edges and (2) backward pass where gradients are propagated along the same edges in the reverse direction.

Tensor operations such as AV and AE, performed by Lambdas, interleave with graph operations. Once a graph operation finishes, it passes data to a Lambda thread, which employs a high-performance linear algebra kernel for tensor computation. Both the forward and backward passes use Lambdas, which communicate frequently with parameter servers (PS) — the forward-pass Lambdas retrieve weights from PSes to compute layer outputs, while the backward-pass Lambdas compute updated weights.

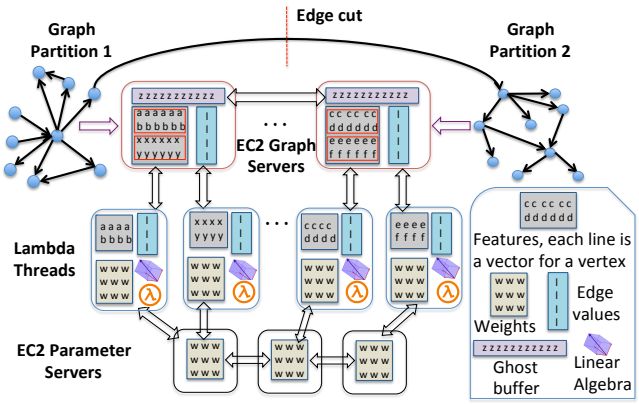


Figure 2: Dorylus's architecture.

4 Tasks and Pipelining

Fine-grained Tasks. As discussed in §1, the first challenge in using Lambdas for training is to decompose the process into a set of fine-grained tasks that can (1) overlap with each other and (2) be processed by Lambdas' weak compute. In Dorylus, task decomposition is done based on both *data type* and *computation type*. In general, computations that involve the adjacency matrix of the input graph (*i.e.*, any computation that multiplies any form of the adjacency matrix A with other matrices) are formulated as graph operations performed on GSes, while computations that involve only tensor data can benefit the most from massive parallelism and hence run in Lambdas. Next, we discuss specific tasks over each training epoch, which consists of a *forward pass* that computes the output using current weights, followed by a *backward pass* that uses a loss function to compute weight updates.

A forward pass can be naturally divided into four tasks, as shown in Figure 1(a). `Gather` (GA) and `Scatter` (SC) perform computation over the graph structure; they are thus graph-parallel tasks for execution on GSes. `ApplyVertex`

(AV) and `ApplyEdge` (AE) multiply matrices involving only features and weights and apply activation functions such as $ReLU$. Hence, they are executed by Lambdas.

For AV, Lambda threads retrieve vertex data (H_L in §2) from GSes and weight data (W_L) from PSes, compute their product, apply $ReLU$, and send the result back to GSes as the input for `Scatter`. When AV returns, SC sends data, along cross-partition edges, to the machines that host their destination vertices.

AE immediately follows SC. To execute AE on an edge, each Lambda thread retrieves (1) vertex data from the source and destination vertices of the edge (*i.e.*, activations vectors), and (2) edge data (such as edge weights) from GSes. It computes a per-edge update by performing model-specific tensor operations. These updates are streamed back to GSes and become the inputs of the next layer's GA task.

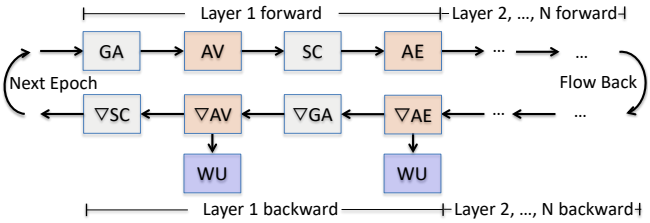


Figure 3: Dorylus's forward and backward dataflow with nine tasks: `Gather` (GA) and `Scatter` (SC) and their corresponding backward tasks ∇GA and ∇SC; `ApplyVertex` (AV), `ApplyEdge` (AE), and their backward tasks ∇AV and ∇AE; the weight update task `WeightUpdate` (WU).

A backward pass involves GSes, Lambdas, and PSes that *coordinate* to run a graph-augmented SGD algorithm, as specified by R2 in §2. For each task in the forward pass, there is a corresponding backward task that either propagates information in the *reverse direction of edges* on the graph or computes the gradients of its trainable weights with respect to a given loss function. Additionally, a backward pass includes `WeightUpdate` (WU), which aggregates the gradients across PSes. Figure 3 shows their dataflow. ∇GA and ∇SC are the same as GA and SC except that they propagate information in the reverse direction. ∇AE and ∇AV are the backward tasks for AE and AV, respectively. AE and AV apply weights to compute the output of the edge and vertex NN. Conversely, ∇AE and ∇AV compute weight updates for the NNs, which are the inputs to WU.

∇AE and ∇AV perform tensor-only computation and are executed by Lambdas. Similar to the forward pass, GA and SC in the backward pass are executed on GSes. WU performs weights updates and is conducted by PSes.

Pipelining. In the beginning, vertex and weight data take their initial values (*i.e.*, H_0 and W_0), which will change as the training progresses. GSes kick off training by running parallel graph tasks. To establish a full pipeline, Dorylus divides vertices in each partition into intervals (*i.e.*, minibatches). For each interval, the amount of tensor computation (done by a Lambda) depends on both the numbers of vertices (*i.e.*, AV) and edges (*i.e.*, AE) in the interval, while the amount of graph computation (on a GS) depends primarily on the number of edges (*i.e.*, GA, and SC). To balance work across intervals, our division uses a simple algorithm to ensure that different intervals have the same numbers of vertices and vertices in each interval have similar numbers of inter-interval edges. These edges incur cross-minibatch dependencies that our asynchronous pipeline needs to handle (see §5).

Each interval is processed by a task. When the pipeline is saturated, different tasks will be executed on distinct *intervals* of vertices. Each GS maintains a *task queue* and enqueues a task once it is ready to execute (*i.e.*, its input is available). To fully utilize CPU resources, the GS uses a thread pool where the number of threads equals the number of vCPUs. When the pool has an available thread, the thread retrieves a task from the task queue and executes it. The output of a GS task is fed to a Lambda for tensor computation.

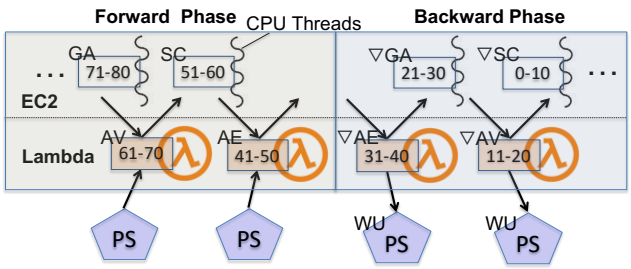


Figure 4: A Dorylus pipeline for an epoch: the number range (*e.g.*, 71-80) in each box represents a particular vertex interval (*i.e.*, minibatch); different intervals are at different locations of the pipeline and processed by different processing units: GS, Lambda, or PS.

Figure 4 shows a typical training pipeline under Dorylus. Initially, Dorylus enqueues a set of GA tasks, each processing a vertex interval. Since the number of threads on each GS is often much smaller than the number of tasks, some tasks finish earlier than others and their results are pushed immediately to Lambda threads for AV. Once they are done, their outputs are sent back to the GS for `Scatter`. During a backward phase, both ∇AE and ∇AV compute gradients and send them to PSes for weight updates.

Through effective pipelining, Dorylus overlaps the graph-parallel and tensor-parallel computations so as to hide Lambdas' communication latency. Note that although pipelining is not a new idea, enabling pipelining in GNN training requires fine-grained tasks and the insight of computation separation, which are our unique contributions.

5 Bounded Asynchrony

To unleash the full power of pipelining, Dorylus performs a unique form of *bounded asynchronous* training so that work-

ers do not need to wait for updates to proceed in most cases. This is paramount for the pipeline's performance especially because Lambdas run in an extremely dynamic environment and stragglers almost always exist. On the other hand, a great deal of evidence [13, 63, 99] shows that asynchrony slows down convergence — fast-progressing minibatches may use out-of-date weights, prolonging the training time.

Bounded staleness [15, 65] is an effective technique for mitigating the convergence problem by employing lightweight synchronization. However, Dorylus faces a unique challenge that does not exist in any existing system, that is, there are *two synchronization points* in a Dorylus pipeline: (1) weight synchronization at each WU task and (2) synchronization of (vertex) activations data from neighbors at each GA.

5.1 Bounded Asynchrony at Weight Updates

To bound the degree of asynchrony for weight updates, we use *weight stashing* proposed in PipeDream [63]. A major reason for slow convergence is that, under full asynchrony, different vertex intervals are at their own training pace; some intervals may use a particular version v_0 of weights during a forward pass to compute gradients while applying these gradients on another version v_1 of weights on their way back in the backward pass. In this case, the weights on which gradients are computed are not those on which they are applied, leading to *statistical inefficiency*. Weight stashing is a simple technique that allows any interval to use the latest version of weights available in a forward pass and stashes (*i.e.*, caches) this version for use during the corresponding backward pass.

Although weight stashing is not new, applying it in Dorylus poses unique challenges in the PS design. Weight stashing occurs at PSes, which host weight matrices and perform updates. To balance loads and bandwidth usage, Dorylus supports multiple PSes and always directs a Lambda to a PS that has the lightest load. Since Lambdas can be in different stages of an epoch (*e.g.*, the forward and backward passes, and different layers), Dorylus lets each PS host a replication of weight matrices of *all layers*, making load balancing much easier to do since any Lambda can use any PS in any stage. Note that this design is different from that of traditional PSes [49], each of which hosts parameters for a layer. Since a GNN often has very few layers, replicating all weights would not take much memory and is thus feasible to do at each PS. Clearly, this approach does *not* work for regular DNNs with many layers.

However, weight stashing significantly complicates this design. A vertex interval can be processed by different Lambdas when it flows to different tasks — *e.g.*, its AV and AE are executed by different Labmdas, which can retrieve weights from different PSes. Hence, if we allow any Lambda to use any PS, each PS has to maintain not only the latest weight matrices but also their stashed versions for *all* intervals in the graph; this is practically impossible due to its prohibitively high memory requirement.

To overcome this challenge, we do *not* replicate all weight stashes across PSes. Instead, each PS still contains a replication of all the latest weights but weight stashes only for a *subset* of vertex intervals. For each interval in a given epoch, the interval's weight stashes are only maintained on the first PS it interacts with in the epoch. In particular, once a Lambda is launched for an AV task, which is the first task that uses weights in the pipeline, its launching GS picks the PS with the lightest load and notifies the Lambda of its address. Furthermore, the GS remembers this choice for the interval — when this interval flows to subsequent tensor tasks (*i.e.*, AE, ∇AV, ∇AE, and WU), the GS assigns the same PS to their executing Lambdas because the stashed version for this interval only exists on that particular PS in this epoch.

PSes periodically broadcast their latest weight matrices.

5.2 Bounded Asynchrony at `Gather`

Asynchronous `Gather` allows vertex intervals to progress independently using stale vertex values (*i.e.*, activations vectors) from their neighbors without waiting for their updates. Although asynchrony has been used in a number of graph systems [82, 15], these systems perform iterative processing with the assumption that with more iterations they will eventually reach convergence. Different from these systems, the number of layers in a GNN is determined statically and an n-layer GNN aims to propagate the impact of a vertex's n-hop neighborhood to the vertex. Since the number of layers cannot change during training, an important question that needs be answered is: can asynchrony change the semantics of the GNN? This boils down to two sub-questions: (1) Can vertices *eventually* receive the effect of their n-hop neighborhood? (2) Is it possible for any vertex to receive the effect of its m-hop neighbor where $m > n$ after many epochs? We provide informal correctness/convergence arguments in this subsection and turn to a formal approach in §5.3.

The answer to the first question is *yes*. This is because the GNN computation is driven by the accuracy of the computed weights, which is, in turn, based on the effects of n-hop neighborhoods. To illustrate, consider a simple 2-layer GNN and a vertex v that moves faster in the pipeline than all its neighbors. Assume that by the time v enters the GA of the second layer, none of its neighbors have finished their first-layer SC yet. In this case, the GA task of v uses stale values from its neighbors (*i.e.*, the same as what were used in the previous epoch). This would clearly generate large errors at the end of the epoch. However, in subsequent epochs, the slow-moving neighbors update their values, which are gradually propagated to v. Hence, the vertex eventually receives the effects of its n-hop neighborhood (collectively) across different epochs depending on its neighbors' progress. After each vertex observes the required values from the n-hop neighborhood, the target accuracy is reached.

The answer to the second question is *no*. This is because the number of layers determines the *farthest distance* the impact of a vertex can travel despite that training may execute

many epochs. When a vertex interval finishes an epoch, it comes back to the initial state where their values are *reset* to their initial feature vectors (*i.e.*, the accumulative effect is cleared). Hence, even though a vertex v progresses asynchronously relative to its neighbors, the neighbors' activation vectors are scattered out in the previous SC and carry the effects of their at most $\{n-1\}$-hop neighbors (after which the next GA will cycle back to effects of 1-hop neighbors), which, for vertex v, belong strictly in its n-hop neighborhood. This means, it is impossible for any vertex to receive the impact of any other vertex that is more than n-hops away.

We use *bounded staleness* at `Gather` — a fast-moving vertex interval is allowed to be at most S epochs away from the slowest-moving interval. This means vertices in a given epoch are allowed to use stale vectors from their neighbors only if these vectors are within S epochs away from the current epoch. Bounded staleness allows fast-moving intervals to make quick progress when recent updates are available (for efficiency), but makes them wait when updates are too stale (to avoid launching Lambdas for useless computation).

5.3 Convergence Guarantee

Asynchronous weight updates with bounded staleness has been well studied, and its convergence has been proved by [30]. The convergence of asynchronous `Gather` with bounded staleness S is guaranteed by the following theorem:

Theorem 1. *Suppose that (1) the activation $\sigma(\cdot)$ is ρ-Lipschitz, (2) the gradient of the cost function $\nabla_z f(y,z)$ is ρ-Lipschitz and bounded, (3) the gradients for weight updates $\|g_{AS}(W)\|_\infty$, $\|g(W)\|_\infty$, and $\|\nabla\mathcal{L}(W)\|_\infty$ are all bounded by some constant $G > 0$ for all $\hat{A}$, X, and W, (4) the loss $\mathcal{L}(W)$ is ρ-smooth*[3]*. Then given the local minimizer W^*, there exists a constant $K > 0$, s.t., $\forall N > L \times S$ where L is the number of layers of the GNN model and S is the staleness bound; if we train a GNN with asynchronous `Gather` under a bounded staleness for $R \leq N$ iterations where R is chosen uniformly from $[1, N]$, we will have*

$$\mathbb{E}_R \left\|\nabla\mathcal{L}\left(W_R\right)\right\|_F^2 \leq 2\frac{\mathcal{L}\left(W_1\right) - \mathcal{L}\left(W^*\right) + K + \rho K}{\sqrt{N}},$$

for the updates $W_{i+1} = W_i - \gamma g_{AS}(W_i)$ and the step size $\gamma = min\left\{\frac{1}{\rho}, \frac{1}{\sqrt{N}}\right\}$.

In particular, we have $\lim_{N\to\infty} \mathbb{E}_R \left\|\nabla\mathcal{L}\left(W_R\right)\right\|^2 = 0$, indicating that asynchronous GNN training will eventually converge to a local minimum. The full-blown proof can be found in Dorylus' arXiv version [77]. It mostly follows the convergence proof of the *variance reduction (VR)* algorithm in [9]. However, our proof differs from [9] in two major aspects: (1) Dorylus performs whole-graph training and updates weights only once per layer per epoch, while VR samples the graph and trains on mini-batches and thus it updates weights multiple times per layer per epoch; (2) Dorylus's asynchronous GNN training can use neighbor activations that are up to S-epoch stale, while VR can take only 1-epoch-stale neighbor activations. Since S is always *bounded* in Dorylus, the convergence is guaranteed regardless of the value of S.

Note that compared to a sampling-based approach, our asynchronous computation is guaranteed to converge. On the contrary, although sampling-based training converges often in practice, there is no guarantee for trivial sampling methods [9], not to mention that sampling incurs a per-epoch overhead and reduces accuracy.

6 Lambda Management

Each GS runs a Lambda controller, which launches Lambdas, batches data to be sent to each Lambda, monitors each Lambda's health, and routes its result back to the GS.

Lambda threads are launched by the controller for a task t at the time t's previous task starts executing. For example, Dorylus launches n Lambda threads, preparing them for AV when n GA tasks start to run. Each Lambda runs with OpenBLAS library [93] that is optimized to use AVX instructions for efficient linear algebra operations. Lambdas communicate with GSes and PSes using ZeroMQ [97].

All of our Lambdas are deployed inside the virtual private cloud (VPC) rather than public networks to maximize Lambdas' bandwidth when communicating with EC2 instances. When a Lambda is launched, it is given the addresses of its launching GS and a PS. Once initialized, the Lambda initiates communication with the GS and the PS, pulling vertex, edge and weight data from these servers. Since Lambda threads are used throughout the training process, these Lambdas quickly become "warm" (*i.e.*, the AWS reuses a container that already has our code deployed instead of cold-starting a new container) and efficient. Our controller also times each Lambda execution and relaunches it after timeout.

Lambda Optimizations. One significant challenge to overcome is Lambdas' limited network bandwidth [29, 42]. Although AWS has considerably improved Lambdas' network performance [4], the per-Lambda bandwidth goes down as the number of Lambdas increases. For example, for each GS, when the number of Lambdas it launches reaches 100, the per-Lambda bandwidth drops to ∼200Mbps, which is more than 3× lower than the peak bandwidth we have observed (∼800Mbps). We suspect that this is because many Lambdas created by the same user get scheduled on the same machine and share a network link.

Dorylus provides three optimizations for Lambdas:

The first optimization is task fusion. Since AV of the last layer in a forward pass is connected directly to ∇AV of the last layer in the next backward pass (see Figure 4), we merge them into a single Lambda-based task, reducing invocations

[3] $\mathcal{L}$ is *ρ-Lipschitz smooth* if $\forall W_1, W_2, |\mathcal{L}(W_2) - \mathcal{L}(W_1) - \langle\nabla\mathcal{L}(W_1), W_2 - W_1\rangle| \leq \frac{\rho}{2}\|W_2 - W_1\|_F^2$, where $\langle A, B\rangle = tr(A^T B)$ is the inner product of matrix A and matrix B, and $\|A\|_F$ is the Frobenius norm of matrix A.

of thousands of Lambdas for each epoch and saving a round-trip communication between Lambdas and GSes.

The second optimization is tensor rematerialization [33, 41]. Existing frameworks cache intermediate results during the forward pass as these results can be reused in the backward pass. For GNN training, for instance, $\hat{A}HW$ is such a computation whose result needs to be cached. Here tensor computation is performed by Lambdas while caching has to be done on GSes. Since a Lambda's bandwidth is limited and network communication is a bottleneck, it is more profitable to rematerialize these intermediate tensors by launching more Lambdas rather than retrieving them from GSes.

The third optimization is Lambda-internal streaming. In particular, if a Lambda is created to process a data chunk, we let the Lambda retrieve the first half of the data, with which it proceeds to computation while simultaneously retrieving the second half. This optimization overlaps computation with communication from within each Lambda, leading to reduced Lambda response time.

Autotuning Numbers of Lambdas. Due to inherent dynamism in Lambda executions, it is not feasible to statically determine the number of Lambdas to be used. On the performance side, the effectiveness of Lambdas depends on whether the pipeline can be saturated. In particular, since certain graph tasks (such as SC) rely on results from tensor tasks (such as AV), too few Lambdas would not generate enough task instances for the graph computation G to saturate CPU cores. On the cost side, too many Lambdas *overstaturate* the pipeline — they can generate too many CPU tasks for the GS to handle. The optimal number of Lambdas is also related to the pace of the graph computation, which, in turn, depends on the graph structure (*e.g.*, density) and partitioning that are hard to predict before execution.

To solve the problem, we develop an autotuner that starts the pipeline by using `min`(#`intervals`, 100) as the number of Lambdas where `intervals` represents the number of vertex intervals on each GS. Our autotuner auto-adjusts this number by periodically checking the size of the CPU's task queue — if the size of the queue constantly grows, this indicates that CPU cores have too many tasks to process, and hence we scale down the number of Lambdas; if the queue quickly shrinks, we scale up the number of Lambdas. The goal here is to stabilize the size of the queue so that the number of Lambdas matches the pace of graph tasks.

7 Evaluation

We wrote a total of 11629 SLOC in C++ and CUDA. There are 10877 of the lines of C++ code: 5393 for graph servers, 2840 for Lambda management (and communication), 1353 for parameter servers, and 1291 for common libraries and utilities. There are 752 lines of CUDA code for GPU kernels including common graph operations like GCN and mean-aggregators with cuSPARSE [66]. Our CUDA code includes deep learning operations such as dense layer and activation layer with cuDNN [12]. Dorylus supports common stochastic optimizations including *Xavier initialization* [22], *He initialization* [27], a *vanilla SGD optimizer* [38], and an *Adam optimizer* [39], which help training converge smoothly.

Graph	Size ($\|V\|$, $\|E\|$)	# features	# labels	Avg. degree
Reddit-small [25]	(232.9K, 114.8M)	602	41	492.9
Reddit-large [25]	(1.1M, 1.3B)	301	50	645.4
Amazon [60, 28]	(9.2M, 313.9M)	300	25	35.1
Friendster [48]	(65.6M, 3.6B)	32	50	27.5

Table 1: We use 4 graphs, 2 with billions of edges.

7.1 Experiment Setup

We experimented with four graphs, as shown in Table 1. `Reddit-small` and `Reddit-large` are both generated from the Reddit dataset [68]. `Amazon` is the *largest graph* in RoC's [34] evaluation. We added a larger 1.8 billion (undirected) edge `Friendster` social network graph to our experiments. For GNN training, we turned undirected edges into two directed edges, effectively doubling the number of edges (which is consistent with how edge numbers are reported in prior GNN work [34, 55]). The first three graphs come with features and labels while `Friendster` does not. For scalability evaluation we generated random features and labels for `Friendster`.

We implemented two GNN models on top of Dorylus: graph convolutional network (GCN) [40] and graph attention network (GAT) [96] with 279 and 324 lines of code. GCN is a popular network that has AV but not AE, while GAT is a recently-developed recurrent network with both AV and AE. Their development is straightforward and other GNN models can be easily implemented on Dorylus as well. Each model has 2 layers, consistent with those used in prior work [34, 55].

Value is the major benefit Dorylus brings to training GNNs. We define value as a system's *performance per dollar*, computed as $V = 1/(T \times C)$ where T is the training time and C is the monetary cost. For example: if system A trains a network twice as fast as system B, and yet costs the same to train, we say A has twice the value of B. If one has a time constraint, the most inexpensive option to train a GNN is to pick the system/configuration that meets the time requirement with the best value. In particular, value is important for training since users cannot take the cheapest option if it takes too long to train; neither can they take the fastest option if it is extremely expensive in practice. Throughout the evaluation, we use both value and performance (runtime) as our metrics.

We evaluated several aspects of Dorylus. First, we compared several different instance types to determine the configurations that give us the optimal value for each backend. Second, we compared several synchronous and asynchronous variants of Dorylus. In later subsections, we use our best variant (which is asynchronous with a staleness value of 0) in comparisons with other existing systems. Third, we compared the effects of Lambdas using Dorylus against more traditional CPU- and GPU-only implementations in terms of value, per-

formance, and scalability. Next, we evaluate Dorylus against existing systems. Finally, we break down our performance and costs to illustrate our system's benefits.

Backend	Graph	Instance Type	Relative Value
CPU	Reddit-large	r5.2xlarge (4)	1
		c5n.2xlarge (12)	4.46
	Amazon	r5.xlarge (4)	1
		c5n.2xlarge (8)	2.72
GPU	Amazon	p2.xlarge (8)	1
		p3.2xlarge (8)	4.93

Table 2: Comparison of the values provided by different instance types. r5 and p2 instances provided significantly lower values than the (c5 and p3) instances we chose.

7.2 Instance Selection

To choose the instance types for our evaluation, we ran a set of experiments to determine the types that gave us the best value for each backend. We compared across memory optimized (r5) and compute optimized (c5) instances, as well as the p2 and p3 GPU instances, which have K80 and V100 GPUs, respectively. As r5 offers high memory, we were able to fit the graph in a smaller number of instances, lowering costs in some cases. However, due to the smaller amount of computational resources available, training on the r5 instances typically took nearly $3\times$ as long as computation on c5. Therefore, as shown in Table 2 the average increases in value c5 instances provided relative to r5 instances are 4.46 and 2.72, respectively, for `Reddit-large` and `Amazon`. We therefore selected c5 as our choice for any CPU based computation.

Similarly, for GPU instances, training on `Amazon` with 8 K80s took 1578 seconds and had a total cost of $3.16. Using 8 V100s took 385 seconds and cost $2.62—it improves both costs and performance, resulting in a value increase of $4.93\times$ compared to training on K80 GPUs. As value is the main metric which we use to evaluate our system, we choose the instance type which gives the best value to each different backend to ensure a fair comparison.

Given these results, we selected the following instances to run our evaluation: (1) c5, compute-optimized instances, and (2) c5n, compute and network optimized instances. c5n instances have more memory and faster networking, but their CPUs have slightly lower frequency than those in c5. The base c5 instance has 2 vCPU, 4 GB RAM, and 10 Gbps per-instance network bandwidth costing $0.085/h[4]. The base c5n instance has 2 vCPU, 5.25 GB RAM (33% more), and 25 Gbps per-instance network bandwidth, costing $0.108/h. We used the base p3 instance, p3.2xlarge, with Telsa V100 GPUs. Each p3 base instance has 1 GPU (with 16 GB memory), 8 vCPUs, and 61 GB memory, costing $3.06/h.

Each Lambda is a container with 0.11 vCPUs and 192 MB memory. Lambdas have a static cost of $0.20 per 1 M requests, and a compute cost of $0.01125/h (billed per 100 ms). This billing granularity enables serverless threads to handle short bursts of massive parallelism much better than CPU instances.

[4]These prices are from the Northern Virginia region.

Model	Graph	CPU cluster	GPU cluster
GCN	Reddit-small	c5.2xlarge (2)	p3.2xlarge (2)
	Reddit-large	c5n.2xlarge (12)	p3.2xlarge (12)
	Amazon	c5n.2xlarge (8)	p3.2xlarge (8)
	Friendster	c5n.4xlarge (32)	p3.2xlarge (32)
GAT	Reddit-small	c5.2xlarge (10)	p3.2xlarge (10)
	Amazon	c5n.2xlarge (12)	p3.2xlarge (12)

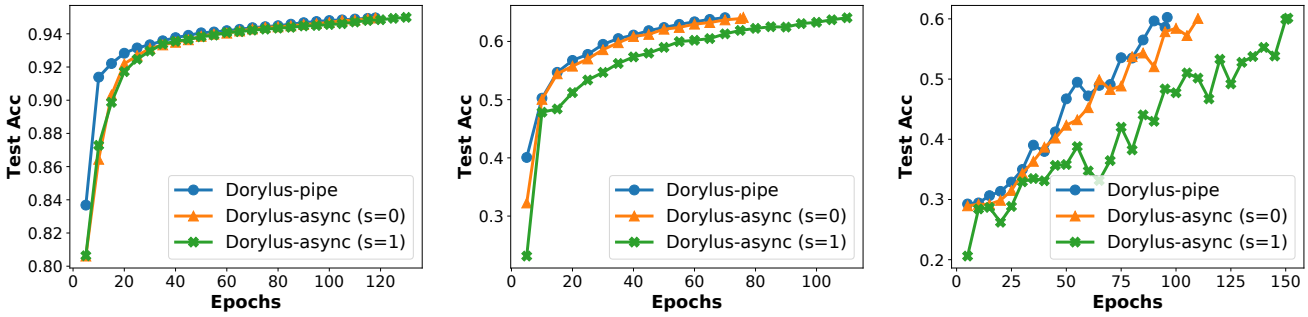
Table 3: We used (mostly) c5n instances for CPU clusters, and equivalent numbers of p3 instances for GPU clusters.

Table 3 shows our CPU and GPU clusters for each pair of model and graph we evaluated. For each graph, we picked the number of servers such that they have just enough memory to hold the graph data and their tensors. For example, `Amazon` needs 8 c5n.2xlarge servers (with 16 GB memory) provide enough memory. For `Friendster` we need 32 c5n.4xlarge instances (with a total of 1344 GB memory). Our goal is to train a model with the minimum amount of resources. Of course, using more servers will lead to better performance and higher costs (discussed in §7.4). For all experiments (except `Reddit-small`), c5n instances offered the best value.

TPU has become an important type of computation accelerator for machine learning. This paper focuses on AWS and its serverless platform, and hence we did not implement Dorylus on TPUs. Although we did not compare directly with TPUs, we note several important features of GNNs that make the limitations of TPUs comparable to GPUs. First, GNNs are unlike conventional DNNs in that they require large amounts of data movement for neighborhood aggregation. As a result, GNN performance is mainly bottlenecked by memory constraints and the resulting communication overheads (*e.g.*, between GPUs or TPUs), *not* computation efficiency [34]. Second, GNN training involves computation on large sparse tensors that incur irregular data accesses, resulting in sub-optimal performance on TPUs which are optimized for dense matrix operations over regularly structured data.

7.3 Asynchrony

We compare three versions of Dorylus: a synchronous version with full intra-layer pipelining (pipe), and two asynchronous versions using $s = 0$ and $s = 1$ as the staleness values over all four graphs. Pipe synchronizes at each `Gather` — a vertex cannot go into the next layer until all its neighbors have their latest values scattered. As a result, all vertex intervals have to be in the same layer in the same epoch. However, inside each layer, pipelining is enabled, and hence different tasks are fully overlapped. Async enables both pipelining and asynchrony (*i.e.*, stashing weights and using stale values at GA). When the staleness value is s = 0, Dorylus allows a vertex to use a stale value from a neighbor as long as the neighbor is in the same epoch (*e.g.*, can be in a previous layer). In other words, Async (s=0) enables fully pipelining across different layers



(a) Reddit-small
R[s=0]:1.00, R[s=1]:1.07, Accuracy:94.96%

(b) Amazon
R[s=0]:1.09, R[s=1]:1.57, Accuracy:64.08%

(c) Reddit-large
R[s=0]:1.14, R[s=1]:1.58, Accuracy:60.07%

Figure 5: Asynchronous progress for GCN: All three versions of Dorylus achieve the final accuracy *i.e.*, **94.96%**, **64.08%**, **60.07%** for the three graphs). Friendster is not included because it does not come with meaningful features and labels.

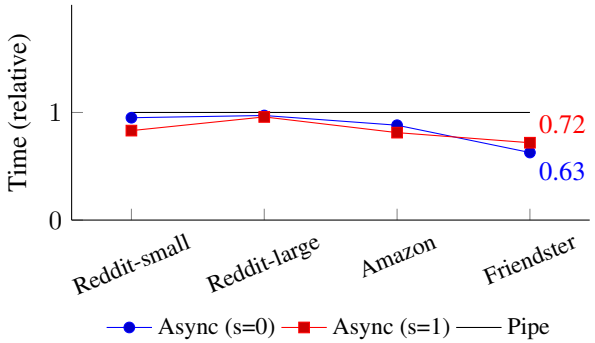


Figure 6: Per-epoch GCN time for async (s=0) and async (s=1) normalized to that of pipe.

in the same epoch, but pipelining tasks in different epochs are not allowed and synchronization is needed every epoch. Similarly, async (s=1) enables a deeper pipeline across two consecutive epochs.

Training Progress. Due to the use of asynchrony, it may take the asynchronous version of Dorylus more epochs to reach the same accuracy as pipe. To enable a fair comparison, we first ran Dorylus-pipe until convergence (*i.e.*, the difference of the model accuracy between consecutive epochs is within 0.001, unless otherwise stated) and then used this accuracy as the target accuracy to run async when collecting training time. However, this approach does not work for Friendster, because it uses randomly generated features/labels and hence accuracy is not a meaningful target. To solve this problem, we computed an average ratio, across the other three graphs, between the numbers of epochs needed for async and pipe, and used this ratio to estimate the training time for Friendster. For example, this ratio is 1.08 for s=0 and 1.41 for s=1. As such, we let async (s=0) run N$\times$1.08 epochs and async (s=1) run N$\times$1.41 epochs when measuring performance for Friendster where N is the number of epochs pipe runs.

Figure 5 reports the GCN training progress for each variant, that is, how many epochs it took for a version to reach the target accuracy. Annotated with each figure are two ratios: R[s=0] and R[s=1], representing the ratio between the number of epochs needed by async (s=0/1) and that by Dorylus-pipe to reach the same target accuracy. On average, async (s=0/1) increases the number of epochs by 8%/41%.

Figure 6 compares the per-epoch running time for each version of Dorylus, normalized to that of pipe. As expected, async has lower per-epoch time; in fact, async (s=0) achieves almost the same reduction ($\sim$15%) in per-epoch time as s=1. This indicates that choosing a large staleness value has little usefulness — it cannot further reduce per-epoch time and yet the number of epochs grows significantly.

To conclude, asynchrony can provide overall performance benefits in general but too large a staleness value leads to slow convergence and poor performance, although the per-epoch time reduces. This explains why async (s=0) outperforms async (s=1) by a large margin. Overall, async (s=0) is **1.234**$\times$ faster than pipe and **1.233**$\times$ than async (s=1). It also provides **1.288**$\times$ and **1.494**$\times$ higher value than pipe and async (s=1) respectively. Thus we choose it as the default Lambda variant in our following experiments unless otherwise specified. From this point on, Dorylus refers to this particular version.

7.4 Effects of Lambdas

We developed two traditional variants of Dorylus to isolate the effects of serverless computing using Lambdas, one using CPU-only servers for computations, and the other using GPU-only servers (both without Lambdas). These variants perform all tensor and graph computations directly on the graph server. They both use Dorylus' (tensor and graph) computation separation for scalability. Note that without computation separation, no existing GPU-based training system has been shown to scale to a billion-edge graph.

Since Lambdas have weak compute that we cannot find in regular EC2 instances, it is not possible for us to translate Lambda resources directly into equivalent EC2 resources, keeping the total amount of compute constant when selecting the number of servers for each variant. To address this con-

cern, we compared the value of different systems in addition to their absolute times and costs.

Model	Graph	Mode	Time (s)	Cost ($)
GCN	Reddit-small	Dorylus	860.6	0.20
		CPU only	1005.4	0.19
		GPU only	162.9	0.28
	Reddit-large	Dorylus (pipe)	1020.1	1.69
		CPU only	1290.5	1.85
		GPU only	324.9	3.31
	Amazon	Dorylus	512.7	0.79
		CPU only	710.2	0.68
		GPU only	385.3	2.62
	Friendster	Dorylus	1133.3	13.8
		CPU only	1990.8	15.3
		GPU only	1490.4	40.5
GAT	Reddit-small	Dorylus	496.3	1.15
		CPU only	1270.4	1.20
		GPU only	130.9	1.11
	Amazon	Dorylus	853.4	2.67
		CPU only	2092.7	3.01
		GPU only	1039.2	10.60

Table 4: We ran Dorylus in 3 different modes: "Dorylus", our best Lambda variant using async(s=0) (except in one case), the "CPU only" variant, and the "GPU only" variant. For each mode we used multiple combinations of models and graphs. For each run we report the total end-to-end running time and the total cost.

We ran GCN and GAT on our graphs (Table 4). We only ran the GAT model on one small and large graph because it was simply too monetarily expensive (even for our system!). GAT has an intensive AE computation, which adds cost. Note that this is *not* a limitation of our system—our system can scale GAT to graphs larger than `Amazon` if cost is not a concern.

Performance and cost by themselves do not properly illustrate the value of Dorylus. For example, training GAT on `Amazon` with Dorylus is both more efficient and cheaper than the CPU- and GPU-only variants. Hence, we report the value results as well. Recall that to compute the value, we take the reciprocal of the total runtime (*i.e.*, the performance or rate of completion) and divide it by the cost. In this case Dorylus with Lambdas provides a 2.75× higher value than CPU-only (*i.e.*, $1/(853.4 \times 2.67)$ compared to $1/(2092.7 \times 3.01)$). Figure 7 shows the value results for all our runs, *normalized to GPU-only servers*.

Dorylus adds value for large, sparse graphs (*i.e.*, `Amazon` and `Friendster`) for both GCN and GAT, compared to CPU- and GPU-only variants. Sparsity of each graph can be seen from the average vertex degree reported in Table 1. As shown, `Amazon` and `Friendster` are much more sparse than `Reddit-small` and `Reddit-large`. For these graphs, the GPU-only variant has the lowest value, even compared to the CPU-only variant. In most cases, the CPU-only variant provides twice as much value (*i.e.*, performance per dollar)

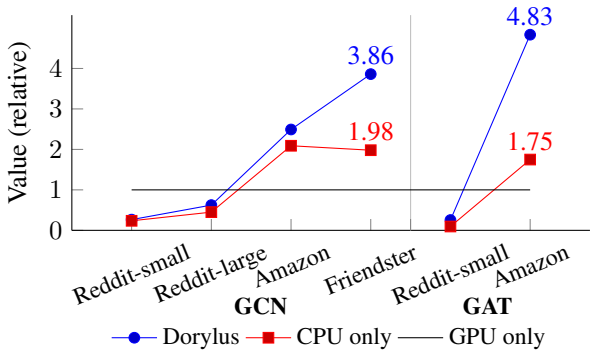


Figure 7: Dorylus, with Lambdas, provides up to 2.75× performance-per-dollar than using the CPU-only variant.

than the GPU-only variant. Dorylus adds another leap in value over the CPU-only variant.

However, for small dense graphs (*i.e.*, `Reddit-small` and `Reddit-large`), both Dorylus and the CPU-only variant have a value lower that that of the GPU-only variant (*i.e.*, below 1 in Figure 7). Dorylus always provides more value than the CPU-only variant. These results suggest that GPUs may be better suited to process small, dense graphs rather than large, sparse graphs.

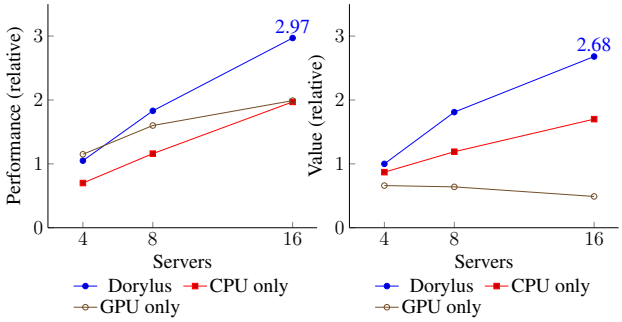


Figure 8: Normalized GCN training performance and value over `Amazon` with varying numbers of graph servers.

Scaling Out. Dorylus can gain even more value by scaling out to more servers, due to the burst parallelism provided by Lambdas and deep pipelining. To understand the impact of the number of servers on performance/costs, we varied the number of GSes when training a GCN over `Amazon`. In particular, we ran Dorylus and the CPU-only variant with 4, 8, and 16 c5n.4xlarge servers, and the GPU-only variant with the same numbers of p3.xlarge servers. Figure 8 reports their performance and values, normalized to those of Dorylus under 4 servers.

In general, Dorylus scales well in terms of both performance and value. Dorylus gains a 2.82× speedup with only 5% more cost when the number of servers increases from 4 to 16, leading to a 2.68× gain in its value. As shown in Figure 8(b), Dorylus's value curve is always above that of the

CPU-only variant. Furthermore, *Dorylus can roughly provide the same value as the CPU-only variant with only half of the number of servers*. For example, Dorylus with 4 servers provides a comparable value to the CPU-only variant with 8 servers; Dorylus with 8 servers provides more value to the CPU-only variant with 16 servers. These results suggest that as more servers are added, the value provided by Dorylus increases, at a rate much higher than the value increase of the CPU-only variant. As such, Dorylus is always a better choice than the CPU-only variant under the same monetary budget.

Other Observations. In addition to the results discussed above, we make three other observations on performance.

Our first observation is that the more sparse the graph, the more useful Dorylus is. For `Amazon` and `Friendster`, Dorylus even outperforms the GPU-only version for two reasons:

First, for all the three variants, the fraction of time on `Scatter` is significantly larger when training over `Friendster` and `Amazon` than `Reddit-small` and `Reddit-large`. This is, at first sight, counter-intuitive because one would naturally expect less efforts on inter-partition communications for sparse graphs than dense graphs. A thorough inspection discovered that the `Scatter` time actually depends on a *combination* of the number of ghost vertices and inter-partition edges. For the two Reddit graphs, they have many inter-partition edges, but very few ghost vertices, because (1) their $|V|$ is small and (2) many inter-partition edges come from/go to the same ghost vertices due to their high vertex degrees.

Second, `Scatter` takes much longer time in GPU clusters. Moving ghost data between GPU memories on different nodes is much slower than data transferring between CPU memories. As a result, the poor performance of the GPU-only variant is due to a combinatorial effect of these two factors: Dorylus scatters significantly more data for `Friendster` and `Amazon`, which amplifies the negative impact of poor scatter performance in a GPU cluster. Note that p3 also offers multi-GPU servers, which may potentially reduce scatter time. We have also experimented with these servers, but we still observed long scatter time due to extensive communication between between servers and GPUs. Reducing such communication costs requires fundamentally different techniques such as those proposed by NeuGraph [55]. We leave the incorporation of such techniques to future work.

Our second observation is that Lambda threads are more effective in boosting performance for GAT than GCN. This is because GAT includes an additional AE task, which performs intensive per-edge tensor computation and thus benefits significantly from a high degree of parallelism.

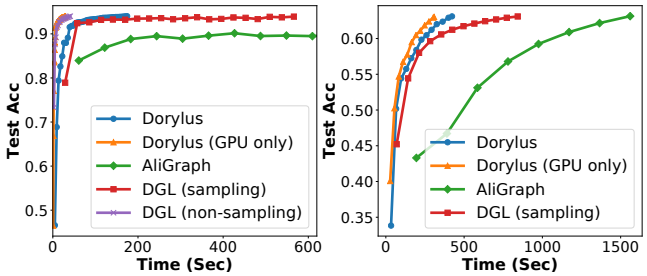
Our third observation is that Dorylus achieves comparable performance with the CPU-only variant that uses twice as many servers. For example, the training time of Dorylus under 4 servers is only 1.1× longer than that of the CPU only variant with 8 servers. Similarly, Dorylus under 8 servers is only 1.05× slower than the CPU only variant with 16 servers. These results demonstrate our efficient use of Lambdas.

7.5 Comparisons with Existing Systems

Our goal was to compare Dorylus with all existing GNN tools. However, NeuGraph [55] and AGL [98] are not publicly available; neither did their authors respond to our requests. Roc [34] is available but we could not run it in our environment due to various CUDA errors; we were not able to resolve these errors after multiple email exchanges with the authors. Roc was not built for scalability because each server needs to load the entire graph into its memory during processing. This is not possible when processing billion-edge graphs. This subsection focuses on the comparison of Dorylus, DGL [17], which is a popular GNN library with support for sampling, as well as AliGraph [94], which is also a sampling-based system that trains GNNs only with CPU servers. All experiments use the cluster configuration specified above for each graph unless otherwise stated.

DGL represents an input graph as a (sparse) matrix; both graph and tensor computations are executed by PyTorch or MXNet as matrix multiplications. We experimented with two versions of DGL, one with sampling and one without. DGL-non-sampling does full-graph training on a single machine. DGL-sampling partitions the graph and distributes partitions to different machines. Each machine performs sampling on its partition and trains a GNN on sampled subgraphs.

AliGraph runs in a distributed setting with a server that stores the graph information. A set of clients query the server to obtain graph samples and use them as minibatches for training. Similar to DGL, AliGraph uses a traditional ML framework as a backend and performs all of its computation as tensor operations.



(a)`Reddit-small` (b) `Amazon`

Figure 9: Accuracy comparisons between Dorylus, Dorylus (GPU only), AliGraph, DGL (sampling), and DGL (non-sampling). DGL (non-sampling) uses a single V100 GPU and could not scale to `Amazon`. Each dot indicates five epochs for Dorylus and DGL (non-sampling), and one epoch for DGL (sampling) and AliGraph.

Accuracy Comparison with Sampling. Figure 9 reports the accuracy-time curve for five configurations: Dorylus, Dorylus (GPU-only), DGL (sampling), DGL (non-sampling),

Graph	System	Time (s)	Cost ($)
Reddit-small	Dorylus	165.77	0.045
	Dorylus (GPU only)	28.06	0.052
	DGL (sampling)	566.33	0.480
	DGL (non-sampling)	33.64	0.028
	AliGraph	–	–
Amazon	Dorylus	415.23	0.654
	Dorylus (GPU only)	308.27	2.096
	DGL (sampling)	842.49	5.728
	DGL (non-sampling)	–	–
	AliGraph	1560.66	1.498

Table 5: Evaluation of end-to-end performance and costs of Dorylus and other GNN training systems. Each time reported is the time to reach the target accuracy.

and AliGraph, over `Reddit-small` and `Amazon`. When run enough epochs to fully converge, Dorylus can reach an accuracy of **95.44%** and **67.01%**, respectively, for the two graphs. DGL (non-sampling) can run only on the `Reddit-small` graph, reaching 94.01% as the highest accuracy. DGL (sampling) is able to scale to both graphs, and its accuracy reaches 93.90% and 65.78%, respectively, for `Reddit-small` and `Amazon`. AliGraph is able to scale to both `Reddit-small` and `Amazon`. On `Reddit-small` it reaches a maximum accuracy of 91.12% and 65.23% on `Amazon`.

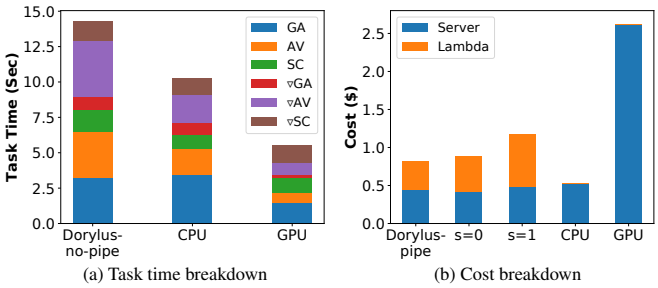
Performance. To enable meaningful performance comparisons and make training finish in a reasonable amount of time, we set 93.90% and 63.00% as our target accuracy for the two graphs. As shown in Figure 9(a), Dorylus (GPU only) has the best performance, followed by DGL (non-sampling). Since `Reddit-small` is a small graph that fits into the memory of a single (V100) GPU, DGL (non-sampling) performs much better than DGL (sampling), which incurs *per-epoch* sampling overheads. To reach the same accuracy (93.90%), Dorylus is 3.25× faster than DGL (sampling), but 5.9× slower than Dorylus (GPU only). AliGraph is unable to reach our target accuracy after many epochs.

For the `Amazon` graph, DGL cannot scale without sampling. As shown in Figure 9(b), to reach the same target accuracy, Dorylus is 1.99× faster than DGL (sampling), and 1.37× slower than Dorylus (GPU only). AliGraph is able to reach the target accuracy for `Amazon`. However, Dorylus is significantly faster. As these results show, graph sampling improves scalability at the cost of increased overheads and reduced accuracy.

The times reported for Dorylus and its GPU-only variant in Table 5 are smaller than those reported in Table 4. This is due to the lower target accuracy we set for these experiments.

Value Comparison. To demonstrate the promise of Dorylus, we compared these systems using the value metric. As expected, given the small size of the `Reddit-small` graph, the GPU-based systems perform quite well. In fact, in this case the normalized value of DGL (non-sampling) is 1.48, providing a higher value than Dorylus (GPU only). However, as mentioned earlier, DGL cannot scale without sampling; hence, this benefit is limited only to small graphs. As we process `Amazon`, the value of Dorylus quickly improves as is consistent with our findings earlier (on large, sparse graphs). With this dataset, Dorylus provides a higher performance-per-dollar rate than *all* the other systems—17.7× the value of DGL (sampling) and 8.6× the value of AliGraph.

7.6 Breakdown of Performance and Costs



(a) Task time breakdown (b) Cost breakdown

Figure 10: Time and cost breakdown for the `Amazon` graph.

Figure 10 shows a breakdown in task time (a) and costs (b) for training a GCN over the `Amazon` graph. In Figure 10(a), to understand the time each task spends, we disabled pipelining and asynchrony in Dorylus, producing a version referred to as no-pipe, in which different tasks never overlap. This makes it possible for us to collect each task's running time. Note that no-pipe represents a version that uses Lambdas naïvely to train a DNN. Without pipelining and overlapping Lambdas with CPU-based tasks, we saw a **1.9**× degradation, making no-pipe lose to both CPU and GPU in training time.

As shown, the tasks GA, AV, and ∇AV take the majority of the time. Another observation is that to execute the tensor computation AV, GPU is the most efficient backend and Lambda is the least efficient one. This is expected — Lambdas have less powerful compute (much less than CPUs in the c5 family) and high communication overheads. Nevertheless, these results also demonstrate that *when CPUs on graph servers are fully saturated with the graph computation, large gains can be obtained by running tensor computation in Lambdas that fully overlap with CPU tasks!*

To compute the cost breakdown in Figure 10(b), we simply calculated the total amounts of time for Lambdas and GSes for each of the five Dorylus variants and used these times to compute the costs of Lambdas and servers. Due to Dorylus' effective use of Lambdas, we were able to run a large number of Lambdas for the forward and backward pass. As such, the cost of Lambdas is about the same as the cost of CPU servers.

8 Related Work

Dorylus is the first system that successfully uses tiny Lambda threads to train a GNN by exploiting various graph-related optimizations. There are three categories of techniques in par-

allelization (§8.1), GNN training (§8.2), and graph systems (§8.3).

8.1 Parallel Computation for Model Training

How to exploit parallelism in model training is a topic that has been extensively studied. There are two major dimensions in how to effectively parallelize the training work: (1) what to partition and (2) how to synchronize between workers.

What to Partition. The most straightforward parallelism model is *data parallelism* [8, 14, 16, 24, 72, 73, 76, 99], where *inputs* are partitioned and processed by individual workers. Each worker learns parameters (weights) from its own portion of inputs and periodically shares its parameters with other workers to obtain a global view. Both share-memory systems [8, 73, 24] and distributed systems [49, 99, 13] have been developed for data-parallel training. Another parallelization strategy is to partition the work, often referred to as *model parallelism* [61] where the operators in a model are partitioned and each worker evaluates and updates only a subset of parameters *w.r.t.* its model partition for all inputs.

A recent line of work develops techniques for *hybrid parallelism* [63, 31, 36, 44]. PipeDream [63] adds pipelining into model parallelism to fully utilize compute without introducing significant stalls. Although Dorylus also uses pipelining, tasks on a Dorylus pipeline are much finer-grained. For example, instead of splitting a model into layers, we construct graph and tensor tasks in such a way that graph tasks can be parallelized on graph servers, while each tensor task is small enough to fit into a Lambda's resource profile. Dorylus uses pipelining to overlap graph and tensor computations specifically to mitigate Lambdas' network latency. FlexFlow [36] automatically splits an iteration along four dimensions.

How Workers Synchronize. When workers work on different portions of inputs (*i.e.*, data parallelism), they need to share their learned parameters with other workers. Parameter updating requires synchronization between workers. For share-memory systems, they often rely on primitives such as `all_reduce` [8] that broadcasts each worker's parameters to all other workers. Distributed systems including Dorylus use parameter servers [49, 99, 13], which periodically communicate with workers for updating parameters. The most commonly-used approach for synchronization is the bulk synchronous parallel (BSP) model, which poses a barrier at the end of each epoch. All workers need to wait for gradients from other workers at the barrier. Wait-free backpropagation [99] is an optimization of the BSP model.

Since synchronous training often introduces computation stalls, *asynchronous* training [8, 16] has been proposed to reduce such stalls — each worker proceeds with the next input minibatch before receiving the gradients from the previous epoch. An asynchronous approach reduces time needed for each epoch at the cost of increased epochs to reach particular target accuracy. This is because allowing workers to use parameters learned in epoch m to perform forward computations in epoch n ($n \neq m$) leads to *statistical inefficiency*. This problem can be mitigated with a hybrid approach such as *bounded staleness* [63, 15, 82, 65].

8.2 GNN Training and Graph Systems

As the GNN family keeps growing [91, 96, 18, 47, 95, 103, 51, 35, 94, 98], developing efficient and scalable GNN training systems becomes popular. GraphSage [25] uses graph sampling, NeuGraph [55] extends GNN training to multiple GPUs, and RoC [34] uses dynamic graph partitioning to achieve efficiency. Other systems that can scale to large graphs are all based on sampling [94, 98].

Programming frameworks such as DGL [17] have been proposed to create a graph-parallel interface (*i.e.*, GAS) for developers to easily mix graph operations with NNs. However, such frameworks still represent the graph as a matrix and push it to an underlying training framework such as TensorFlow for training. We solve this fundamental scalability problem with a ground-up system redesign that separates the graph computation from the tensor computation.

8.3 Graph-Parallel Systems

There exists a body of work on scalable and efficient graph systems of many kinds: single-machine share-memory systems [75, 64, 21, 59, 58], disk-based out-of-core systems [46, 70, 105, 87, 52, 102, 86, 26, 84, 56, 79, 1, 88], and distributed systems [57, 54, 23, 10, 69, 11, 104, 101, 74, 81, 62, 90, 7, 80, 83]. These systems were built on top of a graph-parallel computation model, whether it is vertex-centric or edge-centric. Inspired by these systems, Dorylus formulates operations involving the graph structure as graph-parallel computation and runs it on CPU servers for scalability.

9 Conclusion

Dorylus is a distributed GNN training system that scales to large billion-edge graphs with low-cost cloud resources. We found that CPU servers, in general, offer more performance per dollar than GPU servers for large sparse graphs. Adding Lambdas added $2.75\times$ more performance-per-dollar than CPU only servers, and $4.83\times$ more than GPU only servers. Compared to existing sampling-based systems Dorylus is up to $3.8\times$ faster and $10.7\times$ cheaper. Based on the trends we observed Dorylus can scale to even larger graphs than we evaluated, offering even higher values.

Acknowledgments

We thank the anonymous reviewers for their comments. We are grateful to our shepherd Amar Phanishayee for his feedback. This work is supported by NSF grants CCF-1629126, CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CNS-1901510, CNS-1943621, CHS-1956322, CNS-2007737, CNS-2006437, CNS-2106838, ONR grants N00014-16-1-2913 and N00014-18-1-2037, as well as a Sloan Fellowship.

A Artifact Appendix

A.1 Artifact Summary

Dorylus is a distributed GNN training system that scales to large billion-edge graphs using cheap cloud resources—specifically CPU servers and serverless threads. It launches a set of graph servers which are used for processing graph data and doing operations such as gather and scatter. In addition, parameter servers hold the weights for the model. It can be configured to run with multiple different backends, such as a pure CPU backend and a GPU backend. By separating the graph and tensor components of a graph neural network Dorylus is able to effectively utilize serverless threads by providing a deep asynchronous-parallel pipeline in which tensor and graph operations are overlapped. By doing this Dorylus significantly improves the performance-per-dollar of serverless training over both the CPU and GPU backends.

A.2 Artifact Check-list

- **Hardware: AWS cloud account**
- **Public link:** **https://github.com/uclasystem/dorylus**
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 Dorylus's Codebase

Dorylus contains the following three components:

- The Graph Server which performs graph operations and manages Lambda threads (which can also use CPU and GPU backends)
- The Weight Server which holds the model parameters and sends them to the workers
- The Lambda functions which can be uploaded to AWS to be used during training

A.3.2 Deploying Dorylus

To build Dorylus, the first step is to make sure you have the following dependencies installed on your local machine:

- `awscli`
- `python3-venv`

Make sure to run `aws configure` and set up your credentials to allow you to have access to AWS services. Once these are installed, we need to download the code and setup the environment:

```
git clone
git@github.com:uclasystem/dorylus.git

cd dorylus/
git checkout v1.0 # artifact tag

python3 -m venv venv
source venv/bin/activate
pip install -U pip
pip install -r requirements.txt
```

Set Up the Cluster. We now discuss how to setup the cluster with all different roles. To do this, we use the `ec2man` python module. To start, setup the profile in the following way:

```
$ vim ec2man/profile

default # Profile from ~/.aws/credentials
ubuntu # Cluster username
${HOME}/.ssh/id_rsa # Path to SSH key
us-east-2 # AWS region
```

As mentioned previously, we work with two types of workers which we call 'contexts', specifically graph and weight servers. To add machines to these two contexts, we use one of the following commands:

```
python -m ec2man allocate --ami [AMI]
  --type [ec2 type] --cnt [#servers]
  --sg [security group]
  --ctx [weight|graph]

python -m ec2man add [graph|weight]
  [list of ec2 ids]
```

Run the first command with an AMI ID that presents a fresh install of Ubuntu, ideally with about 36 GB of storage. Alternatively if you have created instances already, say 4 graph servers you can add them to the module using the `add` command with a list of their IDs. Finally, run the command `python -m ec2man setup` to get the data about the instances so they can be managed by the module. To make sure everything is setup correctly, try SSHing into graph server 0 using `python -m ec2man graph 0 ssh`.

Building Dorylus. The next step is to make sure all dependencies are installed to build Dorylus on the cluster machines. To do this, run the following commands:

```
local$ ./gnnman/send-source [--force]
  # '--force' removes existing code

local$ ./gnnman/install-dep
```

This will sync the source code with the nodes on the cluster. Then, it will install all dependencies required to build Dorylus.

If this fails for some reason you may need to ssh into each node, move into the `dorylus/gnnman/helpers` directory, and run:

```
remote$ ./graphserver.install
remote$ ./weightserver.install
```

Parameter Files. There are a number of parameter files relating to things such as the ports used during training. Most of these will be fine as they are and should only be changed if there is a conflict.

Compiling the Code. To build and synchronize the code on all nodes in the cluster run:

```
local$ ./gnnman/setup-cluster
local$ ./gnnman/build-system
  [graph|weight] [cpu|gpu]
```

The first command sets up each node of the cluster to be aware of each other. This is important as we only build the code on node 0 and distribute it to other nodes. The second command runs CMake to build the actual system. Not specifying a context builds for all contexts. Not specifying either `cpu` or `gpu` as the backend builds the serverless version.

Setting up Lambda Functions. To install the Lambda functions, you can SSH into one of the weight or graph servers. Once there, run the following commands:

```
# Install the Lambda dependencies
remote$ ./funcs/manage-funcs.install

# Build and upload the function to the cloud
remote$ cd src/funcs
remote$ ./<function-name>/upload-func
```

A.3.3 Preparing the Data

There are 4 main inputs to Dorylus:

- The graph structure
- Graph partition info
- Input features
- Training labels

Graph Input. To prepare an input graph for Dorylus, the format should be a binary edge list with vertices numbered from 0 to $|V|$ with no breaks using 4 byte values. The file should be named `graph.bsnap`.

Partition Info. Dorylus uses edge-cut partitioning. While we do limit partitioning to edge-cuts, we allow flexibility in how the edge cut is implemented by partitioning at runtime. Provide a text file that lists partition assignments line by line, where each line number corresponds to the vertex ID and the number is the partition to which it is assigned. The file should be called `graph.bsnap.parts`.

Input Features. The input features take the form of a tensor of size $|V| \times d$ where d is the number of input features. The file should be binary and take the format of:

```
[numFeats][v0_feats][v1_feats][v2_feats]...
```

The file should be called `features.bsnap`.

Training Labels. The labels file should be binary and take the form:

```
[numLabels][label0][label1]...
```

This file should be called `labels.bsnap`.

Preparing the NFS Server. On an NFS server setup the dataset in the following format under a directory called `/mnt/filepool/`. If the dataset we are preparing is called `amazon`, the directory structure would look like this:

```
amazon
|-- features.bsnap
|-- graph.bsnap
|-- labels.bsnap
|-- parts_<#partitions>/
  |-- graph.bsnap.edges
  |-- graph.bsnap.parts
```

where `graph.bsnap.edges` is a symlink to `../graph.bsnap`. Use the `add` command from above to add the NFS server to a special context called `nfs` so that `ec2man` knows where to look for it. Finally, run

```
local$ ./gnnman/mount-nfs-server
```

A.3.4 Running Dorylus.

Once the cluster has been setup, the code compiled, the Lambda functions installed, and the datasets prepared, we can run Dorylus. To run it use the following command from the `dorylus/` directory on your local machine:

```
# <dataset>: the dataset you prepared
# --l: the #lambdas/server
# --p: enable asynchronous pipelining
# --s: degree of staleness
# [cpu|gpu]: backend to use (blank means lambda)

./run/run-dorylus <dataset>
  [--l=#lambdas] [--lr=learning_rate]
  [--p] [--s=staleness] [cpu|gpu]
```

You will see the output of the Graph Servers, but can see the output of both the Graph and Weight Servers in `graphserver-out.txt` and `weightserver-out.txt`. More details of Dorylus's installation and deployment can be found in Dorylus's code repository.

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX ATC*, pages 125–137, 2017.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

[3] Amazon. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/, 2020.

[4] A. AWS. Announcing improved vpc networking for aws lambda functions. https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/, 2019.

[5] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

[6] X. Bresson and T. Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017.

[7] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 8(2):161–172, Oct. 2014.

[8] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[9] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 942–950, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[10] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, pages 215–226, 2014.

[11] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning, 2014.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[14] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, 2019.

[15] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, June 2014.

[16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a gpu-specialized parameter server. In *EuroSys*, 2016.

[17] DeepGraphLibrary. Why DGL? https://www.dgl.ai/pages/about.html, 2018.

[18] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844—3852, Red Hook, NY, USA, 2016.

[19] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, pages 2224–2232, 2015.

[20] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[21] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of A parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[22] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, pages 249–256, 2010.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[24] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[25] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[26] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034, 2015.

[28] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, pages 507–517, 2016.

[29] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *CIDR*, 2019.

[30] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, Red Hook, NY, USA, 2013.

[31] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[32] C. Huyen. Key trends from NeurIPS 2019. https://huyenchip.com/2019/12/18/key-trends-neurips-2019.html, 2019.

[33] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, pages 497–511, 2020.

[34] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with Roc. In *MLSys*, 2020.

[35] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken. Redundancy-free computation for graph neural networks. In *KDD*, pages 997–1005, 2020.

[36] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In *MLSys*, 2019.

[37] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *SoCC*, pages 445–451, 2017.

[38] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[39] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[40] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[41] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization. In *ICLR*, 2021.

[42] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *USENIX ATC*, pages 789–794, 2018.

[43] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, pages 427–444, 2018.

[44] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[45] M. Kustosz and B. Osinski. Trends and fads in machine learning – topics on the rise and in decline in ICLR submissions. https://deepsense.ai/key-findings-from-the-international-conference-on-learning-representations-iclr/, 2020.

[46] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, pages 31–46, 2012.

[47] J. B. Lee, R. A. Rossi, X. Kong, S. Kim, E. Koh, and A. Rao. Graph convolutional networks with motif-based attention. In *CIKM*, pages 499–508, 2019.

[48] J. Leskovec. Stanford network analysis project. https://snap.stanford.edu/, 2020.

[49] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[50] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR*, 2016.

[51] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016.

[52] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, , and U. Kang. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData*, pages 159–164, 2014.

[53] Q. Liu, M. Nickel, and D. Kiela. Hyperbolic graph neural networks. In *NIPS*, pages 8230–8241. Curran Associates, Inc., 2019.

[54] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[55] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, pages 443–457, 2019.

[56] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*, pages 527–543, 2017.

[57] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[58] M. Mariappan, J. Che, and K. Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *EuroSys*, page 83–98, 2021.

[59] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, page 25:1–25:16, 2019.

[60] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*, pages 43–52, 2015.

[61] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *ICML*, pages 2430—-2439, 2017.

[62] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[63] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *SOSP*, page 1–15, 2019.

[64] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[65] F. Niu, B. Recht, C. Re, and S. J. Wright. HOGWILD! a lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693—-701, 2011.

[66] NVIDIA. The cuSPARSE CUDA toolkit. https://docs.nvidia.com/cuda/cusparse/index.html, 2020.

[67] N. Peng, H. Poon, C. Quirk, K. Toutanova, and W. Yih. Cross-sentence n-ary relation extraction with graph LSTMs. *TACL*, 5:101–115, 2017.

[68] Reddit. The reddit datasets. https://www.reddit.com/r/datasets/, 2020.

[69] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[70] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[71] F. Scarselli and et al. The graph neural network model. *IEEE Trans. Neur. Netw.*, 20(1):61–80, Jan. 2009.

[72] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

[73] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, pages 235–239, 2014.

[74] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *USENIX ATC*, pages 317–332, 2016.

[75] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[76] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49—-66, 2005.

[77] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. https://arxiv.org/abs/2105.11118, 2021.

[78] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[79] K. Vora. LUMOS: Dependency-driven disk-based graph processing. In *USENIX ATC*, pages 429–442, 2019.

[80] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, 2016.

[81] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, pages 237–251, 2017.

[82] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA*, pages 861–878, 2014.

[83] K. Vora, C. Tian, R. Gupta, and Z. Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *ASPLOS*, page 223–236, 2017.

[84] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2016.

[85] A. D. Vose, J. Balma, D. Farnsworth, K. Anderson, and Y. K. Peterson. PharML.Bind: Pharmacologic machine learning for protein-ligand interactions, 2019.

[86] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, pages 389–404, 2017.

[87] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

[88] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, pages 763–782, 2018.

[89] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[90] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[91] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. *AAAI*, 33:346–353, Jul 2019.

[92] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[93] Z. Xianyi and M. Kroeker. OpenBLAS. https://www.openblas.net, 2019.

[94] H. Yang. Aligraph: A comprehensive graph neural network platform. In *KDD*, pages 3165–3166, 2019.

[95] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, pages 974–983, 2018.

[96] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim. Graph transformer networks. In *Annual Conference on Neural Information Processing Systems 2019*, pages 11960–11970, 2019.

[97] ZeroMQ. ZeroMQ networking library for C++. https://zeromq.org/, 2020.

[98] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, 2020.

[99] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181—-193, 2017.

[100] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung. GaAN: Gated attention networks for learning on large and spatiotemporal graphs. In A. Globerson and R. Silva, editors, *UAI*, pages 339–349, 2018.

[101] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[102] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[103] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[104] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[105] X. Zhu, W. Han, and W. Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.

GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs

Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding
University of California, Santa Barbara

Abstract

As the emerging trend of graph-based deep learning, Graph Neural Networks (GNNs) excel for their capability to generate high-quality node feature vectors (embeddings). However, the existing one-size-fits-all GNN implementations are insufficient to catch up with the evolving GNN architectures, the ever-increasing graph sizes, and the diverse node embedding dimensionalities. To this end, we propose **GNNAdvisor**, an adaptive and efficient runtime system to accelerate various GNN workloads on GPU platforms. First, GNNAdvisor explores and identifies several performance-relevant features from both the GNN model and the input graph, and uses them as a new driving force for GNN acceleration. Second, GNNAdvisor implements a novel and highly-efficient 2D workload management, tailored for GNN computation to improve GPU utilization and performance under different application settings. Third, GNNAdvisor capitalizes on the GPU memory hierarchy for acceleration by gracefully coordinating the execution of GNNs according to the characteristics of the GPU memory structure and GNN workloads. Furthermore, to enable automatic runtime optimization, GNNAdvisor incorporates a lightweight analytical model for an effective design parameter search. Extensive experiments show that GNNAdvisor outperforms the state-of-the-art GNN computing frameworks, such as Deep Graph Library (3.02× faster on average) and NeuGraph (up to 4.10× faster), on mainstream GNN architectures across various datasets.

1 Introduction

Graph Neural Networks (GNNs) emerge to stand on the frontline for handling many graph-based deep learning tasks (*e.g.,* node embedding generation for node classification [9, 14, 23] and link prediction [6, 28, 51]). Compared with standard methods for graph analytics, such as random walks [16, 47] and graph Laplacians [7, 34, 35], GNNs highlight themselves with the interleaved two-phase execution of both graph operations (scatter-and-gather [15]) at the *Aggregation* phase, and Neural Network (NN) operations (matrix multiplication) at the *Update* phase, to achieve significantly higher accuracy [27, 52, 55] and better generality [17]. Yet, the state-of-the-art GNN frameworks [11, 36, 53, 54], which follow a one-size-fits-all implementation scheme, often suffer from poor performance when handling more complicated GNN architectures (*i.e.,* more layers and higher hidden dimensionality in each layer) and diverse graph datasets.

Specifically, previous work that supports both GNN training and inference can be classified into two categories. The first type [36, 54] is built on popular graph processing systems and is combined with NN operations. The second type [11, 53], in contrast, starts with deep learning frameworks and is extended to support vector-based graph operations. However, these existing solutions are still preliminary and inevitably fall short in the following three major aspects, even on common computing platforms such as GPUs.

Failing to leverage GNN input information. GNN models demonstrate great diversity in terms of layer sequences, types of aggregation methods, and the dimension size of node embeddings. These profoundly impact the effectiveness of various system optimization choices. The diversity of input graphs further complicates the problem. Unfortunately, current GNN frameworks [11, 36, 53] follow a one-size-fits-all optimization scheme and fail to craft an optimization strategy that maximizes efficiency for a particular GNN application's settings. Some classical graph systems [2, 3] have exploited input characteristics to facilitate more efficient optimizations, but they only focus on simple graph algorithms like PageRank [45] while having no support for GNN models.

Optimizations not tailored to GNN. While the update phase in GNNs involves NN operations that are dense in computation and regular in memory access, the aggregation phase is usually sparse in computation and highly irregular in memory access. Without dedicated optimization, it will inevitably become the performance bottleneck. Existing GNN frameworks [11, 36, 53] simply extend the optimization schemes from classical graph systems [26, 54], and do not address the difference between GNN and graph processing. For example,

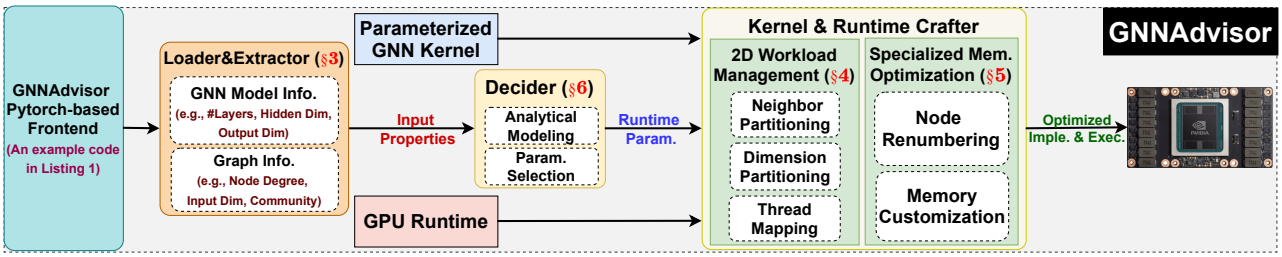


Figure 1: Overview of GNNAdvisor.

each node is associated with an embedding attribute in GNNs while each node only has a single scalar attribute in traditional graph processing. Such difference invokes novel design principles for GNNs towards more aggressive parallelism and more efficient memory optimization.

Listing 1: Example of a 2-layer GCN in GNNAdvisor.

```
import GNNAdvisor as GNNA
import torch
# import other packages ...

# Create a GCN class.
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim, nLayers):
        self.layers = torch.nn.ModuleList()
        self.layers.append(GNNA.GCNConv(inDim, hiDim))
        for i in range(nLayers - 2):
            layer = GNNA.GCNConv(hiDim, hiDim)
            self.layers.append(layer)
        self.layers.append(GNNA.GCNConv(hiDim, outDim))
        self.softmax = torch.nn.Softmax()

    def forward(self, X, graph, param):
        for i in range(len(self.layers)):
            X = self.layers[i](X, graph, param)
            X = self.ReLU(X)
        X = self.softmax(X)
        return X

# Define a two-layer GCN model.
model = GCN(inDim=100, hiDim=16, outDim=10, nLayers=2)

# Loading graph and extracting input propertities.
graphObj, inputInfo = GNNA.LoaderExtractor(graphFile,
                                           model)
# Set runtime parameters automatically.
X, graph, param = GNNA.Decider(graphObj, inputInfo)

# Run model.
predict_y = model(X, graph, param)

# Compute loss and accuracy.
# Gradient backpropagation for training.
```

Poor runtime support for input adaptability. Prior GNN frameworks [11,36,53] rely on a Python-based high-level programming interface for ease of user implementation. These frameworks employ static optimizations through a compiler or manually-optimized libraries. Nevertheless, some critical performance-related information for a GNN is only available at runtime (*e.g.*, node degree and embedding size). Without adaptable designs that can leverage such runtime information, we would easily suffer from an inferior performance because of the largely under-utilized the GPU computing resources and inefficient irregular memory access. This limitation motivates the need for runtime environments with flexible designs to handle a wide spectrum of inputs effectively.

To this end, we propose, GNNAdvisor, an adaptive and efficient runtime system for GNN acceleration on GPUs. GNNAdvisor leverages Pytorch [46] as the front-end to improve programmability and ease user implementation. We show a representative 2-layer Graph Convolutional Network (GCN) [27] in GNNAdvisor at Listing 1. At the low level, GNNAdvisor is built with C++/CUDA and integrated with Pytorch framework by using Pytorch Wrapper. It can be viewed as a new type of Pytorch operator with a set of kernel optimizations and runtime support. It can work seamlessly with existing operators from the Pytorch Framework. Data is loaded with the data loader written in Pytorch and passed as a Tensor to GNNAdvisor for computation on GPUs. Once the GNNAdvisor completes its computation at the GPU, it will pass the data Tensor back to the original Pytorch framework for further processing. As detailed in Figure 1, GNNAdvisor consists of several key components to facilitate the GNN optimization and execution on GPUs. First, GNNAdvisor introduces an input **`Loader&Extractor`** to exploit the input-level information that can guide our system-level optimizations. Second, GNNAdvisor incorporates a **`Decider`** consisting of analytical modeling for automatic runtime parameter selection to reduce manual effort in design optimization, and a lightweight node renumbering routine to improve graph structural locality. Third, GNNAdvisor integrates a **`Kernel&Runtime Crafter`** to customize our parameterized GNN kernel and CUDA runtime, which consists of an effective 2D workload management (considering both the number of neighbor nodes and the node embedding dimensionality) and a set of GNN-specialized memory optimizations.

Note that in this project, we mainly focus on the setting of single-GPU GNN computing, which is today's most popular design adopted as the key component in many state-of-the-art frameworks, such as DGL [53] and PyG [11]. Single-GPU GNN computing is desirable for two reasons: First, many GNN applications with small to medium size graphs (*e.g.*, molecule structure) can easily fit the memory of a single GPU. Second, in the case of large-size graphs that can only be handled by out-of-GPU-core and multi-GPU processing, numerous well-studied graph partition strategies (*e.g.*, METIS [22]) can cut the giant graphs into small-size subgraphs to make them suitable for a single GPU. Therefore, the optimization of

both the out-of-GPU-core (*e.g.*, GPU streaming processing) and multi-GPU GNN computation still largely demands performance improvements on a single GPU. Moreover, while our paper focuses on GNNs, our proposed methodology can be applied to optimize various types of irregular workload (*e.g.*, social network analysis) targeting GPUs as well.

Overall, we make the following contributions:

- We are the first to explore GNN input properties (§3) (*e.g.*, GNN model architectures and input graphs), and give an in-depth analysis of their importance in guiding system optimizations for GPU-based GNN computing.
- We propose a set of GNN-tailored system optimizations with parameterization, including a novel 2D workload management (§4) and specialized memory customization (§5) on GPUs. We incorporate the analytical modeling and parameter auto-selection (§6) to ease the design space exploration.
- Comprehensive experiments demonstrate the strength of GNNAdvisor over state-of-the-art GNN execution frameworks, such as Deep Graph Library (average 3.02×) and NeuGraph (average 4.36×), on mainstream GNN architectures across various datasets.

2 Background and Related Work

In this section, we introduce the basics of GNNs and two major types of GNN computing frameworks: *GPU-based graph systems* and *deep learning frameworks*.

2.1 Graph Neural Networks

Figure 2 visualizes the computation flow of GNNs in one iteration. GNNs compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding information at layer k ($k \geq 0$), as shown in Equation 1,

$$\begin{aligned} a_v^{(k+1)} &= \textbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \textbf{Update}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \quad (1)$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $h_v^{(0)}$ is computed from the task-specific features of a vertex (e.g., the text associated with the vertex, or some scalar properties of the entity that the vertex represents) via some initial embedding mapping that is used only for this ingest of symbolic values into the embedding space; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [17,27] just rely on the neighboring nodes while others [52] also leverage edge properties, by combining the dot product of the end-point nodes of each edge, along with any edge features (edge type and other attributes). The update function is generally composed of standard NN operations, such as a single fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the learnable weight and bias parameters, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers.

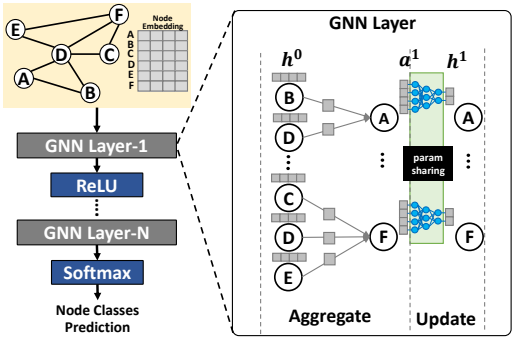


Figure 2: GNN General Computation Flow.

After passing through several iterations of aggregation and update (*i.e.*, several GNN layers), we will get the output embedding of each node, which can usually be used for various downstream graph-based deep learning tasks, such as node classification [9, 14, 23] and link prediction [6, 28, 51]. Note that the initial node embedding for GNN's input layer may come with the original graph dataset or can be generated by a set of graph embedding algorithms, such as [5, 10, 16], which is not included in the computation of GNNs models (generating the hidden and output node embeddings).

2.2 Graph Processing Systems

Numerous graph processing systems [26, 32, 33, 38, 54] have been proposed to accelerate traditional graph algorithms. The major commonalities of these systems include the vertex/node-centric programming abstraction, edge-centric processing, and system optimizations to reduce the computation irregularity (*e.g.*, workload imbalance) and memory access irregularity (*e.g.*, non-coalesced global memory access). However, extending these graph processing systems to support GNN computing meets with substantial challenges.

First, the common algorithm optimizations in graph processing may not benefit GNNs. For example, graph traversal algorithms, such as Breadth-first Search, rely on iterative computing on node frontiers (active neighbors). Therefore, a set of frontier-based optimizations, such push-pull traversal [32,33], and frontier filtering [32,33,54], have been extensively studied. However, GNNs consistently maintain fixed-sized frontiers (all neighbors) of each node across iterations.

Second, the system optimization techniques for graph processing would benefit GNNs only after careful adaption and calibration. For example, node/edge-centric processing [33, 54] and shard-based graph representation [26] are

tailored for processing nodes/edges represented with a single scalar attribute. In GNNs, there's another dimension for data parallelism, namely the embedding dimension, which tends to be large Therefore, previous design trade-offs between the coarse-grained node-level parallelism and node-value locality should be further extended to balance dimension-wise parallelism and node-embedding locality at a finer granularity.

Third, some essential functionalities of GNN computing are missing in graph systems. For example, the node update based on NN computing for both the forward value propagation and the complicated backward gradient propagation is not available in graph systems [26, 29, 32, 33, 38, 49, 54]. In contrast, Pytorch [46] and Tensorflow [1] feature an analytic differentiation function for automatic gradient computations on various deep learning model architectures and functions. Therefore, extending the graph-processing system to support GNN computing requires non-trivial efforts, and thus we develop GNNAdvisor on top of a deep learning framework.

2.3 Deep Learning Frameworks

Various NN frameworks have been proposed, such as Tensorflow [1], and Pytorch [46]. These frameworks provide the end-to-end training and inference support for traditional deep-learning models with various NN operators, such as linear and convolutional operators. These operators are highly optimized for Euclidean data (*e.g.*, image) but lack support for non-Euclidean data (*e.g.*, graph) in GNNs. Extending NN frameworks to support GNN that takes the highly-irregular graphs as the input is facing several challenges.

First, NN-extended GNN computing platforms [11, 53] focus on programmability and generality for different GNN models but lack efficient backend support to achieve high performance. For example, Pytorch-Geometric (PyG) [11] uses the torch-scatter [12] library implemented with CUDA as its major building block of graph aggregation operations. The torch-scatter implementation scales poorly when encountering large sparse graphs with high-dimensional node embedding because its kernel design essentially borrows the design principles of graph-processing systems by using excessive high-overhead atomic operations to support node embedding propagation. A similar scalability problem is also observed in Deep Graph Library (DGL) [53], which incorporates an off-the-shelf Sparse-Matrix Multiplication (SpMM) (*e.g.*, *csrmm2* in cuSparse [39]) for simple sum-reduced aggregation [17, 27] and leverages its own CUDA kernel for more complex aggregation scheme with edge attributes [52, 55].

Second, major computation kernels [11, 53] are hard-coded without design flexibility, which is essential to handle diverse application settings with different input graph sizes and node embedding dimensionality. From the high-level interface, users are only allowed to define the way of composing these kernels externally. Users are not allowed to customize kernels internally based on the known characteristics of GNN model architectures, GPU hardware, and graph properties.

3 Input Analysis of GNN Applications

In this section, we argue that the GNN input information can guide the system optimization, based on our key observation that different GNN application settings would favor different optimization choices. We introduce two types of GNN input information and discuss their potential performance benefits and extraction methods.

3.1 GNN Model Information

While the GNN update phase follows a relatively fixed computing pattern, the GNN aggregation phase shows high diversity. The mainstream aggregation methods of GNNs can be categorized into two types:

The first type is aggregation (*e.g.*, *sum*, and *min*) with only the embeddings of neighbor nodes, as in Graph Convolutional Network (GCN) [27]. For GNNs with this type of aggregation, the common design practice is to reduce the node embedding dimensionality during the update phase (*i.e.*, multiplying the node embedding matrix with the weight matrix) [11, 27, 53] before the aggregation (gather information from neighbor node embedding) at each GNN layer, thereby, largely reducing the data movements during the aggregation. In this case, improving memory locality would be more beneficial, in that more node embeddings can be cached in fast memory (*e.g.*, L1 cache of GPUs) to exploit performance benefits.

The second type is aggregation with special edge features (*e.g.*, weights, and edge vectors that are computed by combining source and target nodes) applied to each neighbor node, as in Graph Isomorphism Network (GIN) [55]. This type of GNN must work on large full-dimensional node embeddings to compute the special edge features at the node aggregation. In this case, the fast memory (*e.g.*, shared memory of GPU Stream-Multiprocessors) is not large enough to exploit memory locality. However, improving computation parallelization (*e.g.*, workload partitioning along the embedding dimension) would be more helpful, considering that workloads can be shared among more concurrent threads for improving overall throughput.

We illustrate this aggregation-type difference with the mathematical equations for GCN and GIN. With GCN, the output embedding $\mathbf{X}$ is computed as follows:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W}, \tag{2}$$

where $\hat{\mathbf{D}}$ is the diagonal node degree matrix; $\mathbf{W}$ is the weight matrix; $\hat{\mathbf{A}}$ is the graph adjacency matrix. For GIN, the output embedding $\mathbf{X}$ for each layer is computed as follows:

$$\mathbf{x}_i' = h\left((1+\varepsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \tag{3}$$

where h denotes a neural network, *e.g.*, an MLP, which maps node features x with input embedding dimension and output

embedding dimension; ϵ is a configurable/trainable parameter depending on the users' demands or application settings; $\mathcal{N}(i)$ denotes the neighbor IDs of the node i.

Assume we have GCN and GIN with hidden dimension 16, and the input dataset has a node embedding dimension of 128. In the case of GCN, we will first do node update (GEMM[1]-based linear transformation) of the node embedding, thus, at the aggregation, we only need to do aggregation on nodes with hidden dimension 16. In the GIN case, we have to do neighbor aggregation on nodes with 128 dimensions then do node update to linearly transform node embedding from 128 to 16 dimensions. Such an aggregation difference would also lead to different optimization strategies, where GCN would prefer more memory optimization on the small node embeddings while GIN would prefer more computing parallelism on the large node embeddings.

To conclude, the type of aggregation in GNNs should be considered for system-level optimization and it can be obtained by GNNAdvisor's built-in parser of GNN model proprieties.

3.2 Graph Information

Node Degree & Embedding Dimensionality: Real-world graphs generally follow the power-law distribution [50] of node degrees. Such distribution already causes workload imbalance in traditional graph processing systems [18, 25, 32]. In GNN aggregation, such workload imbalance would be exacerbated due to the higher dimensionality of the node embeddings if we perform node-centric workload partitioning. Moreover, node embedding would invalidate some cache-based optimizations that are originally applied to graph processing systems, since caches are usually small in size and insufficient to hold enough nodes with their embeddings. For example, in the graph processing scenarios with a scalar attribute for each node, we can improve performance by putting 16×10^3 nodes on the 64KB L1 cache of each GPU thread block. However, in typical GNNs with a 64-dimension embedding for each node, we can only fit 256 nodes on each GPU block's cache.

With node degree and embedding dimensionality information, new optimization opportunities for GNNs may appear because we can estimate the node's workload and its concrete composition based on such input information. If the workload size is dominated by the number of node neighbors (*e.g.*, large node degree), we may customize the design that could concurrently process more neighbors to increase the computing parallelism among neighbors. On the other hand, if the workload size is dominated by node embedding size (*e.g.*, high-dimensional node embedding), we may consider boosting the computing parallelism along the node embedding dimension. Note that the node degree and embedding dimension information can be extracted based on the loaded graph structure and node embedding vectors. GNNAdvisor manages the GNN workload based on such information (Section 4).

[1] General Matrix-Matrix Multiplication.

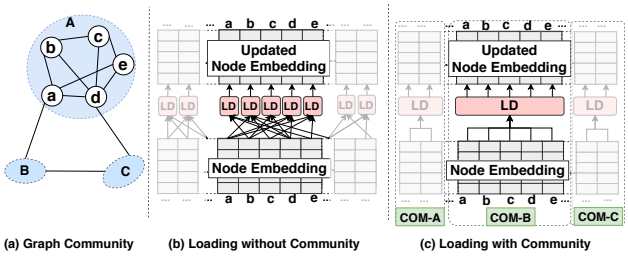


Figure 3: Graph community and its potential benefits. Note that "LD": loading operation. "COM": community.

Graph Community: Graph community [13, 30, 37] is one key feature of real-world graphs, which describes that a small group of nodes tend to hold "strong" intra-group connections (many edges) while maintaining "weak" connections (fewer edges) with the remaining part of the graph. A motivating example of GNN optimization with graph community structure is shown in Figure 3a. Existing node-centric aggregation employed by many graph processing systems [26, 54] is shown in Figure 3b, where each node will first load its neighbors and then do aggregation independently. This strategy can achieve great computation parallelism when each neighbor has a lightweight scalar attribute. In this case, the benefit of loading parallelization would offset the downside of duplicate loading of some shared neighbors. However, in GNN computing where node embedding size is large, this node-centric loading would trigger significant unnecessary memory access since the cost of duplicate neighbor loading is now dominant and not offset by per-node parallelism For example, aggregation of node a, b, c, d, and e would load the embeddings of 15 nodes in total and most of these loads are repeated (both node a and b load the same node d during the aggregation). Such loading redundancy is exacerbated with the increase of embedding dimensionality. On the other side, by considering the community structure of real-world graphs, unnecessary data loading for these "common" neighbors can be well reduced (Figure 3c), where aggregation only requires loads of 5 distinct nodes.

This idea sounds promising, but the effort to realize its benefits on GPUs is non-trivial. Existing approaches [19, 37] of exploiting the graph communities mainly target CPU platforms with a limited number of parallelized threads and MB-level cache sizes for each thread. Their major goal is to exploit the data locality for every single thread. GPUs, on the other side, are equipped with a massive number of parallel threads and KB-level cache sizes per thread. Therefore, the key to exploiting graph community on GPUs is to effectively exploit the data locality among threads by leveraging the L1 cache. Specifically, we need first capture the communities of a graph and then map such locality from input level (node-ID adjacency) to underlying GPU kernels (thread/warp/block-ID

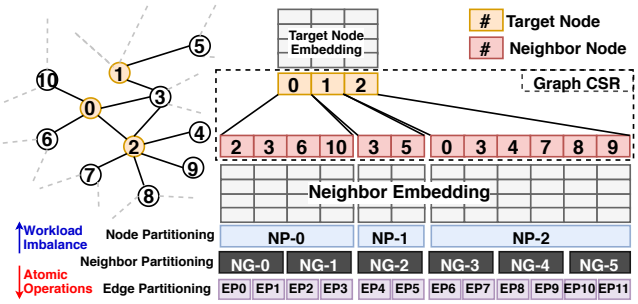


Figure 4: Neighbor Partitioning. Note that "NP": Node Partitioning; "EP": Edge Partitioning; "NG": Neighbor Group.

adjacency). The major hardware-level insight is that threads close in their IDs are more likely to share memory and computing resources, thus, improving the data spatial and temporal locality. GNNAdvisor handles all these details through community-aware node renumbering and GNN-specialized memory optimizations (Section 5).

4 2D Workload Management

GNNs employ a unique space in graph computations, due to the representation of each node by a high-dimensional feature vector (the embedding). GNN workloads grow in two major dimensions: *the number of neighbors* and *the size of the embedding dimension*. GNNAdvisor incorporates an input-driven parameterized 2D workload management tailored for GNNs, including three techniques: *coarse-grained neighbor partitioning*, *fine-grained dimension partitioning*, and *warp-based thread alignment*.

4.1 Coarse-grained Neighbor Partitioning

Coarse-grained neighbor partitioning is a novel workload balance technique tailored to GNN computing on GPUs. It aims to tackle the challenge of *inter-node workload imbalance* and *redundant atomic operations*.

Specifically, based on the loaded graph compressed-sparse row (CSR) representation, our coarse-grained neighbor partitioning will first break down the neighbors of a node into a set of equal-sized neighbor groups, and treat the aggregation workload of each neighbor group (NG) as the basic workload unit for scheduling. Figure 4 exemplifies an undirected graph and its corresponding neighbor partitioning result. The neighbors of Node-0 are divided into two neighbor groups (NG-0 and NG-1) with a pre-determined group size of 2. Neighbors (Node-3 and Node-5) of Node-1 are covered by NG-2, while the neighbors of Node-2 are spread among NG-{3,4,5}. To support the neighbor group, we introduce two components, the neighbor-partitioning module and the neighbor-partitioning graph store. The former is a lightweight module built on top of the graph loader by partitioning the graph CSR into equal-size groups. Note that each neighbor group only covers the neighbors of one target node for ease of scheduling and synchronization. The neighbor-partitioning graph store maintains the tuple-based meta-data of each neighbor group, including its IDs, starting and ending position of its neighbor nodes in the CSR representation, and the source node. For example, the meta-data of NG-2 will be stored as (2, 1, (4, 6)), where 2 is the neighbor-group ID, 1 is the target node ID, (4, 6) is the index range of the neighbor nodes in CSR.

The benefits of applying the aggregation based on partitioning neighbors are three-fold: 1) compared with the more coarse-grained aggregation based on node/vertex-centric partitioning [26], neighbor partitioning can largely mitigate the size irregularity of the workload units, which would improve GPU occupancy and throughput performance; 2) compared with the more fine-grained edge-centric partitioning (used by existing GNN frameworks, such as PyG [11], for batching and tensorization, and graph processing systems [33, 54] for massive computing parallelization), the neighbor-partitioning solution can avoid the overheads of managing many tiny workload units that might hurt the performance in many ways, such as scheduling overheads and the excessive amount of synchronizations; 3) it introduces a performance-related parameter, **neighbor-group** size (*ngs*), which is used for design parameterization and performance tuning. Neighbor partitioning works at a coarse granularity of individual neighbor nodes. It can largely mitigate the workload imbalance problem for low-dimension settings. For high-dimensional node embeddings, we employ a fine-grained dimension partitioning discussed in the next subsection to further distribute workloads of each neighbor group to threads. Note that when the number of neighbors is not divisible by the neighbor group size, it will raise neighbor-group imbalance. Such irregularity can be amortized by setting the neighbor-group size to a small number (*e.g.*, 3).

4.2 Fine-grained Dimension Partitioning

GNN distinguishes itself from traditional graph algorithms in its computation on the node embedding. To explore the potential acceleration parallelism along this dimension, we leverage a fine-grained dimension partitioning to further distribute the workloads of a neighbor group along the embedding dimension to improve aggregation performance. As shown in Figure 5, the original neighbor-group workloads are evenly distributed to 11 consecutive threads, where each thread manages the aggregation along one dimension independently (*i.e.*, accumulation of all neighbor node embeddings towards the target node embedding). If the dimension size is larger than the number of working threads, more iterations would be required to finish the aggregation.

There are two major reasons for using dimension partitioning. First, it can accommodate a more diverse range of

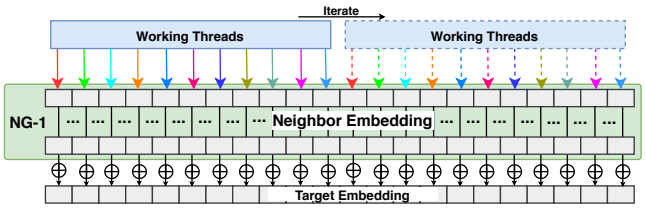


Figure 5: Dimension Partitioning. ⊕: Accumulated add.

embedding dimension sizes. We can either increase the number of concurrent dimension workers or enable more iterations to handle the dimension variation flexibly. This is essential for modern GNNs with increasingly complicated model structures and different sizes of embedding dimension. Second, it introduces another performance-related parameter – the number of working threads (**dimension-worker** (*dw*)) for design customization. The value of this parameter can help to balance the thread-level parallelism and the single thread efficiency (*i.e.*, computation workload per thread).

4.3 Warp-based Thread Alignment

While the above two techniques answer how we balance GNN workloads logically, how to map these workloads to underlying GPU hardware for efficient execution is still unresolved. One straightforward solution is to assign consecutive threads to concurrently process workloads from different neighbor groups (Figure 6a). However, different behaviors (*e.g.*, data manipulation and memory access operations) among these threads would result in thread divergence and GPU underutilization. Threads from the same warp proceed in a single-instruction-multiple-thread (SIMT) fashion and the warp scheduler can only serve one type of instruction per cycle. Therefore, different threads have to wait for their turn for execution until the Stream-Multiprocessor (SM) warp scheduler issues their corresponding instructions.

To tackle this challenge, we introduce a warp-aligned thread mapping in coordination with our neighbor and dimension partitioning to systematically capitalize on the performance benefits of balanced workloads. As shown in Figure 6b, each warp will independently manage the aggregation workload from one neighbor group. Therefore, the execution of different neighbor groups (*e.g.*, NG-0 to NG-5) can be well parallelized without inducing warp divergence. There are several benefits in employing warp-based thread alignment. First, inter-thread synchronization (*e.g.*, atomic operations) can be minimized. Threads of the same warp are working on different dimensions of the same neighbor group, thus no conflicts occur for either global or shared memory accesses by threads from the same warp.

Second, the workload of a single warp is reduced and different warps will process more balanced workloads. Therefore, more small warps can be managed flexibly by SM warp schedulers to improve overall parallelism. Considering the

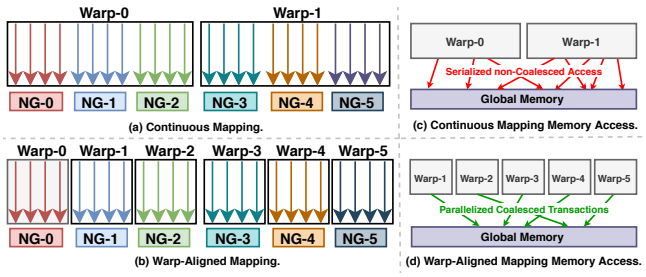


Figure 6: Warp-based Thread Alignment.

unavoidable global memory access of each warp during aggregation, increasing the number of warps can improve SM occupancy to hide latency. Third, memory access can be coalesced. Threads with consecutive IDs from the same warp will access continuous memory addresses in global memory for node embeddings. Therefore, compared with continuous thread mapping (Figure 6c), warp-aligned thread mapping can merge memory requests from the same warp into one global memory transaction (Figure 6d).

5 Specialized Memory Optimization

To further exploit the benefits of 2D workload, we introduce GNN-specialized memory optimizations, *community-aware node renumbering* and *warp-aware memory customization*.

5.1 Community-aware Node Renumbering

To explore the performance benefits of graph community (Section 3.2), we incorporate lightweight node renumbering by reordering node IDs to improve the temporal/spatial locality during GNN aggregation without compromising output correctness. The key idea is that the proximity of node IDs would project to the adjacency of computing units on GPU where they get processed. In GNNAdvisor, our 2D workload management assigns neighbor groups of a node to consecutive warps based on their node ID. If two nodes are assigned with consecutive IDs, their corresponding neighbor groups (warps) would be close to each other in their warp IDs as well. Thus, they are more likely to be scheduled closely on the same GPU SM with a shared L1 cache to improve the data locality on loaded common neighbors. To apply node renumbering effectively, two key questions must be addressed.

When to apply: While graph reordering provides potential benefits for performance, we still need to figure out *what kind of graph would benefit from such reordering optimization*. Our key insight is that for graphs already in a shape approximating block-diagonal pattern in their adjacency matrix (Figure 7a), reordering could not bring more locality benefits, since nodes within each community are already close to each other in terms of their node-IDs. For graphs with a more irregular shape (Figure 7b), where edge connections are distributed among nodes with an irregular pattern, the reordering could

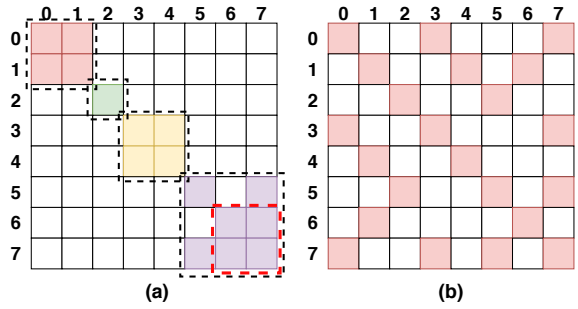


Figure 7: Graph Edge Connection Patterns. Note that each colored square represents the edge between two nodes. Different colors in (a) represent edges from different communities. The red dot-line box indicates the sub-community.

bring notable performance improvement (up to 2× speedup, later discussed in Section 7.5). To this end, we propose a new metric – *Averaged Edge Span* (AES), to determine whether it is beneficial to conduct a graph reordering.

$$\textbf{AES} = \frac{1}{\#E} \sum_{(src_{id}, trg_{id}) \in E} |src_{id} - trg_{id}| \tag{4}$$

where E is the edge set of the graph; $\#E$ is the number of total edges; src_{id} and trg_{id} are the source and target node IDs of each edge. Computing AES is lightweight and can be done on-the-fly during the initial graph loading. Our profiling of a large corpus of graphs also shows that when $\sqrt{AES} > \lfloor \frac{\sqrt{\#N}}{100} \rfloor$ node numbering is more likely to improve runtime performance.

How to apply: We leverage Rabbit Reordering [2], which is a fully parallelized and low-cost graph reordering technique. Specifically, it first maximizes the graph modularity by hierarchically merging edges and clustering nodes. And it then generates node order within each cluster through DFS traversal. Rabbit Reordering has also been proved to outperform other graph clustering approaches [4, 8, 21, 22, 48], including Community-based methods, such as METIS [22], and BFS-based methods, such as Reverse Cuthill-McKee (RCM) [8]) in terms of better quality (data locality) of the captured graph communities, the ease of parallelization, and performance. More importantly, Rabbit Reordering can capture the graph communities hierarchically (*i.e.*, a set of smaller sub-communities are included in a larger community, as exemplified in Figure 7a). Such communities at different levels of granularities would be a good match for the GPU cache hierarchy, where smaller sub-communities (occupying one SM) can enjoy the data locality benefit from the L1 cache, while larger communities (occupying multiple SMs) can enjoy the data locality from the larger L2 cache. We quantitatively discuss such a locality benefit in Section 7.4.

5.2 Warp-aware Memory Customization

Existing works [11, 54] utilize a large number of global memory accesses for reading and writing the embedding and a large number of atomic operations for aggregation (a reduction operation). However, this approach leads to heavy overhead and fails to exploit the potential benefits from shared memory. In particular, when aggregating on a target node with k neighbor groups (each has ngs neighbors with Dim-Dimensional embeddings) into a Dim-dimensional embedding, it involves $O(k \cdot ngs \cdot Dim)$ atomic operations and $O(k \cdot ngs \cdot Dim)$ global memory accesses.

By contrast, we propose a warp-centric shared memory optimization technique. Our key insight is that by customizing shared memory layout according to the block-level warp organization pattern (Figure 7), we can significantly reduce the number of atomic operations and global memory access. First of all, we reserve a shared memory space ($4 \times Dim$ bytes for floating-point embeddings) for the target node of each neighbor group (warp), such that the threads from a warp can cache the intermediate results of reduction in shared memory. Later on, within a thread block, we designate only one warp (called *leader*) for copying the intermediate results of each target node to global memory considering that neighbors of each node can be spread across different warps. The detailed customization procedure is described in Algorithm 1. Specifically, each warp (maintained in *warpPtr*) has three properties: *nodeSharedAddr* (a shared memory address for the aggregation result of a neighbor-group), *nodeID* (the ID of the target node), and *leader* (a boolean flag indicating whether the current warp is a leader warp for flushing out the result from the shared memory to the global memory). The major customization routine (Line 4 to Line 22) handles different warps based on their index position relative to thread blocks. Note that such a shared memory customization is low-cost and is done only once on-the-fly with the regular graph initialization process before the GPU kernel execution.

In our design, when a target node with k neighbor groups (each has ngs neighbors with Dim-dimensional embeddings), it involves $O(Dim)$ atomic operations and $O(Dim)$ global memory accesses. To this end, we can save the atomic operations and global memory access by $(k \cdot ngs)\times$, thus significantly accelerating the aggregation operations. Here, we treat ngs as a hyper-parameter to balance memory access efficiency and computation parallelism, and we further discuss its value selection in Section 6.

6 Design Optimization

The parameters in our GPU kernel configurations can be tuned to accommodate various GNN models with graph data sets. But it is not yet known how to automatically select the parameters which can deliver the optimal performance. In this section, we introduce the analytical model and the auto parameter selection in the **`Decider`** of GNNAdvisor.

Analytical Modeling: The performance/resource analytical model of GNNAdvisor has two variables, workload per

Algorithm 1 Warp-aware Memory Customization.

```
▷ Compute #neighbor-groups (#warps).
warpNum = neighborGroups = computeGroups(ngs);
▷ Compute the number of warps per thread block.
warpPerBlock = floor(threadPerBlock/threadPerWarp)
▷ Initialize tracking variables.
cnt = 0; local_cnt = 0; last = 0;
while cnt < warpNum do
    ▷ Warp in the front of a thread block.
    if cnt % warpPerBlock == 0 then
        warpPtr[cnt].nodeSharedAddr = local_cnt × Dim;
        last = warpPtr[cnt].nodeID;
        warpPtr[cnt].leader = true;
    ▷ Warp in the middle of a thread block.
    else
        ▷ Warp with the same target node as
          its predecessor warp.
        if warpPtr[cnt].nodeID == last) then
            warpPtr[cnt].nodeSharedAddr = local_cnt;
        ▷ Warp with the different target node as
          its predecessor warp.
        else
            local_cnt++;
            warpPtr[cnt].nodeSharedAddr = local_cnt;
            last = warpPtr[cnt].nodeID;
            warpPtr[cnt].leader = true;
        end if
    end if
    ▷ Next warp belongs to a new thread block.
    if (++cnt)%warpPerBlock == 0 then
        local_cnt = 0;
    end if
end while
```

thread (*WPT*), and shared memory usage per block (*SMEM*).

$$\textbf{WPT} = ngs \times \frac{Dim}{dw}, \quad \textbf{SMEM} = \frac{tpb}{tpw} \times Dim \times FloatS \tag{5}$$

where *ngs* and *dw* is the neighbor-group and dimension-worker size (Section 4.2), respectively; *Dim* is the node embedding dimension; *IntS* and *FloatS* are both 4-byte on GPUs; *tpb* is the thread-per-block and *tpw* is the thread-per-warp; *tpw* is 32 for GPUs, while *tpb* is selected by users.

Parameter Auto Selection: To determine the value of the *ngs* and *dw*, we follow two steps. First, we determine the value of *dw* based on *tpw* (hardware constraint) and *Dim* (input property), as shown in Equation 6. Note that we develop this equation by profiling different datasets and GNN models.

$$dw = \begin{cases} tpw & Dim \geq tpw \\ \frac{tpw}{2} & Dim < tpw \end{cases} \tag{6}$$

Second, we determine the value of *ngs* based on the selected *dw* and the user-specified *tpb*. The constraints include making $WPT \approx 1024$ and $SMEM \leq SMEMperBlock$. Note that $SMEMperBlock$ is 48KB to 96KB on modern GPUs [42, 44]. Across different GPUs, even though the number of CUDA cores and global memory bandwidth would be different, the single-thread workload capacity (measured by *WPT*) remains similar. *tpb* is usually chosen as a power of 2 but less than or equal 1024. Our insight based on micro-benchmarking and previous literature [56] shows that smaller blocks (1 to 4 warps, *i.e.*, $32 \leq tpb \leq 128$) can improve SM warp scheduling flexibility and avoid tail effects, thus leading to higher GPU occupancy and throughput. We further demonstrate the effectiveness of our analytical model in Section 7.5.

7 Evaluation

In this section, we comprehensively evaluate GNNAdvisor in terms of the performance and adaptability on various GNN models, graph datasets, and GPUs.

7.1 Experiment Setup

Benchmarks: We choose the two most representative GNN models widely used by previous work [11, 36, 53] on node classification tasks to cover different types of aggregation. 1) Graph Convolutional Network (GCN) [27] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [17], and differentiable pooling (Diffpool) [57]. Therefore, improving the performance of GCN will also benefit a broad range of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions*, which is also the setting from the original paper [27]. 2) Graph Isomorphism Network (GIN) [55]. GIN differs from GCN in its aggregation function, which weighs the node embedding values from the node itself. In addition, GIN is also the reference architecture for many other advanced GNNs with more edge properties, such as Graph Attention Network (GAT) [52]. For GIN evaluation, we use the setting: *5 layers with 64 hidden dimensions*, which is the setting used in the original paper [55].

Baselines: we choose several baseline implementations for comparison. 1) Deep Graph Library (DGL) [53] is the state-of-the-art GNN framework on GPUs, which is built upon the famous tensor-oriented platform – Pytorch [46]. DGL significantly outperforms the other existing GNN frameworks [11] over various datasets on many mainstream GNN architectures. Therefore, we make an in-depth comparison with DGL in our evaluation; 2) Pytorch-Geometric (PyG) [11] is another GNN framework in which users can define their edge convolutions when building customized GNN aggregation layers; 3) NeuGraph [36] is a dataflow-centered GNN system on GPUs built on Tensorflow [1]; 4) Gunrock [54] is the GPU-based graph processing framework with state-of-the-art performance on traditional graph algorithms (*e.g.*, PageRank).

Datasets: We cover all three types of datasets, which have been used in previous GNN-related work [11, 36, 53]. Type I graphs are the typical datasets used by previous GNN algorithm papers [17, 27, 55]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. Type II graphs [24] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [11]. Each dataset consists of a set of

Table 1: Datasets for Evaluation.

Type	Dataset	#Vertex	#Edge	Dim.	#Class
I	Citeseer	3,327	9,464	3,703	6
	Cora	2,708	10,858	1,433	7
	Pubmed	19,717	88,676	500	3
	PPI	56,944	818,716	50	121
II	PROTEINS_full	43,471	162,088	29	2
	OVCAR-8H	1,890,931	3,946,402	66	2
	Yeast	1,714,644	3,636,546	74	2
	DD	334,925	1,686,092	89	2
	TWITTER-Partial	580,768	1,435,116	1,323	2
	SW-620H	1,889,971	3,944,206	66	2
III	amazon0505	410,236	4,878,875	96	22
	artist	50,515	1,638,396	100	12
	com-amazon	334,863	1,851,744	96	22
	soc-BlogCatalog	88,784	2,093,195	128	39
	amazon0601	403,394	3,387,388	96	22

small graphs, which only have intra-graph edge connections without inter-graph edge connections. Type III graphs [27,31] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in structure, which is challenging for most of the existing GNN frameworks. Details of these datasets are listed in Table 1.

Platforms & Metrics: We implement GNNAdvisor's backend with C++ and CUDA C and its front-end with Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU [20] and a Quadro P6000 [42] GPU. Besides, we use Tesla V100 [44] GPU on the DGX-1 system [40] to demonstrate the generality of GNNAdvisor. Runtime parameters of different input settings are optimized by GNNAdvisor `Decider`. To measure the performance speedup, we calculate the averaged latency of 200 end-to-end inference (forward propagation) or training (forward+backward propagation).

7.2 Compared with DGL

In this section, we first conduct a detailed experimental analysis and comparison with DGL on GNN inference, then extend our comparison for GNN training. As shown in Figure 8, GNNAdvisor achieves 4.03× and 2.02× speedup on average compared to DGL [53] over three types of datasets for GCN and GIN on inference, respectively. We next provide detailed analysis and give insights for each type of datasets.

Type I Graphs: The performance improvement against DGL is significantly higher for GCN (on average 6.45×) than GIN (on average 1.17×). The major reason is their different GNN computation patterns. For GCN, node dimension reduction (DGEMM) is always placed before aggregation. This largely reduce data movement and thread synchronization overheads during the aggregation phase, which could gain more benefits from GNNAdvisor's 2D workload management and specialized memory optimization for data locality improvements. GIN, on the other side, has aggregation phase that must be finished before the node dimension reduction. Thus, it cannot avoid high-volume memory access and data

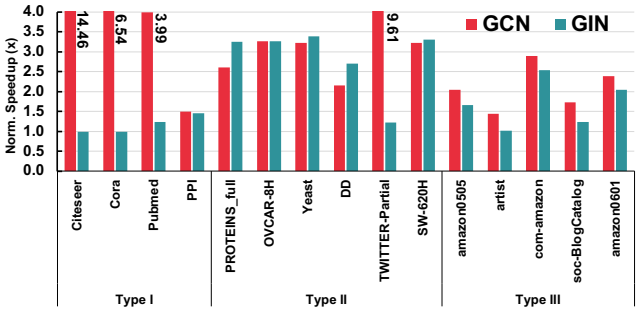


Figure 8: Inference speedup (×) over DGL on GCN and GIN.

movements during the aggregation phase. Therefore, it gets lower benefits from the data locality and the shared memory on GPUs for fast and low-overhead memory access. However, our fine-grained dimension partitioning can still handle these high-dimensional cases effectively.

Type II Graphs: Performance shows less difference between GCN (4.02×) and GIN (2.86×) on the same datasets except for *TWITTER-Partial*, which has the highest node embedding dimension (1323) in Type II graphs. It is worth noticing that the speedup for GIN is consistently better compared with Type I. There are two major reasons: 1) node feature dimension is much lower (average 66.5, excluding *TWITTER-Partial*) versus Type I (average 1421), which can gain more performance benefits from data spatial and temporal locality of our specialized memory optimizations; 2) Type II graphs intrinsically have good locality in their graph structure. The reason is that Type II datasets consist of small graphs with very dense intra-graph connections but no inter-graph edges, plus nodes within each small graph are assigned with consecutive IDs. Therefore, the performance gains of such graph-structure locality can be scaled up when combining with GNNAdvisor's efficient workload and memory optimizations.

Type III Graphs: The speedup is also evident (average 2.10× for GCN and average 1.70× for GIN) on graphs with a large number of nodes and edges, such as *amazon0505*. The reason is the high overhead inter-thread synchronization and global memory access can be well reduced through our 2D workload management and specialized memory optimization. Besides, our community-aware node renumbering further facilitates an efficient workload sharing among adjacent threads (working on a group of nodes) through improving the data spatial/temporal locality. On the dataset *artist*, which has the smallest number of nodes and edges within Type III, we notice a lower performance speedup for GIN. And we find that the *artist* dataset has the highest standard deviation of graph community sizes within Type III graphs, which makes it challenging to 1) use the group community information to capture the node temporal and spatial locality in the GNN aggregation phase, and 2) capitalize on the performance benefits of using such a community structure for guiding system-level optimizations (*e.g.*, warp-aligned thread mapping and shared memory customization) on GPUs, which have a fixed number

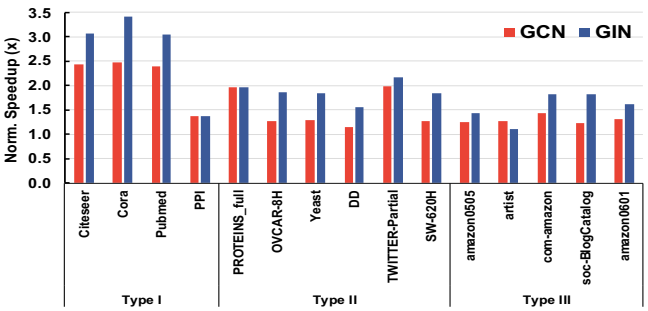


Figure 9: Training speedup (×) over DGL on GCN and GIN.

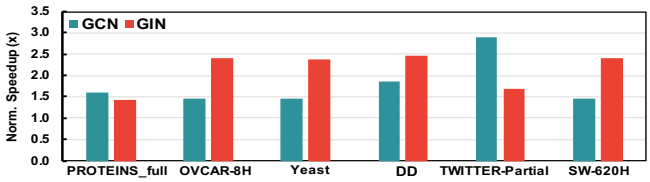


Figure 10: Training speedup (×) over PyG on GCN and GIN.

Table 2: Latency (ms) comparison with NeuGraph (NeuG).

Dataset	NeuG (ms)	Ours (ms)	Speedup
reddit-full	2460	599.69	4.10×
enwiki	1770	443.00	3.99×
amazon	1180	474.57	2.48×

of computation and memory units within each block/SM.

Kernel Metrics: For detailed kernel metrics analysis, we utilize NVProf [41] to measure two performance-critical (computation and memory) CUDA kernel metrics: *Stream Processor (SM) efficiency* and *Cache (L1 + L2 + Texture) Hit Rate*. GNNAdvisor achieves on average 24.47% and 12.02% higher SM efficiency compared with DGL for GCN and GIN, respectively, which indicates that our 2D workload management can strike a good balance between the single-thread efficiency and the multi-thread parallelism that are crucial to the overall performance improvement. GNNAdvisor achieves on average 75.55% and 126.20% better cache hit rate compared with DGL for GCN and GIN, correspondingly, which demonstrates the benefit of specialized memory optimizations.

Training Support: We also evaluate the training performance of GNNAdvisor on all three types of datasets compared with the DGL on both GCN and GIN. Compared with inference, training is more challenging, since it involves more intensive computation with the forward value propagation and the backward gradient propagation, both of which heavily rely on the underlying graph aggregation kernel for computation. As shown in Figure 9, GNNAdvisor consistently outperforms the DGL framework with average 1.61× and average 2.00× speedup on GCN and GIN, respectively, which shows the strength of our input-driven optimizations. The key difference between training and inference of GNNs is two-fold: First, backpropagation is needed in training. This step benefits from our improvements, as the backpropagation step is similar to the forward computation during the inference, and all the proposed methods are still beneficial; Second, training incurs extra memory and data movement overheads for storing/accessing the activations of the forward pass until gradients can be propagated back.

7.3 Compared with other Frameworks

We compare with DGL on all input settings, since DGL is the overall best-performance GNN framework. In this section, we further compare GNNAdvisor with three other representative GNN computing frameworks on their best settings.

Compared with PyG: As shown in Figure 10, GNNAdvisor can outperform PyG with 1.78× and 2.13× speedup on average for GCN and GIN, respectively. For GCN, GNNAdvisor achieves significant speedup on datasets with high-dimensional node embedding, such as *TWITTER-Partial*, through 1) node dimension reduction before aggregation and 2) workload sharing among neighbor partitions and dimension partitions. For GIN, GNNAdvisor reaches 2.45× speedup on datasets with a higher average degree, such as *DD*, since GNNAdvisor can effectively distribute the workload of each node along their embedding dimension to working threads while balancing the single-thread efficiency and inter-thread parallelism. PyG, however, achieves inferior performance because 1) it has poor thread management in balancing workload and controlling synchronization overhead; 2) it heavily relies on the scatter-and-gather kernel, which lacks flexibility.

Compared with NeuGraph: For a fair end-to-end training comparison with NeuGraph that has not open-sourced its implementation and datasets, we 1) use the GPU (Quadro P6000 [42]) that is comparable with the GPU of NeuGraph (Tesla P100 [43]) in performance-critical factors, such as GPU architecture (both have the Pascal architecture) and the number of CUDA cores; 2) use the same set of inputs as NeuGraph on the same GNN architecture [36]; 3) use the datasets that are presented in their paper and are also publicly available. As shown in Table 2, GNNAdvisor outperforms NeuGraph with a significant amount of margin (1.3× to 7.2× speedup) in terms of computation and memory performance. NeuGraph relies on general GPU kernel optimizations and largely ignores the input information. Moreover, the optimizations in NeuGraph are built-in and fixed inside the framework without performance tuning flexibility. In contrast, GNNAdvisor leverages GNN-featured GPU optimizations and demonstrates the key contribution of input insights for system optimizations.

Compared with Gunrock: We make a performance comparison between GNNAdvisor and Gunrock [54] on a single neighbor aggregation kernel of GNNs (*i.e.*, the Sparse-Matrix Dense-Matrix Multiplication (SpMM)) over the Type III graphs. As shown in Figure 11, GNNAdvisor outperforms Gunrock with 2.89× to 8.41× speedup. There are two major reasons behind such a evident performance improvement on the sparse GNN computation: 1) Gunrock focuses on graph-

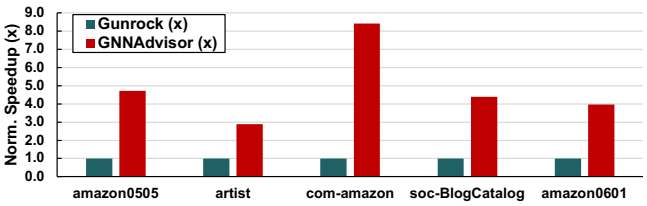


Figure 11: Speedup (×) comparison with Gunrock.

algorithm operators (*e.g.*, frontier processing) but lacks efficient support for handling high-dimensional node embedding; 2) Gunrock leverages generic optimizations without considering the input differences, thus, losing the adaptability for handling different GNN inputs efficiently.

7.4 Optimization Analysis

In this section, we explore and analyze the optimizations used in Sections 4 and 5 in detail.

Neighbor partitioning: From Figure 12a, we can see that with the increase of the neighbor-group size, the running time of GNNAdvisor will first decrease. The increase of the neighbor-group size saturates the computation capability of each thread meanwhile improving the data locality and reducing the number of atomic operations (*i.e.*, inter-thread synchronization overhead). However, when the neighbor-group size becomes larger than a certain threshold (*e.g.*, 32 for the *artist* dataset), each thread reaches its computation capacity upper bound, and further increasing the neighbor-group size offers no more performance benefit instead increases the overall latency.

Dimension partitioning: As shown in Figure 12b, the dimension worker impact is more evident in performance compared with the neighbor-group size at the range from 1 to 16. When the number of dimension worker increases from 16 to 32, the runtime performance shows very minor difference due to the already balanced single-worker efficiency and multi-worker parallelism. Therefore, further increase the number of dimension workers brings no more benefits.

Node renumbering: We demonstrate the benefit of node renumbering by profiling Type III datasets for GCN and GIN. As shown in Figure 12c, renumbering nodes within a graph can bring up to 1.74× and 1.49× speedup for GCN and GIN, respectively. The major reason is that our community-aware node renumbering can increase the data spatial and temporal locality during GNN aggregation.

To quantify such locality benefits, we extract the detailed GPU kernel metric – memory access in terms of read and write bytes from DRAM for illustration. Our CUDA kernel metric profiling results show that node renumbering can effectively reduce the memory access overhead (on average 40.62% for GCN and 42.33% for GIN) during the runtime since more loaded node embeddings are likely to be shared among the nodes with consecutive IDs. We also notice one input case that benefits less from our optimization – *artist*, since 1) the community size inside *artist* displays a large variation (high standard deviation), making it challenging to capture the neighboring adjacency and locality; 2) such a variation hurdles system-level (computation and memory) optimizations to effectively capitalize on the locality benefits of renumbering.

Block-level optimization: We show the optimization benefits of our block-level optimization (including warp-aligned thread mapping, and warp-aware shared memory customization). We analyze two kernel metrics (*atomic operations reduction* and *DRAM access reduction*) on three large graphs for illustration. As shown in Figure 12d, GNNAdvisor can effectively reduce the atomic operations and DRAM memory access by an average 47.85% and 57.93%. This result demonstrates 1) warp-aligned thread mapping based on neighbor partitioning can effectively reduce a large portion of atomic operations; 2) warp-aware shared memory customization can avoid a significant amount of global memory access.

7.5 Additional Studies

Hidden dimensions of GNN: In this experiment, we analyze the impact of the GNN architecture in terms of the size of the hidden dimension for GCN and GIN. As shown in Figure 13a, we observe that with the increase of hidden dimension of GCN, the running time of GNNAdvisor is also increased due to more computation (*e.g.*, additions) and memory operations (*e.g.*, data movements) during the aggregation phase and a larger size of the node embedding matrix during the node update phase. Meanwhile, we also notice that GIN shows a larger latency increase versus GCN, mainly because of the number of layers (2-layer GCN *vs.* 5-layer GIN) that make such a difference more pronounced.

Overhead analysis: Community-aware node renumbering is the major source of overhead for leveraging GNN input information, and other parts are negligible. Here as a case study, we evaluate its overhead on the training phase of GCN on Type III graphs, given the optimization decision from our GNNAdvisor `Decider` (as discussed in Section 5). Here we use training for illustration; inference in a real GNN application setting would also use the same graph structure many times [17, 27, 27] with different node embeddings inputs. As shown in Figure 13b, node-renumbering overhead is consistently small (average 4.00%) compared with overall training time. We thus conclude that such one-time overhead can be amortized over GNN running time, which demonstrates its applicability in real-world GNN applications.

Performance on Tesla V100: To demonstrate the potential of GNNAdvisor in the modern data-center environment, we showcase the performance of GNNAdvisor on an enterprise-level GPU – Tesla V100 [44]. As shown in Figure 13c, GNNAdvisor can scale well towards such a high-end device, which can achieve 1.97× and 1.86× speedup compared with P6000 for GCN and GIN due to more computa-

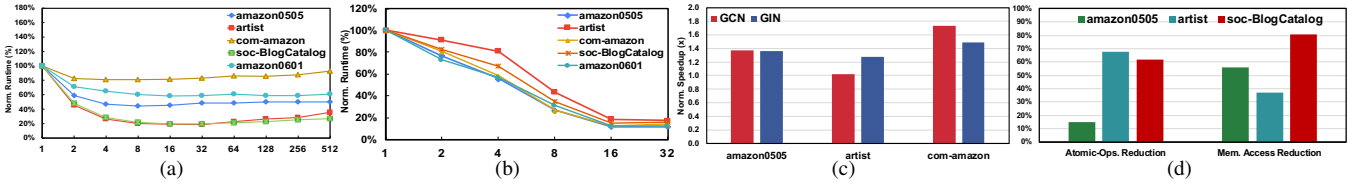


Figure 12: Optimization Analysis. (a) Normalized latency as the neighbor group size (*ngs*) grows (latency at $ngs = 1$ is set as 100%); (b) Normalized latency as the number of dimension workers grows (latency at $dw = 1$ is set as 100%); (c) Normalized speedup when using node renumbering compared to without renumbering; (d) Normalized GPU kernel metrics when using block-level optimizations compared to without block-level optimizations.

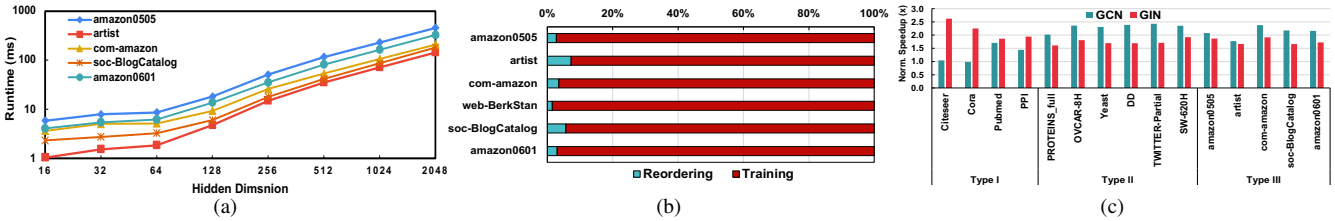


Figure 13: Additional Studies. (a) Latency (*ms*) analysis as the hidden dimension grows on GCN; (b) Overhead (%) analysis for node renumbering; (c) Speedup (×) on Tesla V100 over Quadro P6000 (set as 1×).

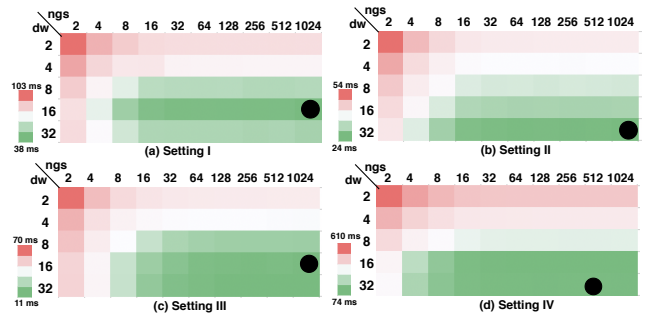


Figure 14: Parameter Selection for Four Settings. Note that the solid-black dot indicates the parameter (*dw* and *ngs*) selected by GNNAdvisor **`Decider`** based on analytical modeling.

tion resources (*e.g.*, 2.6× SMs, and 1.33× CUDA cores, and 1.13× throughput performance) and higher memory bandwidth (*e.g.*, 2.08× peak memory bandwidth). This comparison shows that GNNAdvisor well adapts towards more advanced GPU hardware for seeking better performance. We also foresee that our current work of GNNAdvisor can be extended to the multi-GPU or distributed data center, benefiting overall performance by improving single GPU efficiency.

Parameter selection: To show the effectiveness of our analytical modeling in kernel parameter selection, we consider four different settings: I: *amazon0505* on GCN at P6000 GPU as our base setting; II: *amazon0505* GCN on V100 to demonstrate device adaptation; III: *amazon0505* and *soc-BlogCatalog* on P6000 to demonstrate adaptation to different datasets; IV: *amazon0505* on GIN at P6000 to demonstrate adaptation to a different GNN model architectures. As shown in Figure 14, our parameter selection strategy can pinpoint the optimal low-latency design for the above four settings. This demonstrates the effectiveness of our analytical modeling in assisting parameter selection to optimize the performance of GNN computation.

8 Conclusion

In this work, we propose, GNNAdvisor, an adaptive and efficient runtime system for GNN acceleration on GPUs. Specifically, we explore the potential of GNN input-level information in guiding system-level optimizations. We further propose a set of GNN-tailored system-level optimizations (*e.g.*, 2D workload management, and specialized memory optimizations) and incorporate them into our parameterized designs to improve performance and adaptability. Extensive experiments on a wide range of datasets and mainstream GNN models demonstrate the effectiveness of our design. Overall, GNNAdvisor provides users a handy tool to accelerate GNNs on GPUs systematically and comprehensively.

9 Acknowledgment

We would like to thank our shepherd, Petros Maniatis, and the anonymous OSDI reviewers. This work was supported in part by NSF 1925717. Use was made of computational facilities purchased with funds from the National Science Foundation (OAC-1925717) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 1720256) at UC Santa Barbara.

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] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[3] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web (WWW)*, 2011.

[5] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

[6] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*. IEEE, 2005.

[7] De Cheng, Yihong Gong, Xiaojun Chang, Weiwei Shi, Alexander Hauptmann, and Nanning Zheng. Deep feature learning via structured graph laplacian embedding for person re-identification. *Pattern Recognition*, 2018.

[8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, 1969.

[9] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[10] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint*, 2015.

[11] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019.

[12] Matthias Fey and Jan E. Lenssen. Pytorch extension library of optimized scatter operations, 2019.

[13] Santo Fortunato. Community detection in graphs. *Physics reports*, 2010.

[14] Jaume Gibert, Ernest Valveny, and Horst Bunke. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition*, 2012.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[18] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *The VLDB Endowment*, 2014.

[19] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 2000.

[20] Intel. Xeon sliver 4110. `https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache\-2-10-ghz.html`.

[21] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, Mara J Garzaran, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[22] George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. `http://www.cs.umn.edu/~metis`, 2009.

[23] Riesen Kaspar and Bunke Horst. *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[24] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[25] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[26] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC)*, 2014.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

[28] Jérôme Kunegis and Andreas Lommatzsch. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[30] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 2008.

[31] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[32] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[33] Hang Liu and H Howie Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *USENIX Annual Technical Conference (ATC)*, 2019.

[34] Dijun Luo, Chris Ding, Heng Huang, and Tao Li. Non-negative laplacian embedding. In *Ninth IEEE International Conference on Data Mining (ICDM)*, 2009.

[35] Dijun Luo, Feiping Nie, Heng Huang, and Chris H Ding. Cauchy graph embedding. In *The 28th International Conference on Machine Learning (ICML)*, 2011.

[36] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference (ATC'19)*.

[37] Mark EJ Newman. Spectral methods for community detection and graph partitioning. *Physical Review E*, 2013.

[38] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[39] Nvidia. Cuda sparse matrix library (cusparse). `developer.nvidia.com/cusparse`.

[40] Nvidia. Dgx-1. `https://www.nvidia.com/en-us/data-center/dgx-1/`.

[41] Nvidia. Profiling tools. `docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[42] Nvidia. Quardo p6000 gpu. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p6000-data-sheet-us-nv\-704590-r1.pdf`.

[43] Nvidia. Tesla p100. `https://www.nvidia.com/en-us/data-center/tesla-p100/`.

[44] Nvidia. Tesla v100. `https://www.nvidia.com/en-us/data-center/v100/`.

[45] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, 1999.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward 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 *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

[48] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 2007.

[49] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[50] Alessandra Sala, Haitao Zheng, Ben Y. Zhao, Sabrina Gaito, and Gian Paolo Rossi. Brief announcement: Revisiting the power-law degree distribution for social graph analysis. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2010.

[51] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[53] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[54] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[55] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[56] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, 2018.

[57] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *The 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

A Artifact Appendix

Abstract Summary

GNNAdvisor is an efficient and adaptive runtime system for GNN computing on GPUs. GNNAdvisor consists of two parts. The first part is the host-side CPU program. It is responsible for dataset loading, runtime configuration generation, and invoking the GPU-side program. The second part is the device-side GPU program. It is responsible for the major computation of the GNN model on sparse neighbor-aggregation and dense node-update phase. GNNAdvisor improves the performance of GNN computing with its highly configurable and efficient 2D workload management and specialized memory design. Moreover, the runtime configuration generation on the host-side CPU program makes GNNAdvisor more adaptive towards various kinds of input settings.

Artifact Checklist

- **Link:** `github.com/YukeWang96/OSDI21_AE.git`.
- **Hardware**:
  - Intel CPU x86_64 with host memory >= 32GB. Tested on Intel Xeon Silver 4110 (8-core 16-thread) CPU with 64GB host memory.
  - NVIDIA GPU (arch>=*sm*_60) with devcie memory >= 16GB. Tested on NVIDIA Quadro P6000 (*sm*_61), Tesla V100 (*sm*_70), and RTX3090 (*sm*_86). Note that upon creating this artifact, we mainly evaluate our design on RTX3090. The execution time may be different across different devices but the overall trend of performance (speedup) is similar.
- **OS & Compiler**: Ubuntu 16.04+, GCC 7.5+, CMAKE 3.14+, CUDA 10.2+.

Environment Setup

Step-1: Setup the basic environment. Two options:

- Setup the environment via Docker (**Recommended**).
- Setup via conda and pip.

Details of the above two options can be found in `README.md`.

Step-2: Install GNNAdvisor Pytorch Binding.

- Go to `GNNAdvisor/GNNConv`, then `python setup.py install` to install the `GNNAdvisor` modules.
- Go to `rabbit_module/src`, then `python setup.py install` to install the rabbit reordering modules.

Step-3: Download the graph datasets. Our preprocessed graph datasets in `.npy` format can be downloaded via this link [2] (filename: `osdi-ae-graphs.tar.gz`). Unzip the graph datasets `tar -zxvf osdi-ae-graphs.tar.gz` at the project root directory. Note that node initial embedding is not included, and we generate an all 1s embedding matrix according to users input dimension parameter at the runtime for just performance evaluation.

Experiments

- Running DGL baseline on GNN training (Figure 9).
  - Go to `dgl_baseline/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run DGL and generate `.csv` result for GCN and GIN.
- Running PyG baseline on GNN training (Figure 10).
  - Go to `pyg_baseline/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run PyG and generate `.csv` result for GCN and GIN.
- Running Gunrock for single SpMM (neighbor aggregation) kernel.
  - Go to `Gunrock/` call `./build_spmm.sh`.
  - `./0_bench_Gunrock.py` for profile `spmm`.
- Running GNNAdvisor (Figure 9 and 10).
  - Go to `GNNAdvisor/` directory.
  - `./0_run_gcn.sh` and `./0_run_gin.sh` to run GNNAdvisor and generate `.csv` for GCN/GIN.
- Running some additional studies (Figure 11(a,b,c), and 12(a)). Detailed commands of running all these studies can be found in `README.md`.

Note that accuracy evaluation are omitted for all implementations and each sparse kernels are tested via the unitest.py We focus on the training evaluation of the GNNs, and the reported time per epoch only includes the GNN model forward and backward computation, excluding the data loading and some preprocessing. Since the paper draft submission and the creation of this artifact, DGL has update several of its kernel library (from v0.52 to v0.60). In this comparion we focus on the latest DGL version (v0.60). Based on our profiling on RTX3090 and Quadro P6000, our design would show minor speedup on the simple GCN model (2-layer and 16 hidden dimension), but show more evident speedup on more complicated GIN model (5-layer and 64 hidden dimension), which can still demonstrate the effectiveness of our optimizations. Our observation is that on small Type I graphs, our frameworks achieve significant speedup for both GCN and GIN model on RTX3090 and Quadro P6000. On larger Type II and Type III datasets, our GIN model implementation would show more evident speedups.

[2] https://bit.ly/3ys86a5

Marius: Learning Massive Graph Embeddings on a Single Machine

Jason Mohoney, Roger Waleffe, Henry Xu*, Theodoros Rekatsinas, Shivaram Venkataraman
University of Wisconsin-Madison

Abstract

We propose a new framework for computing the embeddings of large-scale graphs on a single machine. A graph embedding is a fixed length vector representation for each node (and/or edge-type) in a graph and has emerged as the de-facto approach to apply modern machine learning on graphs. We identify that current systems for learning the embeddings of large-scale graphs are bottlenecked by data movement, which results in poor resource utilization and inefficient training. These limitations require state-of-the-art systems to distribute training across multiple machines. We propose Marius, a system for efficient training of graph embeddings that leverages partition caching and buffer-aware data orderings to minimize disk access and interleaves data movement with computation to maximize utilization. We compare Marius against two state-of-the-art industrial systems on a diverse array of benchmarks. We demonstrate that Marius achieves the same level of accuracy but is up to one order of magnitude faster. We also show that Marius can scale training to datasets an order of magnitude beyond a single machine's GPU and CPU memory capacity, enabling training of configurations with more than a billion edges and 550 GB of total parameters on a single machine with 16 GB of GPU memory and 64 GB of CPU memory. Marius is open-sourced at `www.marius-project.org`.

1 Introduction

Graphs are used to represent the relationships between entities in a wide array of domains, ranging from social media and knowledge bases [38, 7] to protein interactions [3]. Moreover, complex graph analysis has been gaining attention in neural network-based machine learning with applications in clustering [30], link prediction [39, 32], and recommendation systems [37]. However, to apply modern machine learning on graphs one needs to convert discrete graph representations (e.g., traditional edge-list or adjacency matrix) to continuous vector representations [10]. To this end, learnable *graph embedding* methods [9, 5, 35] are used to assign each node (and/or edge) in a graph to a specific continuous vector representation such that the structural properties of the graph (e.g., the existence of an edge between two nodes or their proximity due to a short path) can be approximated using these vectors. In general, graph embedding models aim to capture the global structure of a graph and are complementary to graph neural networks (GNNs) [19]. Graph embedding models are primarily used in *link prediction* tasks and can also be used to obtain vector representations that form the input to GNNs.

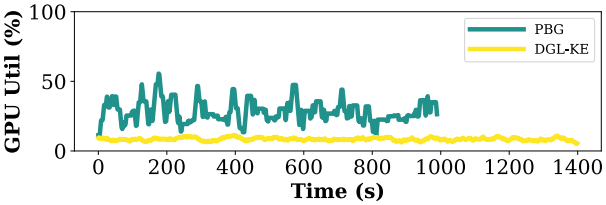


Figure 1: The GPU utilization of DGL-KE and PBG for one training epoch of ComplEx embeddings on the Freebase86m knowledge graph.

However, learning a graph embedding model is a resource intensive process. First, training of graph embedding models can be compute intensive: many graph embedding models assign a high-dimensional continuous vector to each node in a graph [2, 36, 33]. For example, it is common to assign a 400-dimensional continuous vector to each node [18, 40]. Consequently, the computational capabilities of GPUs and optimization methods such as mini-batch Stochastic Gradient Descent (SGD) are needed to accelerate training. Second, graph embedding models are memory intensive: the model from our previous example needs 1600 bytes of storage per node and requires 80 GB (the largest GPU memory) for a modest 50 million node graph. Thus, it is necessary to store the learnable parameters in off-GPU memory. Third, the training of graph embedding models requires optimizing over loss functions that consider the edges of the graph as training examples (e.g., the loss can enforce that the cosine similarity between the vector representations of two connected nodes is close to one, see Section 2.1) making training IO-bound for models that do not fit in GPU memory. This limitation arises due to irregular data accesses imposed by the graph structure. As a result,

*Currently at Maryland, work done while at UW-Madison.

training of large graph embedding models is a non-trivial challenge.

Due to the aforementioned factors, scaling graph embedding training to instances that do not fit in GPU memory introduces costly data movement overheads that can result in poor resource utilization and slow training. In fact, current state-of-the-art systems, including DGL-KE [40] from Amazon, and Pytorch BigGraph (PBG) [18] from Facebook, exhibit poor GPU utilization due to these overheads. Figure 1 shows the GPU utilization during a training epoch when using a single GPU for DGL-KE and PBG. As shown, DGL-KE only utilizes 10% of the GPU, and average utilization for PBG is less than 30%, dropping to zero during data movement.

GPU under-utilization can be attributed to how these systems handle data movement: To support out-of-GPU-memory training, DGL-KE stores parameters in CPU memory and uses synchronous GPU-based training over minibatches. However, the core computation during graph embedding training corresponds to dot-product operations between vectors (see Section 2), and thus, data transfers dominate the end-to-end run time. Moreover, DGL-KE is fundamentally limited by CPU memory capacity. To address this last limitation, PBG uses a different approach for scaling to large graphs. PBG partitions the embedding parameters into disjoint, node-based partitions and stores them on disk where they can be accessed sequentially. Partitions are then loaded from storage and sent to the GPU where training proceeds synchronously. Doing so avoids copying data from the CPU memory for every batch, but results in GPU underutilization when partitions are swapped.

This problem is exacerbated if the storage device has low throughput. Thus, to scale to large instances both systems opt for distributed training over multiple compute nodes, making training resource hungry. However, the problems these systems face are not insurmountable and can be mitigated. *We show that one can train embeddings on billion-edge graphs using just a single machine.*

We introduce a new pipelined training architecture that can interleave data access, transfer, and computation to achieve high utilization. In contrast to prior systems, our architecture results in high GPU utilization throughout training: for the same workload shown in Figure 1, our approach can achieve an average $\sim 70\%$ GPU utilization while achieving the same accuracy (see Section 5).

To achieve this utilization, our architecture introduces asynchronous training of nodes with *bounded staleness*. We combine this with synchronous training for edge embeddings to handle graphs that may contain edges of different types, for example knowledge graphs where an edge may capture different relationships between nodes. Specifically, we consider learning a separate vector representation for each edge-type. For clarity, we refer to edge-type embeddings as *relation embeddings*. This is because updates to the embedding vectors for nodes are sparse and therefore well suited for asynchronous training. However due to the small number of edge-types in real-world graphs $(10,000s)$, updates to relation embedding parameters are dense and require synchronous updates for convergence. We design the pipeline to maintain and update node embedding parameters in CPU memory asynchronously, allowing for staleness, while keeping and updating relation embeddings in GPU memory synchronously. Using this architecture, we can train graph embeddings for a billion-edge Twitter graph *one order of magnitude faster* than state-of-the-art industrial systems for the same level of accuracy: Using a single GPU, our system requires 3.5 hours to learn a graph embedding model over the Twitter graph. For the same setting, DGL-KE requires 35 hours.

To scale training beyond CPU memory, unlike prior out-of-memory graph processing systems [17], we need to iterate over edges while computing on data associated with both endpoints. We propose partitioning the graph and storing embedding parameters on disk. We then design an in-memory *partition buffer* that can hide and reduce IO from swapping of partitions. Partitions are swapped from disk into the partition buffer in CPU memory and then used by the training pipeline. Our partition buffer supports pre-fetching and async writes of partitions to hide waiting for IO, resulting in a reduction of training time by up to $2\times$. Further, we observe that the order in which edge partitions are traversed can impact the number of IOs. Thus, we introduce a *buffer-aware* ordering that uses knowledge of the buffer size and what resides in it to minimize the number of IOs. We show that this ordering achieves IO close to the lower bound and provides benefits when compared to locality-based orderings such as Hilbert ordering [14].

In summary, the key technical contributions of our work are: 1) to show that existing state-of-the-art graph embedding systems are hindered by IO inefficiencies when moving data from disk and from CPU to GPU, 2) to introduce the *Buffer-aware Edge Traversal Algorithm (BETA)*, an algorithm to generate an IO minimizing data ordering for graph learning, 3) to combine the *BETA* ordering with a partition buffer and async IO via pipelining to introduce the first graph learning system that utilizes the full memory hierarchy (Disk-CPU-GPU).

Our design is implemented in Marius, a graph embedding engine that can train billion-edge graphs on a single machine. Using one AWS P3.2xLarge instance, we demonstrate that Marius improves utilization of computational resources and reduces training time by up to an order of magnitude in comparison to existing systems. Marius is $10\times$ faster than DGL-KE on the Twitter graph with 1.46 billion edges, reducing training times from 35

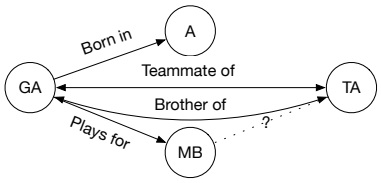


Figure 2: A sample knowledge graph.

hours to 3.5 hours. Marius is 1.5× faster than PBG on the same dataset. On Freebase86m with 86 million nodes and 338 million edges, Marius trains embeddings 3.7× faster than PBG, reducing training times from 7.5 hours to 2 hours. We also show that Marius can scale to configurations where the parameter size exceeds GPU and CPU memory by an order of magnitude, training a configuration with 550 GB of total parameters, 35× and 9× larger than GPU and CPU memory respectively. Finally, we show that despite using a single-GPU on a single-machine, Marius achieves comparable runtime with the multi-GPU configurations of PBG and DGL-KE, thus, providing a cost reduction on cloud resources between 2.9× and 7.5× depending on the configuration.

2 Preliminaries

We first discuss necessary background on graph embeddings and related systems. Then, we review challenges related to optimizing data movement for training large scale graph embedding models. These are the challenges that this work addresses.

2.1 Background and Related Work

Graphs with Multiple Edge Types We focus on graphs with multiple edge types defined as $G = (V, R, E)$ where V is the set of nodes, R is a set of edge-types or *relations*, and E is the set of edges. Each edge $e = (s, r, d) \in E$ is defined as a triplet containing a source node, relation, and destination node. An example of such a graph is a *knowledge graph*, e.g., Freebase [8]. Here, the source node in a triplet defines a subject (an entity), the relation defines a predicate, and the destination node an object (see example in Figure 2). Knowledge graphs are commonly used both in industry and academia to represent real-world facts.

Graph Embedding Models A graph embedding is a fixed length vector representation for each node (and/or edge-type) in a graph. That is, each node and relation is represented by a corresponding d-dimensional vector θ, also known as an *embedding* [10]. There are $d(|V|+|R|)$ total learnable parameters. To learn these vector representations, embedding models rely on *score functions* that capture structural properties of the graph. We denote the score function $f(\theta_s, \theta_r, \theta_d)$ where $\theta_s, \theta_r, \theta_d$ are the vector representations of the elements of a triplet $e = (s, r, d)$. For example, a score function can be the scaled dot product $f(\theta_s, \theta_r, \theta_d) = \theta_s^T \text{diag}(\theta_r)\theta_d$ with the requirement that the parameter vectors are such that $f(\theta_s, \theta_r, \theta_d) \approx 1.0$ if nodes s and d are connected via an edge of type r and $f(\theta_s, \theta_r, \theta_d) \approx 0.0$ otherwise. There are several score functions proposed in the literature ranging from linear score functions [2, 22] to dot products [36, 33, 27] and complex models [11, 10].

Score functions are used to form loss functions for training. The goal is to maximize $f(\theta_s, \theta_r, \theta_d)$ if $e \in E$ and minimize it if $e \notin E$. Triplets that are not present in E are known as *negative edges*. A standard approach [40, 18] is to use the score function $f(\theta_s, \theta_r, \theta_d)$ with a contrastive loss of the form:

$$\mathcal{L} = -\sum_{s,r,d \in E} (f(\mathbf{e}_\theta) - \log(\sum_{s',r',d' \notin E} e^{f(\mathbf{e}_\theta{}')})) \quad (1)$$

where $\mathbf{e}_\theta = (\theta_s, \theta_r, \theta_d)$ and $\mathbf{e}_\theta{}' = (\theta'_s, \theta'_r, \theta'_d)$.

The first summation term is over all true edges in the graph and the second summation is over all negative edges. There are a total of $|V|^2|R| - |E|$ negative edges in a knowledge graph; this makes it computationally infeasible to perform the full summation and thus it is commonly approximated by *negative sampling*, in which a set of negatives edges is generated by taking a (typically uniform) sample of nodes from the graph for each positive edge. With negative sampling the term in the logarithm is approximated as $\sum_{s,r,d' \in N_e} e^{f(\mathbf{e}_\theta{}')}$. Where N_e is the set of negative samples for e.

Graph embeddings are commonly used for *link prediction*, where the similarity of two node vector representations is used to infer the existence of a missing edge in a graph. For example, in the knowledge graph in Figure 2 we can use the vector representation of TA and MB and the relation embedding for *plays-for* to predict the existence of the edge $TA \xRightarrow{\textit{plays-for}} MB$, marked with a questionmark in the figure.

The Need for Scalable Training The largest publicly available multi-relation graphs have hundreds of millions of nodes and tens of thousands of relations [34] (Table 1). Companies have internal datasets which are an order of magnitude larger than these, e.g., Facebook has over 3 billion users [4]. Learning a 400-dimensional embedding for each of the users will require the ability to store and access 5 TB of embedding parameters efficiently, far exceeding the CPU memory capacity of the largest machines. Furthermore, using a larger embedding dimension has been shown to improve overall performance on downstream tasks [31]. For these two reasons it is important that a system for learning graph embeddings can scale beyond the limitations of GPU and CPU memory.

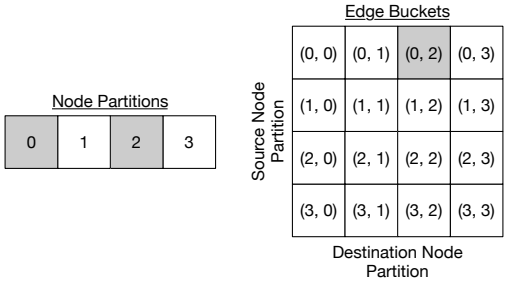


Figure 3: Partitions and edge-buckets with $p = 4$. All edges in edge-bucket (0, 2) have a source node in node-partition 0 and a destination node in node-partition 2.

Scaling Beyond GPU-Memory We review approaches for scaling the training of graph embedding models out of GPU memory. Prior works follow in two categories: 1) methods that use CPU Memory to store embedding parameters, and 2) methods that use block storage and partitioning of the model parameters. We discuss these two approaches in turn.

Following the first approach, systems such as DGL-KE [40] and GraphVite [41], store node embedding parameters in CPU memory and relation embedding parameters in GPU memory. Shown in Algorithm 1, mini-batch training is performed synchronously and batches are formed and transferred on-demand. While synchronous training is beneficial for convergence, it is resource inefficient. The GPU will be idle while waiting for the batch to be formed and transferred; furthermore, gradient updates also need to be transferred from the GPU to CPU memory and applied to the embedding table, adding additional delays. The effect of this approach on utilization can be seen in Figure 1, where DGL-KE on average only utilizes about 10% of the GPU. This approach is also fundamentally limited by the size of the CPU memory, preventing the training of large graph embedding models.

Algorithm 1: Synchronous Embedding Training

for i *in range*$(num\_batches)$ **do**
1. $\mathbf{B_i} = \text{getBatchEdges}(i)$;
2. $\mathbf{\Theta_n} = \text{getCpuParameters}(\mathbf{B_i})$;
3. $\text{transferBatchToDevice}(B_i, \mathbf{\Theta_n})$;
4. $\mathbf{\Theta_r} = \text{getGpuParameters}(\mathbf{B_i})$;
5. $\mathbf{B}_\theta = \text{formBatch}(\mathbf{B_i}, \mathbf{\Theta_n}, \mathbf{\Theta_r})$;
6. $\mathbf{G_n}, \mathbf{G_r} = \text{computeGradients}(\mathbf{B}_\theta)$;
7. $\text{updateGpuParameters}(\mathbf{B_i}, \mathbf{G_r})$;
8. $\text{transferGradientsToHost}(\mathbf{G_n})$;
9. $\text{updateCpuParameters}(\mathbf{B_i}, \mathbf{G_n})$;

The second approach is adopted by PyTorch BigGraph (PBG) [18]. PBG uses uniform partitioning to split up node embedding parameters into p disjoint partitions and stores them on a block storage device (see example in Figure 3). Edges are then grouped according the partition of their source and destination nodes into p^2 edge buckets, where all edges in edge bucket (i, j) have a source node which has an embedding in the i-th partition, and the destination node which has an embedding in the j-th partition. A single epoch of training requires performing mini-batch training over all edge buckets while swapping corresponding pairs of node embedding partitions into memory for each edge bucket. This approach enables scaling beyond CPU memory capacity.

The major drawback of partitioning is that partition swaps are expensive and lead to the GPU being idle while a swap is happening. In fact, utilization goes towards zero during swaps as shown in Figure 1. We find that PBG yields an average GPU utilization of 28%. To best utilize resources, a system using partitioning to scale beyond the memory size of a machine, will need to mitigate overheads that arise from swapping partitions.

2.2 Data Movement Challenges

We discuss how to optimize data movement and related challenges that Marius' architecture addresses; we discuss the architecture in detail in Sections 3 and 4.

Traditional Optimizations for Data Movement Pipelining is a common approach used in a number of system designs to overlap computation with data movement, thereby improving utilization [13, 25, 28]. Using an image classifier as an example, a simple pipeline will consist of multiple worker threads that pre-process training images in parallel, forming batches and transferring them to the GPU. Once on the GPU, batches of training data are pushed onto a queue, with a training process constantly polling the queue for new batches. By keeping the queue populated with new batches, the GPU will be well utilized.

In IO-bound applications, buffer management can also be used to prevent unnecessary IO by caching data in memory. Buffer management is well studied in the area of databases and operating systems and has been applied to a myriad of applications and workloads [29, 12]. When using a buffer, the order in which data is accessed and swapped impacts end-to-end performance. When the data access pattern exhibits good locality, buffer managers typically yield good performance. Additionally, if the ordering is known ahead of time the buffer manager may prefetch data items and use Belady's optimal cache replacement algorithm to minimize IO [1].

In graph processing, locality-aware data layouts of graph edges have been shown to improve locality of accesses and performance of common graph algorithms such as PageRank [24]. One such data layout, utilizes Hilbert space filling curves to define an ordering over the adjacency matrix of the graph. The ordering produced is a 1D index that preserves the locality of the 2D adja-

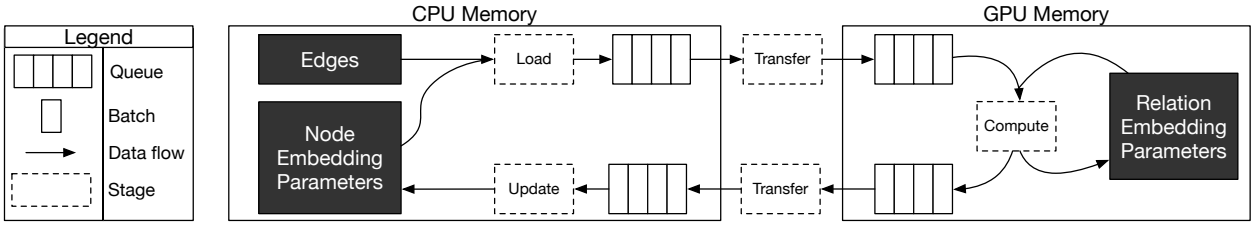


Figure 4: Marius training pipeline.

cency matrix. Storing and accessing edges according to this index improves OS cache hit rates [24, 23].

Challenges for Graph Embeddings For large graphs and embedding sizes, graph embedding models can be multiple orders of magnitude larger than the GPU's memory capacity, a key difference from deep neural network models that typically fit in a single GPU. To design a pipeline for graph embedding training, not only will training data (formed by considering edges) have to be piped to the GPU but also the corresponding model parameters (the node and relation embeddings of the endpoints and the type of each edge). Furthermore, model updates need to be piped back from the GPU and applied to the underlying storage. By pipelining model parameters and updates, we introduce the possibility of stale parameters, which must mitigated (see Section 3).

Buffer management techniques paired with data orderings can be used to buffer partitions in CPU memory to reduce IO from disk. However, we find that prior locality-aware data orderings such as space-filling curves fall short and still result in IO bound training due to a non-optimal amount of swaps (Sections 4.1 and 5.3). To address this challenge we propose a buffer-aware data ordering which results in a near-optimal number of swaps, referred to as *BETA ordering*, in Section 4.

3 Pipelined Training Architecture

We review Marius' pipelined architecture for training graph embedding models. We first discuss the overall design, then the details of each stage, and finally discuss how staleness arises due to interleaving computation with data movement and how we can mitigate it.

Pipeline Design Our architecture follows Algorithm 1 and divides its steps into a five-stage pipeline with queues separating each stage (Figure 4). Four stages are responsible for data movement operations, and one for model computation and in-GPU parameter updates. The four data movement stages have a configurable number of worker threads, while the model computation stage only uses a single worker to ensure that relation embeddings stored on the GPU are updated synchronously.

We now describe the different stages of the pipeline and draw connections to the steps in Algorithm 1:

Stage 1: Load This stage is responsible for loading the edges (i.e., entries that correspond to a pair of node-ids and the type of edge that connects them) and the corresponding node embedding vectors that form a batch of inputs used for training. The edge payload constructed in this stage includes the true edges appearing in the graph and a uniform sample of negative edges (i.e., fake edges) necessary to form the loss function in Equation 1 (Lines 1-2 in Algorithm 1).

Stage 2: Transfer The input to this stage consists of the edges (node-id and edge-type triples) and the node embeddings from the previous stage. Worker threads in this stage asynchronously transfer data from CPU to GPU using `cudaMemCpy` (Line 3 in Algorithm 1).

Stage 3: Compute The compute stage is the only stage that does not involve data movement. This stage takes place on GPU where the payload of edges and node embeddings created in Stage 1 is combined with relation embedding vectors (corresponding to the edge-type associated with each entry) to form a full batch. The worker thread then computes model updates and applies updates to relation embeddings stored in the GPU. The updates to node embeddings (i.e., the scaled gradients that need to be added to the previous version of the node embedding parameters) are placed on the output queue to be transferred from GPU memory (Lines 4-7 in Algorithm 1).

Stage 4: Transfer The node embedding updates are transferred back to the CPU. We use similar mechanisms as in Stage 2 (Line 8 in Algorithm 1).

Stage 5: Update. The final stage in our pipeline applies node embedding updates to stored parameters in CPU memory (Line 9 in Algorithm 1).

This hybrid-memory architecture allows us to execute sparse parameter updates asynchronously (i.e., the node embedding parameter updates) and dense updates (i.e., the relation embedding parameter updates) synchronously, and optimize resource utilization as we show experimentally in Section 5.

Bounded Staleness The main challenge with using a pipelined design as described above, is that it introduces *staleness* due to asynchronous processing [15]. To illustrate this, consider a batch entering the pipeline (Stage 1) with the embedding for node A. Once this batch reaches

the GPU (Stage 3), the gradients for the embedding for A will be computed. While the gradient is being computed, consider another batch that also contains the embedding for node A entering the pipeline (Stage 1). Now, while the updates from the first batch are being transferred back to the CPU and applied to parameter storage, the second batch has already entered the pipeline, and thus it contains a stale version of the embedding for node A.

To limit this staleness, we bound the number of batches in the pipeline at any given time. For example, if the bound is 4, embeddings in the pipeline will be at worst 4 updates behind. However, due to the sparsity of node embedding updates, it is unlikely a node embedding will even become stale. To give a realistic example, take the Freebase86m graph which has 86 million nodes. A typical batch size and staleness bound for this benchmark is 10,000 and 16 respectively. Each batch of 10,000 edges will have at most 20,000 node embeddings and given this staleness bound there can be at most 320,000 node embeddings in the pipeline at any given time, which is just about .4% of all node embeddings. Even with this worst case, only a very small fraction of node embeddings will be operated on at a given time. The same property does not hold for relation embeddings since there are very few of them (15K in Freebase86m), hence our design decision to keep relation embeddings in GPU memory and update them synchronously, bypasses the issue of stale relation embeddings. We study the effect of staleness and Marius' performance as we vary the bound in Section 5.5.

4 Out-of-memory Training

As described in Section 2.1, to learn embedding models for graphs that do not fit in CPU memory, existing systems partition the graph into non-overlapping blocks. They correspondingly partition the parameters as well so that they can be loaded sequentially for processing. However as IO from disk can be slow (e.g, a partition can be around 10s of GB in size), it is desirable to hide the IO wait times and minimize the number of swaps from disk to memory. In this section, we describe how we can effectively hide IO wait time by integrating our training pipeline with a *partition buffer* that constitutes an in-memory buffer of partitions. We also describe how we can minimize the number of swaps from disk to memory by developing a new ordering for traversing graph data.

Partition-based training Consider a graph that is partitioned into p^2 edge buckets corresponding to p node-partitions. Training one epoch requires iterating over all p^2 edge buckets, where each edge in a given bucket (i, j), will have a source node in partition i and destination node in partition j.

When processing an edge bucket (i, j), node partition i and node partition j must be present in the CPU partition buffer in order for learning to proceed using the pipelined training architecture (see Section 3). If either one is not present, it must be loaded from disk and *swapped* into the buffer, replacing an already present partition if the buffer is full. Partition-based training is described in Algorithm 2.

Algorithm 2: Training Using a Partition Buffer

```
Buffer = {};
for k in range(p^2) do
    E_ij, i, j = getEdgeBucket(Ordering[k]);
    if i not in Buffer then
        if Buffer.size() == c then
            Buffer.evictFurthest(Ordering, k);
        Buffer.admit(i);
    if j not in Buffer then
        if Buffer.size() == c then
            Buffer.evictFurthest(Ordering, k);
        Buffer.admit(j);
    Θ_i = Buffer.get(i); // Source Node Partition
    Θ_j = Buffer.get(j); // Destination Node Partition
    trainEdgeBucket(E_ij, Θ_i, Θ_j);
```

Given a partitioned graph, there are a number of edge bucket orderings that can be used for traversal. To minimize the number of times partitions need to be loaded from disk, *we seek an ordering over edge buckets which minimizes the number of required partition swaps*.

We note that once an edge bucket ordering has been selected, we can further mitigate IO overhead by 1) prefetching to load node partitions as they are needed in the near future and 2) using the optimal buffer eviction policy which removes partitions used farthest in the future.

We next discuss the problem of determining an optimal ordering over edge buckets and describe the *BETA ordering*, a new ordering scheme that achieves near-optimal number of partition swaps.

4.1 Edge Bucket Orderings

We develop an edge bucket ordering scheme that minimizes the number of swaps. First, we derive a lower bound on the number of swaps necessary to complete one training epoch for a buffer of size c and p ($p >= c$) partitions. To derive the lower bound, *we view an edge bucket ordering as a sequence of partition buffers* over time, where each item in the sequence describes what node partitions are in the buffer at that point. Each successive buffer differs by one swapped partition.

Given such a sequence, an edge bucket ordering can be constructed by processing edge bucket (i, j) when partitions i and j are in the buffer. For simplicity, we can do this the first time i and j appear together. Note that 1)

Algorithm 3: *BETA* Ordering Buffer Sequence

```
1  PartitionBufferSequence = {};
2  CurrentBuffer = [0 ... c − 1];
3  OnDisk = [c ... p − 1];
4  PartitionBufferSequence.append(CurrentBuffer);
5  while OnDisk.size() > 0 do
6      for i in range(OnDisk.size()) do
7          swap(CurrentBuffer[−1], OnDisk[i]);
8          PartitionBufferSequence.append(CurrentBuffer);
9      n = 0;
10     for i in range(c − 1) do
11         if i ≥ OnDisk.size() then
12             break;
13         n = n + 1;
14         CurrentBuffer[i] = OnDisk[i];
15         PartitionBufferSequence.append(CurrentBuffer);
16     OnDisk = OnDisk[n : end];
17 return PartitionBufferSequence;
```

Algorithm 4: Buffer Seq. to Edge Bucket Order

```
1  EdgeBuckets = {};
2  SeenPairs = zeros(p, p);
3  for Buffer in PartitionBufferSequence do
4      NewEdgeBuckets = {};
5      for i in Buffer do
6          for j in Buffer do
7              if SeenPairs[i, j] == 0 then
8                  SeenPairs[i, j] = 1;
9                  NewEdgeBuckets.append((i, j));
10     shuffle(NewEdgeBuckets);
11     EdgeBuckets.append(NewEdgeBuckets);
12 return EdgeBuckets;
```

i and j must appear together at least once otherwise no ordering over all edge buckets can be constructed, 2) self-edge buckets (i.e. (i, i)) can also be added to the ordering the first time i appears in the buffer, and 3) there are many edge bucket orderings with the same sequence of partition buffers (depending on the order in which edge buckets in a particular buffer are processed). Viewed in this light, *we seek the shortest (min. swaps) buffer sequence where all node partition pairs appear together in the buffer at least once*.

Lower bound We assume that initializing the first full buffer does not count as part of the total number of swaps as all orderings must incur this cost. Thus, there are $\frac{p(p-1)}{2}$ (the total number of pairs) minus $\frac{c(c-1)}{2}$ (the number of pairs we get in the first buffer) remaining partition pairs that must appear together in the partition buffer. On any given swap, the most new pairs we can cover is if the partition entering the buffer has not been paired with anything already in the buffer (everything in the buffer has already been paired with everything else in the buffer). Thus, for each swap the best we can hope for is to get $c - 1$ pairs we have not already seen. With this in mind a lower bound on the minimum number of swaps required is:

$$\left\lceil \frac{\frac{p(p-1)}{2} - \frac{c(c-1)}{2}}{c-1} \right\rceil \tag{2}$$

We use this lower bound to evaluate the performance of different edge bucket orderings in the next section. We experimentally show that the new ordering strategy we propose is nearly optimal with respect to this bound.

***BETA* ordering** We describe the *Buffer-aware Edge Traversal Algorithm (BETA)*, an algorithm to compute the edge bucket ordering that achieves close to optimal number of partition swaps and improves upon locality-aware orderings such as Hilbert space-filling curves [14].

Algorithm 3 describes how the *BETA* ordering of partition buffers is generated. Consider a partition buffer that was initialized with the first c node-partitions in the graph (Line 2 in Algorithm 3). The remaining $p-c$ node-partitions start on disk (Line 3 in Algorithm 3). To generate the partition buffer sequence we then proceed as follows: First we fix the leading $c - 1$ node-partitions in the buffer and swap each of the outstanding partitions into the final buffer spot, one at a time (Line 6-8 in Algorithm 3). Each swap creates a new partition buffer in the sequence. Once this is complete, the fixed $c - 1$ partitions have been paired in the buffer with all other node-partitions and are therefore no longer needed. We refresh our buffer by replacing the finished $c - 1$ partitions with new node-partitions from the unfinished set on disk (Line 10-15 in Algorithm 3). The incoming partitions can then be deleted from the on disk set (Line 16 in Algorithm 3) since they are now in the buffer. As before, each swap results in a partition buffer added to the sequence. We repeat this process until there are no remaining unfinished node-partitions (Line 5 and 11-12 in Algorithm 3). As described at the beginning of Section 4.1 and in more detail in Algorithm 4, the partition buffer sequence can be easily converted to the final edge bucket ordering. We show an example *BETA* ordering in Figure 5.

We observe that our *BETA* ordering has a number of useful properties that make it advantageous to implement in practice. Since all partitions are symmetrically processed we do not need to track any extra state or use any priority mechanisms. Further, for every disk IO (swap) with a fixed set of $c-1$ partitions (Line 7 in Algorithm 3), the incoming node-partition has yet to be paired with any other partition in the buffer. This means we can process $c-1$ edge buckets before performing another swap—the most possible (excluding self edge buckets)—allowing us to hide IO operations behind longer compute times. The only bottleneck arises when the fixed $c-1$ partitions

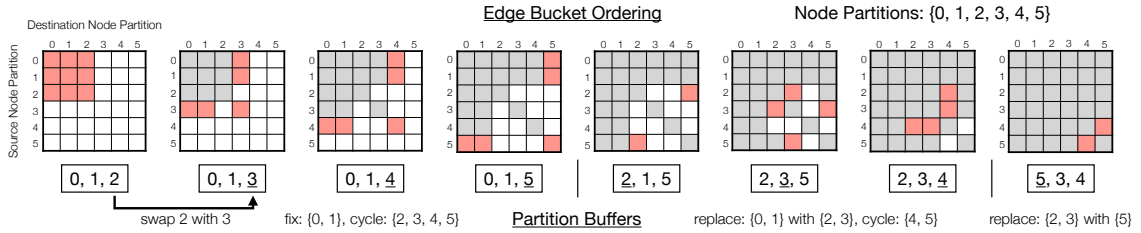
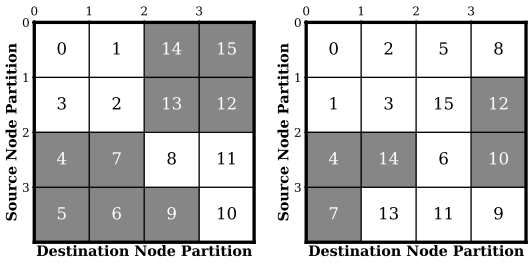


Figure 5: Example *BETA* ordering for $p = 6$ and $c = 3$. The sequence of partition buffers corresponds to first fixing $\{0, 1\}$, then replacing $\{0, 1\}$ with $\{2, 3\}$, fixing $\{2, 3\}$, and finally replacing $\{2, 3\}$ with $\{5\}$. Each successive buffer differs by one swap. A corresponding edge bucket ordering is shown above the buffers. For each partition buffer in the sequence, all previously unprocessed edge buckets which have their source and destination node partitions in the buffer are added to the ordering (red edge buckets). For each buffer, these edge buckets can be added in any order.



(a) Hilbert Ordering (b) *BETA* Ordering

Figure 6: Hilbert and *BETA* edge bucket orderings. Numbers indicate the order in which the bucket is processed. Gray cells indicate misses to the buffer.

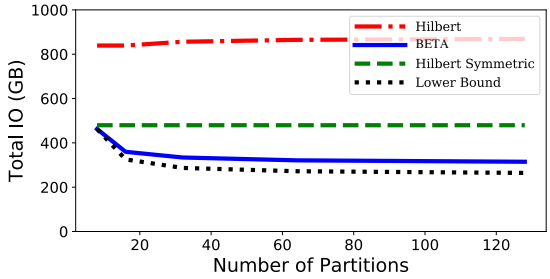


Figure 7: Simulated total IO performed during a single epoch of training Freebase86m with $d = 100$.

are replaced, but this only happens at most $\left\lfloor \frac{p-c}{c-1} \right\rfloor + 1$ times in one epoch. Additionally, the *BETA* ordering can be randomized to create different graph traversals by shuffling which partitions start in the buffer, by permuting the buffer and/or on disk set before Line 6 in Algorithm 3, or by permuting the on disk set before Line 10 in Algorithm 3.

Finally, we analyze the number of swaps generated by the *BETA* ordering: given p partitions and a buffer of size c the number of swaps is

$$(p-c) + (x+1)\left[(p-c) - \frac{1}{2}x(c-1)\right] \quad \text{where } x = \left\lfloor \frac{p-c}{c-1} \right\rfloor. \tag{3}$$

Comparison with Hilbert, lower bound We compare the number of IO operations incurred by the *BETA* ordering with space-filling curve based orderings, and the analytical lower bound. Space filling curve orderings like Hilbert [14] attempt to define a graph traversal that preserves 2D locality over the $n \times n$ matrix of edge buckets. We also compare to a second version of the Hilbert ordering, termed Hilbert Symmetric, which modifies the former by processing edge buckets (i, j) and (j, i) successively. A key advantage of the *BETA* ordering when compared to these methods is that it is buffer-aware, i.e., the algorithm knows the buffer size and specifically aims to minimize partition swaps. In contrast, space-filling curve based orderings are unaware of this information, aiming instead for locality.

We illustrate how the *BETA* ordering compares to a Hilbert space-filling curve on a small $p = 4, c = 2$ case in Figure 6. We see that while the Hilbert ordering has nine buffer misses the *BETA* ordering only has five misses. We also performed simulations to compare each method. Figure 7 shows the number of IO accesses when varying p and using a buffer with size $\frac{p}{4}$ for the *BETA*, Hilbert, and Hilbert Symmetric orderings, together with the lower bound. The *BETA* ordering yields nearly optimal performance across partition configurations and requires significantly less IO than the other methods.

We leave an investigation of a provably-optimal ordering for future work. Our initial studies have shown that there exist cases of p, c where no valid ordering can match the lower bound as well as cases where an ordering which requires slightly fewer swaps than the *BETA* ordering does exist. Thus, the optimal algorithm requires IO-swaps somewhere between the lower bound and the *BETA* ordering in Figure 7.

4.2 Partition Buffer

We next describe mechanisms that we use in the partition buffer to further minimize IO overhead. The par-

tition buffer is a fixed sized memory region that has capacity to store c embedding partitions in memory. We *co-design* the buffer replacement policies with the *BETA* ordering described above. Co-designing the edge traversal with buffer replacement policy means that we have knowledge about which partitions will be accessed in the future. This allows the buffer to use the optimal replacement policy: *evict the partition that will be used furthest in the future* [1]. Given this policy we also design a prefetching mechanism that can minimize the amount of time spent waiting for partitions to swap. Again, based on knowing the order in which partitions are used, we use a prefetching thread that reads the next partition in the background. Correspondingly when a partition needs to be evicted from memory, we perform asynchronous writes using a background writer thread.

5 Evaluation

We evaluate Marius on standard benchmarks using a single AWS P3.2xLarge instance and compare against SoTA graph embedding systems. We show that:

(1) Due to optimized resource utilization, Marius yields up to 10× faster training in comparison to SoTA systems and cost reductions on cloud resources between 2.9× and 7.5× depending on the configuration.

(2) The *BETA* ordering reduces IO required by up to 2× when compared to other locality-based graph orderings, thus alleviating the IO bottleneck during training.

(3) Marius is able to scale to graph embedding models that rely on increased vector dimensions to achieve higher accuracy. Due to the increased vector dimensions these models exceed CPU memory size. For instance, we show that Marius can learn an embedding model using 800-d vector representations on a graph with 86M nodes on a single machine. In this configuration there are 550 GB of total parameters and optimizer state, which is 35× GPU memory size and 9× CPU memory size.

5.1 Setup

Implementation Marius is implemented in about 10,000 lines of C++. We use LibTorch [21], which is the C++ API of PyTorch, as the underlying tensor engine. LibTorch provides access to the wide-ranging functionality of PyTorch, making it easy to extend Marius to support more complex embedding models. We also implement an abstracted storage API, which allows for embedding parameters to be stored and accessed across a variety of backends under one unified API. This allows us to easily switch between storage backends, say from using a CPU memory-based backend to a disk-based backend.

Hardware Setup Single machine experiments are run on a single AWS P3.2xLarge instance which has: 1 Telsa V100 GPU with 16 GB of memory, 8 vCPUs with 64 GB of memory, and an attached EBS volume with 400 MBps of read and write bandwidth. For multi-GPU experiments, we use the AWS P3.16xLarge instance which has 8 Tesla V100 GPUs with 16 GB of memory each, 64 vCPUs, and 524 GB of CPU memory. For distributed multi-core experiments we use 4 c5a.8xLarge instances with 32 vCPUs and 69 GB of CPU memory each. DGL-KE ran out of memory when using a single GPU with the Twitter and Freebase86m datasets. For these cases, we use a larger machine with 1 Telsa V100 GPU with 32 GB of memory and 200 CPUs with 500 GB of memory.

Datasets For our evaluation, we use standard benchmark datasets that include social networks (Twitter [16], Livejournal [20]) and knowledge graphs (FB15k and Freebase86m [18, 40] derived from Freebase [8]). A summary of the dataset properties is shown in Table 1. FB15k uses an 80/10/10 train, validation and test split. All others use a 90/5/5 split.

Embedding Models On FB15k, we use ComplEx [33] and DistMult [36]. On LiveJournal and Twitter we use *Dot* [19], which is a dot product between the node embeddings of an edge. On Freebase86m we use ComplEx embeddings. We chose these models to match the evaluation of Zheng et al. [40] and Lerer et al. [18].

Hyperparameters To ensure fair comparisons, we use the same hyperparameters across each system instead of tuning separately. Hyperparameter values for each configuration were chosen based on those used in the evaluation of DGL-KE and PBG and are shown in Table 1. All systems use the Adagrad optimizer [6] for training, which empirically yields much higher-quality embeddings over SGD. One drawback of using Adagrad is that it effectively requires storing a learning rate per parameter, doubling the overall memory footprint of the embeddings during training. For Marius, we use a staleness bound of 16 for all cases which utilize the pipeline.

Evaluation Task and Metrics We evaluate the quality of the embeddings using the link prediction task. Link prediction is a commonly used evaluation task in which embeddings are used to predict if a given edge is present in the graph. Link prediction metrics reported are Mean Reciprocal Rank (MRR) and Hits@k, which are derived from the rank of the score of each candidate edge, where the scores are produced from the embedding score function f. For a given candidate edge i, it has a rank r_i which denotes the position of the score of the candidate edge in descending sorted array S_i, where S_i contains the score of the candidate edge and the scores of a set of negative samples. Given this, the MRR and Hits@k can be computed from a set of candidate edges C as follows: $\frac{1}{|C|}\sum_{i\in C}\frac{1}{r_i}$ and $\frac{1}{|C|}\sum_{i\in C}\mathbb{1}_{r_i<=k}$ respectively.

Name	Type	|E|	|V|	|R|	Size	Hyperparameters
FB15k	KG	592k	15k	1.3k	52 MB	$d = 400$, $lr = .1$, $b = 10^4$, $n_t = 10^3$, $\alpha_{n_t} = .5$, FilteredMRR
LiveJournal	Social	68M	4.8M	-	1.9 GB	$d = 100$, $lr = .1$, $b = 5 \times 10^4$, $n_t = 10^3$, $\alpha_{n_t} = .5$, $n_e = 10^4$, $\alpha_{n_e} = 0$
Twitter	Social	1.46B	41.6M	-	33.2 GB	$d = 100$, $lr = .1$, $b = 5 \times 10^4$, $n_t = 10^3$, $\alpha_{n_t} = .5$, $n_e = 10^3$, $\alpha_{n_e} = .5$
Freebase86m	KG	338M	86.1M	14.8K	68.8 GB	$d = 100$, $lr = .1$, $b = 5 \times 10^4$, $n_t = 10^3$, $\alpha_{n_t} = .5$, $n_e = 10^3$, $\alpha_{n_e} = .5$

Table 1: Datasets used for evaluation. The size column indicates total size of embedding parameters with the embedding dimension d, including the Adagrad optimizer state. lr: learning rate, b: batch size, n_t: training negatives, α_{n_t}: train degree-based negatives fraction, n_e: evaluation negatives, α_{n_e}: eval degree-based negatives fraction.

Metrics can be filtered or unfiltered. Filtered evaluation involves comparing candidate edges with $|N|$ negative samples, produced using all of the nodes in the graph. Some of the produced negative samples will be false negatives, which will not be used in filtered evaluation. Because all nodes in the graph are used, filtered evaluation is expensive for large graphs. Unfiltered evaluation samples n_e nodes from the graph, with a fraction $\alpha_{n_e} n_e$ by degree and $(1 - \alpha_{n_e})n_e$ uniformly. False negatives are not removed in unfiltered evaluation, but will not be common if $n_e << |V|$. Unfiltered evaluation is much less expensive and is well suited for large scale graphs. We use filtered metrics only on FB15k and unfiltered metrics elsewhere. The same evaluation approach is adopted by prior systems [18].

5.2 Comparison with Existing Systems

To demonstrate that Marius utilizes resources better than current SoTA systems leading to faster training, we compare Marius with PBG and DGL-KE on four benchmark datasets. We do not compare with GraphVite since it is significantly slower than DGL-KE as reported in Zheng et al. [40]. FB15k and LiveJournal fit in the machine's GPU memory and therefore do not have data movement overheads. Twitter exceeds GPU memory which introduces data movement overheads from storing parameters off-GPU. Freebase86m exceeds the CPU memory of the machine, which prevents DGL-KE from training these embeddings on a single P3.2xLarge instance, therefore we only compare against PBG.

FB15k In this experiment, we compare Marius with PBG and DGL-KE on FB15k to show that Marius achieves similar embedding quality as the other systems on a common benchmark. We measure the FilteredMRR, Hits@k, and runtime of the systems when training ComplEx and DistMult embeddings with $d = 400$ to peak accuracy, averaged over five separate runs. Results are shown in Table 2. It should be noted that all parameters and training data fit in GPU memory for this dataset. We find that Marius achieves near identical metrics as PBG when learning the same embeddings, this is expected as both systems have similar implementations for sampling edges and negative samples. DGL-KE on the other hand only achieves a similar FilteredMRR. DGL-KE has implementation differences for initialization and sampling which likely account for the difference in metrics. While Marius is not designed for small knowledge graphs, we can see that it performs comparably to SoTA systems, achieving similar embedding quality in lesser time.

System	Model	Filtered MRR	Hits @1	Hits @10	Time (s)
DGL-KE	ComplEx	.795	.766	.848	35.6s ± .69
PBG	ComplEx	.795	.736	.888	40.3s ± .1
Marius	ComplEx	.795	.736	.888	**27.7s** ± .12
DGL-KE	DistMult	.792	.766	.848	32.8s ± .88
PBG	DistMult	.790	.728	.888	46.2s ± .46
Marius	DistMult	.790	.727	.889	**28.7s** ± .15

Table 2: FB15k Results. All systems reach peak accuracy at about the same number of epochs with 30 and 35 epochs for ComplEx and DistMult respectively.

System	Model	MRR	Hits @1	Hits @10	Time (min)
DGL-KE	Dot	.753	.675	.876	25.7m +-.17
PBG	Dot	.751	.672	.873	23.6m +-.17
Marius	Dot	.750	.672	.872	**12.5m** +-.01

Table 3: LiveJournal results after 25 epochs.

LiveJournal To show that the systems are comparable on social graphs, we compare the quality of 100-dimensional embeddings learned by the three systems using a dot product score function. While Livejournal is two orders of magnitude larger than FB15k, all parameters still fit in GPU memory with a total of 2 GB. As before, we measure MRR, hits@k, and runtime, averaging over three runs; but we use unfiltered MRR instead of FilteredMRR. We do so because FilteredMRR is computationally expensive to evaluate on larger graphs (Section 5.1). Instead of using all nodes in the graph to construct negative samples, we sample 10,000 nodes uniformly for evaluation, as done in Lerer et al. [18]. Results are shown in Table 3. We see that all three systems achieve near identical metrics for this dataset. There are slight differences in runtimes that can be attributed to implementation differences. PBG checkpoints parameters after each epoch, while this is optional in Marius and DGL-KE. Without checkpointing, PBG would likely achieve similar runtimes to DGL-KE and Marius. Overall, we find that Marius performs as well or better than SoTA systems on this social graph benchmark.

Twitter We now move on to evaluating Marius on large-scale graphs for which embedding parameters do not fit

System	Model	MRR	Hits		Time
			@1	@10	
PBG	Dot	.313	.239	.451	5h15m
DGL-KE	Dot	.220	.153	.385	35h3m
Marius	Dot	.310	.236	.445	**3h28m**

Table 4: Twitter results after training for 10 epochs.

System	Model	MRR	Hits		Time
			@1	@10	
PBG	ComplEx	.725	.692	.789	7h27m
Marius	ComplEx	.726	.694	.786	**2h1m**

Table 5: Freebase86m results with embedding size 100 after training for 10 epochs.

in GPU memory. The Twitter follower network has approximately 1.4 billion edges and 41 million nodes. We train 100-dimensional embeddings on each system using a dot product score function. We report results for one run for each system since we observed that training times and MRR are stable between runs. In total, there are 16 GB of embedding parameters with another 16 GB of optimizer state, since all systems use the Adagrad optimizer, as discussed above. To construct negatives for evaluation we use the approach from Zheng et al. [40], where 1,000 nodes are sampled uniformly from the graph, and 1,000 nodes are sampled by degree.

Unlike the previous two datasets, each system uses a different methodology for training embeddings beyond GPU memory sizes. DGL-KE uses the approach described in Algorithm 1, storing parameters in CPU memory and processing batches synchronously while waiting for data movement. PBG does not utilize CPU memory and instead uses the partitioning approach with 16 partitions. Marius stores parameters in CPU memory, and utilizes its pipelined training architecture to overlap data movement with computation.

We compare the peak embedding quality learned by each of the systems after ten epochs of training in Table 4. We find that Marius is able to train similar quality embeddings faster than the other systems, 10× faster than DGL-KE and 1.5× faster than PBG. DGL-KE's long training times can be attributed to data movement wait times inherent in synchronous processing. PBG on the other hand, only pays a data movement cost when swapping partitions. PBG achieves comparable runtimes because this dataset has a large amount of edges relative to the total number of parameters, meaning that computation times dominate partition swapping times.

Turning our attention to the embedding quality, we find that Marius learns embedding of comparable quality to the next-best system: Marius yields an MRR of 0.310 versus 0.313 PBG. On the other hand, DGL-KE only achieves an MRR of 0.220. We attribute this gap in quality to to implementation differences between the systems (all use the same hyperparameters for training).

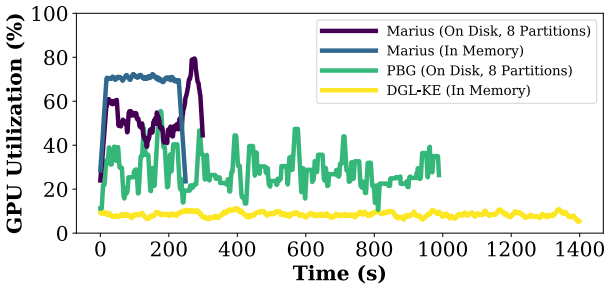


Figure 8: GPU utilization of Marius, DGL-KE and PBG during a single epoch of training $d = 50$ embeddings on Freebase86m. Utilization is smoothed over 25 seconds.

Freebase86m We now evaluate Marius on a large-scale knowledge graph for which embedding parameters do not fit in CPU or GPU memory. We train 100-dimensional ComplEx embeddings for each system. In total, there are about 32 GB of embedding parameters with another 32 GB of Adagrad optimizer state. We do not evaluate DGL-KE on this dataset since it is unable to process this configuration on a single P3.2xLarge instance. For evaluation, we sample 1000 nodes uniformly and 1000 nodes based on degree as negative samples.

We compare the peak embedding quality of Marius and PBG where both systems are trained to 10 epochs in Table 5. Both systems use 16 partitions for training and in Marius we vary the number of partitions we hold in the CPU memory buffer. We find that Marius is able to train to peak embedding quality 3.7× faster when the buffer has a capacity of 8 partitions while reaching a similar accuracy. The runtime difference between the two systems can be attributed to the fewer number of partition swaps Marius performs and the ability to prefetch partitions.

Utilization We include a comparison of GPU utilization during a single epoch of training $d = 50$ embeddings on Freebase86m. Figure 8 shows the utilization of two configurations of Marius compared to DGL-KE and PBG. One configuration of Marius stores embeddings in CPU memory while the other uses eight partitions on disk with four partitions buffered in CPU memory. We see that Marius is able to utilize the GPU 8× more than DGL-KE when training in memory and about 6× more when using the partition buffer. Compared to PBG, our partition buffer design leads to nearly 2× GPU utilization with fewer drops in utilization when waiting for partition swaps. While better than the baseline systems, Marius still doesn't achieve 100% GPU utilization for this configuration. When profiling Marius with NVIDIA's nvprof, we found that all GPU operations were executed on the default CUDA stream, which is the default behavior of PyTorch. We plan to improve our implementation to leverage multiple CUDA streams thereby enabling GPU data transfer and compute to run in parallel, thereby improving GPU utilization. We have also found

System	Deployment	Epoch Time (s)	Per Epoch Cost ($)
Marius	1-GPU	288	**.248**
DGL-KE	2-GPUs	761	1.29
DGL-KE	4-GPUs	426	1.45
DGL-KE	8-GPUs	220	1.50
DGL-KE	Distributed	1237	1.69
PBG	1-GPU	1005	.85
PBG	2-GPUs	430	.73
PBG	4-GPUs	330	1.12
PBG	8-GPUs	273	1.86
PBG	Distributed	1199	1.64

Table 6: Cost comparisons with d=50 on Freebase86m.

System	Deployment	Epoch Time (s)	Per Epoch Cost ($)
Marius	1-GPU	727	**.61**
DGL-KE	2-GPUs	1068	1.81
DGL-KE	4-GPUs	542	1.84
DGL-KE	8-GPUs	277	1.88
DGL-KE	Distributed	1622	2.22
PBG	1-GPU	3060	2.6
PBG	2-GPUs	1400	2.38
PBG	4-GPUs	515	1.75
PBG	8-GPUs	419	2.84
PBG	Distributed	1474	2.02

Table 7: Cost Comparisons with d=100 on Freebase86m.

that the host CPU utilization could be a potential bottleneck (P3.2xLarge instance only has 8 vCPUs) and we plan to study techniques to mitigate CPU bottlenecks.

Comparison vs. Distributed and Multi-GPU We compare the training time and cost per epoch[1] for Marius with the multi-GPU and distributed multi-CPU configurations of PBG and DGL-KE. PBG and DGL-KE support single machine multi-GPU training, and have a distributed multi-machine mode which is CPU-only. In the distributed configurations, the two systems partition parameters across the CPU memory of the machines and perform asynchronous training with CPU workers [18, 40]. Tables 6 and 7 show the configuration for each system and the corresponding epoch runtime and cost based on On-Demand AWS pricing. We observe that despite using a single GPU, Marius achieves comparable runtime with the multi-GPU configurations, while being more cost effective than all cases, ranging from 2.9× to 7.5× cheaper depending on the configuration. We also note that Marius can be extended to the multi-GPU setting; we discuss this in future work.

5.3 Partition Orderings

We now evaluate our buffer-aware *BETA* ordering and compare it to two Hilbert curve based orderings. The first, *Hilbert*, is the ordering generated directly from a Hilbert curve over the $n \times n$ matrix of edge buckets. The second, *HilbertSymmetric*, modifies the previous curve by processing edge buckets (i, j) and (j, i) together, which reduces the overall number of swaps that need to be performed by about 2×. All experiments use 32 partitions and a buffer capacity of 8 partitions.

[1] All three systems converge in a similar number of epochs.

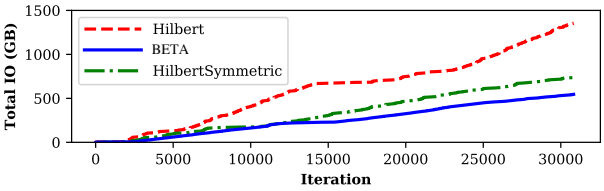


Figure 9: Total IO during a single epoch of training.

We compare the orderings on Freebase86m with $d = 50$ and $d = 100$ sized embeddings, where the latter configuration exceeds CPU memory size. For $d = 50$ we include an in-memory configuration which does not use partitioning as a baseline. Results are shown in Figure 10. We find that the *BETA* ordering reduces training time to nearly in-memory speeds, while only keeping 1/4 of the partitions in memory at any given time. The runtime of the three orderings is directly correlated with the amount of IO required to train a single epoch (Figure 9). Since the *Hilbert* and *HilbertSymmetric* orderings require more IO, training stalls more often waiting for IO to complete. Results for $d = 100$, also in Figure 10, show that *BETA* has the lowest training time, which is directly correlated with the amount of IO performed. Overall, the *BETA* ordering is well suited for training large-scale graph embeddings through reducing IO.

We also compare the orderings on Twitter with $d = 100$ and $d = 200$ sized embeddings. Results are shown in Figure 11. We find that the choice of ordering does not impact runtime for this configuration. Even though *BETA* results in the smallest amount of total IO, the prefetching of partitions to the buffer always outpaces the speed of computation for the other orderings. We see this in Twitter and not Freebase86m, because Twitter has nearly 10× the density of Freebase86m, i.e., more computation needs to be performed per partition. When we increase the embedding dimension to $d = 200$ (Figure 11) we see a difference in running time. By increasing the embedding dimension by 2× we increase the total amount of IO by 2×, and now the prefetching of partitions is outpaced by the computation.

Overall, we see that certain configurations are *data bound* and others are *compute bound*. For data bound configurations like $d = 50$ and $d = 100$ on Freebase86m and $d = 200$ on Twitter, the choice of ordering will impact overall training time, with *BETA* performing best. But for *compute bound* workloads such as $d = 100$ on Twitter, the choice of ordering makes little difference since the prefetching always outpaces computation.

5.4 Large Embeddings

We evaluate the ability of Marius to scale training beyond CPU memory sizes in this section. We vary the embedding dimension from a small dimension of $d = 20$, for which training fits in GPU memory, to a large em-

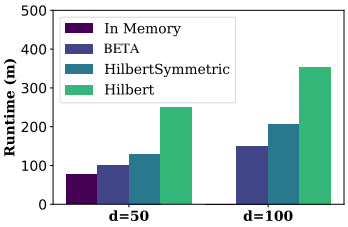


Figure 10: 10 epochs runtime per edge bucket ordering on Freebase86m.

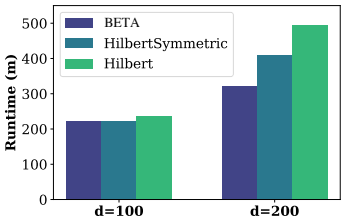


Figure 11: 10 epochs runtime per edge bucket ordering on Twitter.

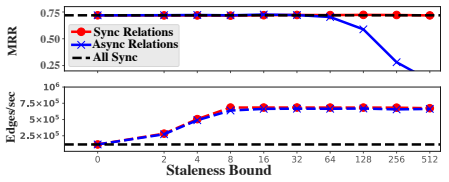


Figure 12: Impact of staleness bound.

d	Size	Partitions	MRR	Runtime (Epoch)
20	13.6 GB	-	.698	4m
50	34.4 GB	-	.722	4.8m
100	68.8 GB	32	.726	12.1m
400	275.2 GB	32	.731	92.4m
800	550.4 GB	64	.731	396m

Table 8: Freebase86m. $d = 400$ and $d = 800$ trained to 5 epochs, other cases are trained to 10.

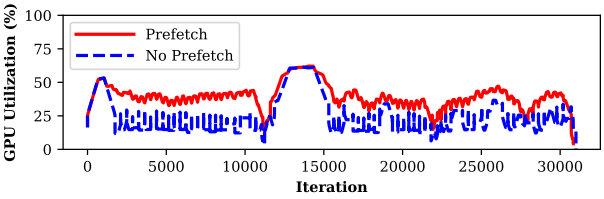


Figure 13: Effect of prefetching with Freebase86m.

bedding dimension $d = 800$, which is well beyond the memory capacity of a single P3.2xLarge instance. The results are shown in Table 8. We find that the embedding quality increases with increased embedding dimension. We also see that as the embedding size increases, the training time increases quadratically. We see this because the number of swaps and total IO scales quadratically with the number of partitions, if the buffer capacity is held fixed. And because training is bottlenecked by IO for large embedding sizes, we see quadratic runtime increases. It should be noted that with a faster disk we would observe improved runtimes, with a $2\times$ faster disk leading to $2\times$ faster training for large embeddings. With NVMe-based SSDs becoming more common, the design of Marius will best be able to leverage future fast sequential storage mediums and scale training to embedding sizes beyond what we show here.

5.5 Microbenchmarks

Bounded Staleness We now show how our pipelined training architecture with bounded staleness affects the embedding quality and throughput of training. We train Marius on Freebase86m with $d = 50$, and vary the number of batches allowed into the pipeline at any given time. We evaluate how the performance and MRR vary as we vary the staleness bound. We compare three cases, synchronous updates to all parameters, synchronous updates to only the relation embeddings, and asynchronous updates to all parameters.[2] Results are shown in Figure 12. We see that increasing the staleness bound when asynchronously updating the relation embeddings results in severe degradation of embedding quality. For synchronous updates of the relation embeddings and asynchronous updates to the node embeddings, we see that MRR does not degrade significantly with increasing staleness bound. This suggests that relation embeddings are sensitive to staleness, which might be due to dense updates. These results additionally show that node embeddings are not sensitive to asynchronous updates, which may be due to sparse updates. We also find that increasing the bound improves the throughput of the system by about a factor of 5 over synchronous training but that the benefits diminish beyond a staleness bound of 8.

Prefetching Effects We evaluate the effect of prefetching partitions to the buffer on GPU utilization. We train Marius on Freebase86m with $d = 100$, 32 partitions, and a buffer capacity of eight. We show the average utilization of the GPU during each iteration of a single epoch of training in Figure 13. We can see that prefetching results in a higher sustained utilization of the GPU since less time is spent waiting for partition swaps. Interestingly, both configurations see a utilization bump starting at about iteration 12,000. This is because the *BETA* ordering does not require any swaps during this period. Overall, prefetching is able to mitigate wait times for partition swaps improving utilization and training times.

6 Discussion

We next discuss some lessons learned from Marius deployments in the cloud and discuss how the *BETA* ordering aims to optimize a workload that is fundamentally different than those considered by prior large-scale graph processing paradigms.

6.1 Deployment Considerations

Given the diverse set of cloud computing instances offered by vendors, there are a wide variety of possible hardware deployments with associated costs and bene-

[2] For asynchronous updates to the relation embeddings, we pipe them to the GPU from CPU memory as with the node embeddings.

fits. It is challenging to determine what the best deployment option is, especially with performance also being impacted by the choice of model and dataset. Here we list some considerations when deploying Marius.

Properties of the Input Graph Training time and storage overhead are largely driven by the size of the input graph. More edges lead to more computation, and more nodes and edge-types results in a larger storage footprint. The density of the graph impacts the bottleneck of the system when using the partition buffer. A graph with high density will be compute bound, as more computation will have to be performed on each node partition, as we see in Figure 11. For sparse graphs the training will be data bound, as we see in Figure 10. For data bound settings, utilizing a storage device with high throughput can improve training times, while for compute bound workloads, more GPUs and parallelism can help.

Model Complexity Some models such as *DistMult*, *ComplEx*, and *Dot* are computationally simple, only requiring dot products and element-wise multiplication, while others such as CapsE [26] utilize convolutions and a capsule neural network. Training simple models requires less compute to perform the forward and backwards pass and therefore is more likely to be data bound. The opposite is true for complex models.

Configuration and Tuning A major challenge with training graph embedding models is the number of hyperparameters which impact embedding quality, training throughput, and convergence rates. In terms of batch size, we observe that large batches ($\approx$10000) can improve training throughput with no impact on model accuracy for large graphs, but throughput benefits diminish after a certain batch size. Throughput can also be increased by using a larger number of partitions (Figure 7), but this affects embedding quality. IO can also be reduced by increasing the capacity of the buffer, which quadratically reduces the number of swaps; thus it is best to size the buffer to the maximum number of partitions that will fit in CPU memory. Finally, as seen in Figure 12, increasing the staleness bound improves training throughput but can negatively impact embedding quality. Overall, the effect of these parameters are graph dependent, and efficiently tuning hyperparameters for a given graph is an interesting direction for future work.

6.2 Out-of-core Graph Processing

The graph embedding workload requires iterating over edges and computing on data associated with both endpoints (i.e., the embeddings of source, destination). The *BETA* ordering is designed to minimize IO when accessing node embedding vectors associated with edges that are being processed. Classic graph processing systems and methods such as GraphChi's Parallel Sliding Window (PSW) [17] are tailored for workloads that iterate over vertices and process data associated with the incoming edges of each node. Applying such schemes (e.g., PSW) to graph embeddings would require performing redundant IO (scaling quadratically with partitions) to access embeddings for both incoming/outgoing vertices. Furthermore, for classic graph processing algorithms such as PageRank, the storage overhead of node data is only a single float or a low dimensional vector. Based on this, traditional graph processing systems make the assumption that storing and accessing node data is inexpensive and fits in memory. In contrast, graph embeddings are high dimensional vectors making storing and accessing node data costly, and hence the workload requires new graph traversal algorithms to minimize IO.

7 Conclusion

We introduced Marius, a new framework for computing large-scale graph embedding models on a single machine. We demonstrated that the key to scalable training of graph embeddings is optimized data movement. To optimize data movement and maximize GPU utilization, we proposed a pipelined architecture that leverages partition caching and the *BETA ordering*, a novel buffer-aware data ordering scheme. We showed using standard benchmarks that Marius achieves the same accuracy but is up to an order-of magnitude faster than existing systems. We also showed that Marius can scale to graph instances with more than a billion edges and up to 550 GB of model parameters on a single AWS P3.2xLarge instance. In the future, we plan to explore how the ideas behind Marius' design and our new data ordering can be applied to distributed setting and help speed up training of graph neural networks.

Acknowledgements This work was supported by NSF under grant 1815538 and DARPA under grant ASKE HR00111990013. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is also supported by the National Science Foundation grant CNS-1838733, a Facebook faculty research award and by the Office of the Vice Chancellor for Research and Graduate Education at UW-Madison with funding from the Wisconsin Alumni Research Foundation.

References

[1] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26:2787–2795, 2013.

[3] Sylvain Brohee and Jacques Van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics*, 7(1):488, 2006.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC 2013*, pages 49–60, 2013.

[5] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[7] Kimm Fairchild, Steven E Poltrock, and George W Furnas. Graphic representations of large knowledge bases. *Cognitive science and its applications for human-computer interaction*, page 201, 1988.

[8] Google. Freebase data dumps. https://developers.google.com/freebase, 2018.

[9] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[10] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584, 2017.

[12] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system*. Now Publishers Inc, 2007.

[13] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[14] David Hilbert. Über die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[15] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[16] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, 2010.

[17] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[18] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287*, 2019.

[19] Jure Leskovec. WWW-18 Tutorial: Representation Learning on Networks. http://snap.stanford.edu/proj/embeddings-www/.

[20] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[21] LibTorch: PyTorch C++ API. https://pytorch.org/cppdocs.

[22] Hailun Lin, Yong Liu, Weiping Wang, Yinliang Yue, and Zheng Lin. Learning entity and relation embeddings for knowledge resolution. *Procedia Computer Science*, 108:345–354, 2017.

[23] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[24] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[25] 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.

[26] Dai Quoc Nguyen, Thanh Vu, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A capsule network-based embedding model for knowledge graph completion and search personalization, 2019.

[27] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 809–816, 2011.

[28] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 291–305, 2019.

[29] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[30] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[31] C. Seshadhri, Aneesh Sharma, Andrew Stolman, and Ashish Goel. The impossibility of low-rank representations for triangle-rich complex networks. *Proceedings of the National Academy of Sciences*, 117(11):5631–5637, 2020.

[32] Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. *Advances in neural information processing systems*, 16:659–666, 2003.

[33] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Eric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 2071–2080, 20–22 Jun 2016.

[34] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[35] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[36] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014.

[37] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[38] Reza Zafarani, Mohammad Ali Abbasi, and Huan Liu. *Social media mining: an introduction*. Cambridge University Press, 2014.

[39] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems*, 31:5165–5175, 2018.

[40] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. DGL-KE: Training knowledge graph embeddings at scale. *arXiv preprint arXiv:2004.08532*, 2020.

[41] Zhaocheng Zhu, Shizhen Xu, Meng Qu, and Jian Tang. Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*, pages 2494–2504. ACM, 2019.

A Artifact Appendix

Abstract

The artifact includes the Marius source code, configuration and scripts for all experiments, including the baselines. Details on how to use the artifact can be found in the README file in our Github repository.

Scope

The artifact can be used to validate and reproduce the results for all experiments. The source code and experiment configuration can be viewed to obtain any implementation details that were not mentioned in the paper for brevity. We do not include the source code to PyTorch Big-Graph and DGL-KE, the versions used in this work can be found at: `https://github.com/facebookresearch/PyTorch-BigGraph/tree/4571deee78d0fff974a81312c0c3231d7dc96a69` and `https://github.com/awslabs/dgl-ke/releases/tag/0.1.1`

Contents

Marius: The source code for Marius is mostly written in C++ with bindings to support a Python API. Located in `/src`. The version of Marius in the artifact (`osdi2021` branch) corresponds to the version used to produce results in this paper. We are also actively making improvements to Marius and the latest version of the system can be found in the main branch of the repository. The main branch of the repository also contains documentation on how to use Marius directly.

Experiment runner: A collection of Python scripts that can be used to run the experiments used in the paper. Located in `/osdi2021`. Experiments can be run with `python3 osdi2021/run_experiment.py --experiment <EXPERIMENT>`. A full list of experiments can be found by passing the `--help` flag. Once an experiment has run to completion, results are output to the terminal and detailed results, metrics and figures can be found in the experiment directory.

Experiment configuration: Configuration files for all experiments and baselines are stored in their corresponding experiment directory. The experiment directories are named based on their corresponding experiment in this paper. For example, the experiment configuration files and results after running the experiments for Section 5.2 can be found in `/osdi2021/system_comparisons/<DATASET>/<SYSTEM/`

Datasets and preprocessing code: Code to download and preprocess the datasets used in this paper are included in the artifact. For Marius they can be found in `/src/python/tools`. For DGL-KE and PBG, they can be found in `/osdi2021/dglke_preprocessing` and `/osdi2021/pbg_preprocessing`. We used four datasets in our evaluation (Table 1). FB15k and Freebase86m are a subset of the Freebase knowledge graph [8], where each edge is encoded as a triple. Each triple encodes general factual information. An example triple is of the form: `(Giannis Antetokounmpo, Plays, Basketball)`. LiveJournal [20] is a social network dataset where nodes represent users and edges denote friendships between them. Similarly, the Twitter [16] dataset contains a follower network between users.

Buffer simulator: The buffer simulator was used to develop and test edge bucket orderings. It computes the number of swaps for any edge bucket ordering for any number of partitions and any buffer size.

Hosting

Artifact:
`https://github.com/marius-team/marius/tree/osdi2021`

Latest version:
`https://github.com/marius-team/marius`

Requirements

Detailed software requirements and dependencies are listed in the artifact README. The artifact must be run on a machine with an NVIDIA GPU. The target deployment for this artifact is the P3.2xLarge instance from AWS. There are a few experiments which cannot run on this instance due to memory limitations. We detail these in the README.

P^3: Distributed Deep Graph Learning at Scale

Swapnil Gandhi*
Microsoft Research

Anand Padmanabha Iyer
Microsoft Research

Abstract

Graph Neural Networks (GNNs) have gained significant attention in the recent past, and become one of the fastest growing subareas in deep learning. While several new GNN architectures have been proposed, the scale of real-world graphs—in many cases billions of nodes and edges—poses challenges during model training. In this paper, we present P^3, a system that focuses on scaling GNN model training to large real-world graphs in a *distributed* setting. We observe that scalability challenges in training GNNs are fundamentally different from that in training classical deep neural networks and distributed graph processing; and that commonly used techniques, such as intelligent partitioning of the graph do not yield desired results. Based on this observation, P^3 proposes a new approach for distributed GNN training. Our approach effectively eliminates high communication and partitioning overheads, and couples it with a new *pipelined push-pull* parallelism based execution strategy for fast model training. P^3 exposes a simple API that captures many different classes of GNN architectures for generality. When further combined with a simple caching strategy, our evaluation shows that P^3 is able to outperform existing state-of-the-art distributed GNN frameworks by up to 7×.

1 Introduction

Deep learning, in the form of Deep Neural Networks (DNNs), has become the de-facto tool for several challenging applications in diverse fields such as computer vision [27], speech recognition [28] and natural language processing [18], where they have produced results on par with human experts [9]. In the recent past, there has been a significant interest in Graph Neural Networks (GNNs)—neural networks that operate on *graph structured data*—which has made them one of the fastest growing subareas in deep learning [25]. Due to the expressiveness of graphs in capturing the rich relational information between input elements, GNNs have enabled breakthroughs in many important domains including recommendation systems [51, 66], knowledge graphs [53], and drug discovery [46, 58].

In a GNN, the nodes in the input graph are associated with features and labels. Typical tasks in GNNs include node classification (predicting the class label of a node) [41], link prediction (predicting the possibility of a link between given nodes) [70] and graph classification (predicting the class label of a graph) [8]. To do these tasks, GNNs combine feature information with graph structure to learn *representations*—low-dimensional vector embeddings—of nodes. Thus, learning such *deep encodings* is the key goal of GNNs. Several novel GNN architectures exist today, including GraphSAGE [24], Graph Convolution Networks (GCNs) [17, 41] and Graph Attention Networks (GATs) [59]. While each have their own unique advantages, they fundamentally differ in *how* the graph structure is used to learn the embeddings and *what* neural network transformations are used to aggregate neighborhood information [64].

At a high level, GNNs learn embeddings by combining iterative graph propagation and DNN operations (e.g., matrix multiplication and convolution). The graph structure is used to determine *what* to propagate and neural networks direct *how* aggregations are done. Each node creates a k-hop *computation graph* based on its neighborhood, and uses their features to learn its embedding. One of the key differentiators between training GNNs and DNNs is the presence of dependencies among data samples: while traditional DNNs train on samples that are independent of each other (e.g., images), the connected structure of graph imposes dependencies. Further, it is common to have a large number of dense features associated with every node—ranging from 100s to several 1000s [29, 66, 68]—in the graph. Due to this, the k-hop computation graphs created by each node can be prohibitively large. Techniques such as *neighborhood sampling* [24] help to some extend, but depending on the graph structure, even a sampled computation graph and associated features may not fit in the memory of a single GPU, making scalability a fundamental issue in training GNNs [71]. With the prevalence of large graphs, with *billions* of nodes and edges, in academia and the industry [55], enabling GNN training in a *distributed* fashion[1] is an important and challenging problem.

In this paper, we propose P^3,[2] a system that enables efficient *distributed* training of GNNs on large input graphs. P^3 is motivated by three key observations. First, due to the data dependency, we find that *in distributed training of GNNs, a major fraction of time is spent in network communication to generate the embedding computation graph with features*. Second, we notice that relying on distributed graph processing techniques such as advanced partitioning schemes, while useful in the context of graph processing, do not benefit GNNs

*Work done during an internship at Microsoft Research.

[1] Using more than one machine, each with 1 or more GPUs.

[2] for *Pipelined Push-Pull*.

and in many cases could be detrimental. Finally, due to the network communication issue, we observed that GPUs in distributed GNN training are underutilized, and spend as much as 80% of the time blocked on communication (§2). Thus, P^3 focuses on techniques that can reduce or even eliminate these inefficiencies, thereby boosting performance.

P^3 is not the first to address GNN scalability challenges. While there are many available frameworks for GNN training, a majority of them have focused on single machine multi-GPU training and limited graph sizes [20, 45, 47]. Popular open-source frameworks, such as the Deep Graph Library (DGL) [1] have incorporated distributed training support. But as we show in this paper, it faces many challenges and exhibits poor performance due to high network communication. ROC [36] is a recent system that shares the same goal as P^3 but proposes a fundamentally different approach. ROC extensively optimizes GNN training using a sophisticated *online* partitioner, memory management techniques that leverage CPU and GPU, and relies on hardware support such as high speed interconnects (NVLink and InfiniBand). In contrast, P^3 only assumes PCIe links and Ethernet connection, and doesn't rely on any intelligent partitioning scheme. During training, ROC requires movement of features across machines, while in P^3, features are never transmitted across the network. Finally, our evaluation datasets are significantly larger than ROCs, which helped us uncover several challenges that may have been missed with smaller graphs.

To achieve its goal, P^3 leverages a key characteristic of GNNs: unlike traditional DNNs where the data samples (e.g., images) are small and model parameters are large (e.g., 8 billion for Megatron [56], 17 billion for TuringNLG [7]), GNNs have small model parameters but large data samples due to the dense feature vectors associated with each node's sampled computation graph. As a result, movement of these feature vectors account for the majority of network traffic in existing GNN frameworks. In P^3, we avoid movement of features entirely, and propose distributing the graph structure and the features across the machines *independently*. For this, it only relies on a random hash partitioner that is fast, computationally simple and incurs minimal overhead. Additionally, the hash based partitioning allows work balance and efficiency when combined with other techniques P^3 incorporates (§3.1).

During embedding computation, P^3 takes a radically different approach. Instead of creating the computation graph by pulling the neighborhood of a node and the associated features, P^3 only pulls the graph structure, which is significantly smaller. It then proposes *push-pull* parallelism, a novel approach to executing the computation graph that combines *intra-layer model parallelism* with *data parallelism*. P^3 *never* moves features across the network, instead it *pushes* the computation graph structure in the most compute intensive layer (layer 1) to all the machines, and thereafter executes operations of layer 1 using *intra-layer model parallelism*. It then *pulls* much smaller *partial activations*, accumulates them, and proceeds to execute operations of the remaining $k-1$ layers using *data parallelism* (§3.2).

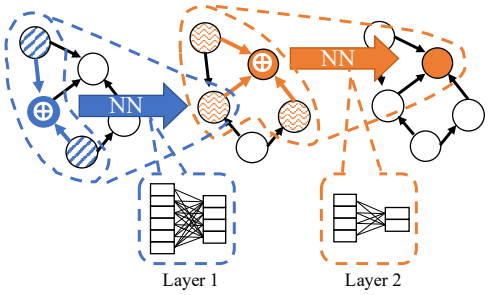


Figure 1: A two-layer GNN that uses DNN at each layer along with iterative graph propagation for learning.

Due to the partitioning strategy and the push-pull parallelism based execution, P^3 is able to use a simple pipelining technique that overlaps most of the computation and communication efficiently, thus effectively hiding (the already small) communication latencies (§3.3). Further, the partitioning strategy also enables P^3 to propose a simple caching mechanism that greedily caches graph and/or feature partitions on multiple machines if memory permits for further reduction in network communication (§3.4). P^3's proposed techniques are general and are applicable to several state-of-the-art GNN architectures. P^3 also wraps all these optimizations in a simple P-TAGS API (`partition`, `transform`, `apply`, `gather`, `scatter` and `sync`) that developers can use to write new GNN architectures that can benefit from its techniques (§3.5).

The combination of these techniques enable P^3 to outperform DGL [1], a state-of-the-art distributed GNN framework, by up to 7×. Further, P^3 is able to support much larger graphs and scale gracefully (§5).

We make the following contributions in this paper:

- We observe the shortcomings with applying distributed graph processing techniques for scaling GNN model training (§2). Based on this, P^3 takes a radically new approach of relying only on *random hash* partitioning of the graph and features *independently*, thus effectively eliminating the overheads with partitioning. (§3.1)
- P^3 proposes a novel *hybrid parallelism* based execution strategy that combines intra-layer model parallelism with data parallelism that significantly reduces network communication and allows many opportunities for pipelining compute and communication. (§3.2)
- We show that P^3 can scale to large graphs gracefully and that it achieves significant performance benefits (up to 7× compared to DGL [1] and up to 2× compared to ROC [36]) that increase with increase in input size. (§5)

2 Background & Challenges

We begin with a brief background on GNNs, and then motivate the challenges with distributed training of GNNs.

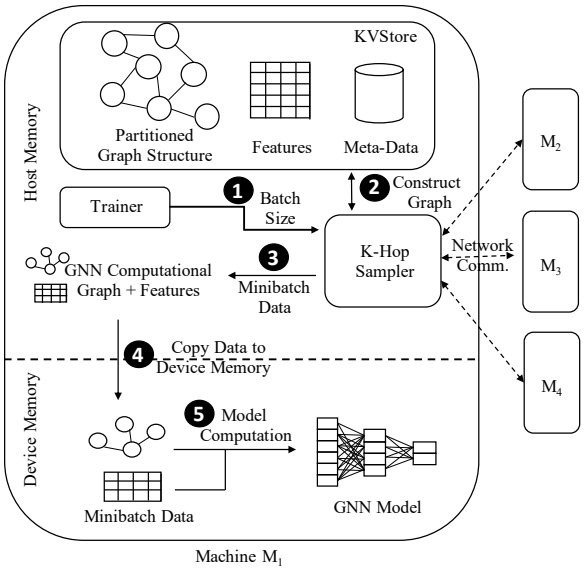


Figure 2: Existing GNN frameworks combine distributed graph processing and DNN techniques.

2.1 Graph Neural Networks

A GNN is a neural network that operates on graph structured data as input. The input graphs contain nodes (entities), and edges (relation between nodes) and features for all nodes. The basic operation in GNNs is to obtain the *representations* of nodes in the graph. They map nodes to a d-dimensional embedding space such that similar nodes (e.g., by proximity) in the graph are embedded close to each other. To obtain these embeddings, GNNs combine feature information associated with the nodes and the graph structure using information propagated and transformed from its neighborhood. In computing the embedding, the graph structure indicates *what* is propagated, and a neural network is used to determine *how* the propagated information is transformed. The exact neighborhood from which the embedding is derived is configurable, and typically GNNs use k (usually 2 or more) hops from a node [26]. The neural network which transforms information at each hop is called a *layer* in the GNN, hence a 2-hop (k-hop) neighborhood translates to a 2 (k) layer GNN (fig. 1).

Theoretically, the embedding z_v of node v after k layers of neighborhood aggregation can be obtained as h_v^k [24], where:

$$h_{N(v)}^k = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{k-1} \mid u \in N(v)\right\}\right) \quad (1)$$

$$h_v^k = \sigma\left(W_k \cdot \text{COMBINE}^{(k)}(h_v^{k-1}, h_{N(v)}^k)\right) \quad (2)$$

Here, h_v^i is the representation of node v after i layers of aggregation and W_i is the trainable weight matrix that is shared by all nodes for layer i ($i \geq 1$). h_v^0 is initialized using the node features. The choice of $\text{AGGREGATE}^{(k)}(.)$ and $\text{COMBINE}^{(k)}(.)$ is crucial for how the layers are defined and the embeddings are computed.

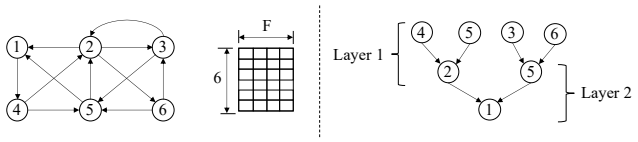


Figure 3: Sampling a two-layer GNN computation graph by restricting the neighborhood size to 2 (minibatch size: 1).

2.2 Distributed Training of GNNs

Existing frameworks for training GNNs, such as the Deep Graph Library (DGL) [1], support distributed training by combining distributed graph processing techniques with DNN techniques, as shown in fig. 2. The input graph along with the features is partitioned across machines in the cluster. Given a batch size (1), the computation graph for each node, commonly referred to as a *training sample*, in the batch is generated by pulling the k-hop neighborhood of each node along with the associated features (2). This requires communication with other machines in the cluster. Once the computation graphs are constructed, standard DNN techniques such as *data parallelism* is used to execute the computation—minibatches are created and copied to the GPU memory (4), and then the model computation is triggered (5).

2.3 Challenges in Distributed GNN Training

There are several challenges that need to be addressed to make distributed GNN training efficient.

2.3.1 Challenge #1: Communication Bottlenecks Due to Data Dependencies

Unlike traditional DNNs where the training data are independent of each other (e.g., images), GNNs impose a dependency between the training inputs in the form of graph structure. Thus, even though provided batch size could be small (e.g., 1000), the computation graph samples could be exponentially larger due to the k-hop neighborhood and the associated features. A major reason for such large size is not the graph structure in itself, but the features, whose sizes typically range in 100s to several 1000s [29, 41, 66, 68]. In real world graphs consisting of billions of nodes and edges [15, 55], the 2-hop neighborhoods could be up to an order of magnitude larger than the 1-hop neighborhood [43]. When combined with the features, the resulting computation graph may easily exceed the memory of a single GPU or even main memory of a server. A common technique used to address such *neighborhood explosion* is sampling [24]. That is, instead of getting all the neighbors of the node in each hop, we select only a fixed number. An example is shown in fig. 3, where node 1's 2-hop computation graph is generated by sampling two neighbors at each hop. However, even with sampling, the size of the computation graph could grow substantially, based on the sampling used and the number of layers in the GNN. Since these neighborhood nodes and their features must be obtained through the network, distributed training of GNNs spend a major fraction of time in network communication.

Scheme	Time(s)	Memory(GB)	Epoch(s)
Hash [48]	2.87	58	9.833
METIS [38]	4264	63	5.295
RandomVertexCut [22]	36.95	185	5.494
GRID [23]	51.82	128	6.866
3D [69]	134	118	6.027

Table 1: Partitioning techniques are not effective in GNNs. All schemes, except hash, reduce the epoch time, but at the cost of significant partitioning time or memory overheads.

2.3.2 Challenge #2: Ineffectiveness of Partitioning

Partitioning is a common approach used to achieve scalability in distributed graph processing, and existing GNN frameworks leverage popular partitioning strategies to distribute the graph and features across machines. However, there are two shortcomings with this approach.

First, many partitioning scheme incur a cost in terms of computation and/or memory overhead. In table 1, we show the partitioning time, memory consumption and the time to complete one epoch of training on a representative GNN, GraphSAGE [24] for four different partitioning schemes: Hash [48], which partitions nodes using random hashing, METIS [38], a balanced min edge-cut partitioner, RandomVertexCut [22] and GRID [23], are vertex-cut partitioners, and 3D [69], a recently proposed scheme for machine learning workloads. We see that the best performing partitioning schemes (e.g., edge-cut) incur high computation overheads. Computationally faster schemes incur either high memory overhead (due to replication, e.g., vertex-cut) or performance hit.

Second, the benefits of partitioning are severely limited as the layers in the GNN increase. Recall that GNNs use k-hop neighborhood to compute the embedding. While partitioning schemes reduce communication, *they only optimize communication at the first hop*. Thus, when the number of layers increase, all partitioning schemes fail.

2.3.3 Challenge #3: GPU Underutilization

Existing GNN frameworks utilize DNN techniques, such as *data parallelism* to train GNN models. In data parallel execution, each machine operates on a different set of samples. However, due to the data dependency induced communication bottleneck described earlier, we observed that in distributed GNN training using the popular framework DGL, GPUs are only being utilized $\approx 20\%$ of the time. For a large fraction ($\approx 80\%$) of the time, GPUs are waiting on communication. Recent work has reported data copy to be the major bottleneck in training GNNs in *single machine multi-GPU* setup [45], but we found that data copy only accounts for 5% of the time while training GNNs using *distributed multi-GPU* setup. We note that the proposed techniques in [45] are orthogonal to our work and can benefit if applied to P^3. Thus, GPUs are heavily underutilized in distributed GNN training due to network communication. Alternative parallelism techniques, such as model parallelism do not work for GNNs. This is because for each layer, they would incur intra-layer communication in-addition to data dependency induced communication and thus perform even worse compared to data parallelism.

3 P^3: Pipelined Push-Pull

P^3 proposes a new approach to distributed GNN training that reduces the overhead with computation graph generation and execution to the minimum. To achieve this, P^3 incorporates several techniques, which we describe in detail in this section.

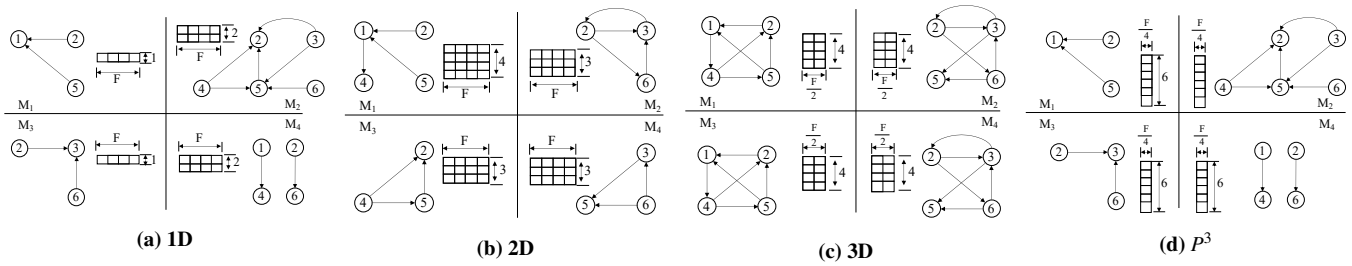
3.1 Independent Hash Partitioning Graph & Features

As we show in §2, partitioning of the input graph in an intelligent manner doesn't benefit GNN architectures significantly due to the characteristics of GNNs. Hence, in P^3, we use the simplest partitioning scheme and advocate for *independently* partitioning the graph and its features.

The nodes in the input graph are partitioned using a random hash partitioner, and the edges are co-located with their incoming nodes. This is equivalent to the commonly used 1D partitioning scheme available in many distributed graph processing frameworks [22, 67], and is computationally simple. Unlike other schemes (e.g., 2D partitioning), this scheme doesn't require any preprocessing steps (e.g., creating local ids) or maintaining a separate routing table to find the partition where a node is present, it can simply be computed on the fly. Note that this partitioning of the graph is only to ensure that P^3 can support large graphs. In several cases, the graph structure (nodes and edges without the features) of real-world graphs can be held in the main memory of modern server class machines. In these cases, P^3 can simply replicate the entire graph structure in every machine which can further reduce the communication requirements.

While the graph structure may fit in memory, the same cannot be said for input features. Typical GNNs work on input graphs where the feature vector sizes range in 100s to several 1000s [29, 41, 66, 68]. P^3 partitions the input features along the feature dimension. That is, if the dimension of features is F, then P^3 assigns F/N features of every node to each of the machines in a N machine cluster. This is in contrast to existing partitioning schemes tuned for machine learning tasks, including the recently proposed 3D partitioning scheme [69]. Figure 4 shows how P^3 partitions a simple graph in comparison with existing popular partitioning schemes.

As we shall see, this independent, simple partitioning of the graph and features enable many of P^3's techniques. Breaking up the input along the feature dimension is crucial, as it enables P^3 to achieve work balance when computing embeddings; as the hash based partitioner ensures that the nodes and features in the layers farther from the node whose embedding is computed to be spread across the cluster evenly. The simplicity of independently partitioning the structure and features also lets P^3 cache structure and features independently in its caching mechanism (§3.4).



(a) 1D (b) 2D (c) 3D (d) P^3

Figure 4: P^3 *independently* partitions the graph structure and the associated features (shown in fig. 3) using simple random hash partitioning in contrast to more intelligent schemes. This allows P^3 to achieve work balance, and enables many of its techniques.

3.2 Push-Pull Parallelism

With the input graph and features partitioned, P^3 adopts the common, minibatch centric computation for GNNs, similar to existing GNN frameworks, where it first generates the computation graph for a node and then executes it. We use fig. 5 to explain this in detail.

3.2.1 Computation Graph Generation

At the beginning of every minibatch, each node whose embedding is being computed generates its computation graph. To do so P^3 *pulls* [3] the k-hop neighborhood for the node. If the GNN architecture supports a sampling based embedding computation, P^3 pulls the sampled k-hop neighborhood, otherwise it pulls the full k-hop neighborhood. Note that unlike existing GNN frameworks, the *features are not pulled* in either case. This significantly reduces the network communication necessary for creating the computation graph. If the entire graph structure is available in every machine, this is a local operation, otherwise it results in minimal network communication as the graph structure is very light weight. At the end of this process, P^3 ends up with the k-layer computation graph of each node in the minibatch at the machine which owns the node (e.g., the four *samples* in fig. 5 correspond to computation graphs of four nodes in the minibatch). Note that existing GNN frameworks *pulls* features in addition to the structure, so in these frameworks, the machine owning the node ends up with both the computation graph and all the features necessary for embedding computation.

In the case of existing GNNs, each machine can now independently execute the complete computation graph with features it obtained in a data parallel fashion, starting at layer 1 and invoking global gradient synchronization at each layer boundary as shown in fig. 5a in the backward pass. However, since P^3 does not move features, the computation graphs cannot be executed in a data parallelism fashion. Here, P^3 proposes a hybrid parallelism approach that combines model parallelism and data parallelism, which we term *push-pull parallelism*. While model parallelism is rarely used in traditional DNNs due to the underutilization of resources and difficulty in determining how to partition the model [49], P^3 uses it to its advantage. Due to the nature of GNNs, the model (embedding computation graphs) is easy to partition cleanly since the boundaries (hops) are clear. Further, due to P^3's partitioning strategy, model parallelism doesn't suffer from underutilization of resources in our context.

[3] *Pulling* refers to copying, possibly over the network if not local.

3.2.2 Computation Graph Execution

To start the execution, P^3 first *pushes* the computation graph for layer 1 to all the machines, as shown in ❶ in fig. 5b. Note that layer 1 is the most compute intensive, as it requires input features from layer 0 (having most fan-out) which are evenly spread in P^3 due to our partitioning scheme. Each machine, once it obtains the computational graph, can start the forward pass for layer 1 in a *model parallel* fashion (layer 1_M). Here, each machine computes *partial* activations for layer 1_M using the partition of input features it owns (❷). Since all GPUs in the cluster collectively execute the layer which requires input from the most fan-out, this avoids underutilization of GPUs. We observed that GPUs in existing GNN frameworks (e.g., DGL) spend $\approx 80\%$ of the time blocked on network compared to $\approx 15\%$ for P^3. Once the partial activations are computed, the machine assigned to each node in our hash partitioning scheme *pulls* them from all other machines. The node receiving the partial activations aggregates them using a reduce operation (❸). At this point, P^3 switches to *data parallelism* mode (layer 1_D). The aggregated partial activations are then passes through the rest of layer 1_D operations (if any, e.g., non-linear operations that cannot be partially computed) to obtain the final activations for layer 1 (❹). The computation proceeds in a *data-parallel* fashion to obtain the embedding at which point the forward pass ends (❺).

The backward pass proceeds similar to existing GNN frameworks in a *data parallel* fashion, invoking global gradient synchronizations until layer 1_D (❻). At layer 1_D, P^3 *pushes* the error gradient to all machines in the cluster (❼) and switches to *model parallelism*. Each machine now has the error gradients to apply the backward pass for layer 1_M *locally* (❽) and the backward pass phase ends.

While the partial activation computation in a *model parallel* fashion seemingly works in the general sense, they are restricted to transformations that can be aggregated from partial results. However, in certain GNN architectures

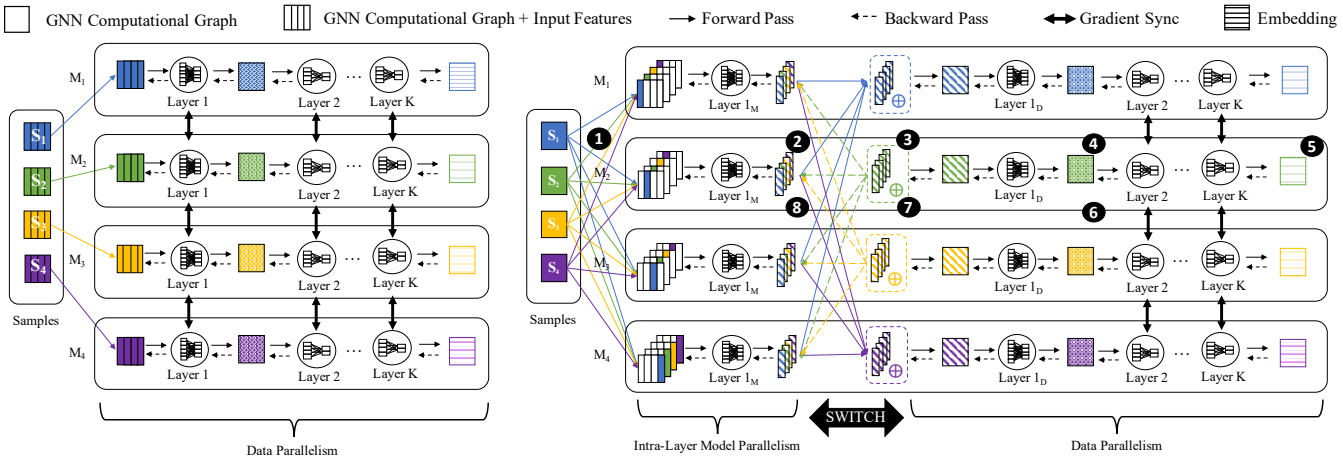


Figure 5: How existing GNN frameworks generate and execute computation graphs (left) and how P^3 does it (right) (§3.2).

(e.g., GAT [59]), layer 1_M in itself may introduce non-linear transformations. P^3 relies on developer input to determine the tensors that require global synchronizations during the model parallel execution to ensure correctness (§3.5).

At a first glance, the additional steps in P^3, namely the need to push graph structure in layer 1, aggregation of partial activations during the forward pass and the additional movement of gradients in the backward pass may seem like overheads that may lead to inefficiencies compared to simply pulling the features along with the graph structure and executing everything locally as in existing GNN frameworks. However, P^3's approach results in significant savings in network communication. First, P^3 doesn't pull features at all which tremendously reduces network traffic—typically the 2-hop neighborhood in the GNN computation graphs is an order of magnitude more than the 1-hop neighborhood. Second, regardless of the number of layers in the GNN, only layer 1 needs to be partially computed and aggregated. Finally, the size of the activations and gradients are small in GNNs (due to the smaller number of hidden dimensions), thus transferring them incurs much less overhead compared to transferring features.

To illustrate this, we use a simple experiment where we run a representative GNN, a 2-layer GraphSAGE [24] on the open-source OGB-Product [29] dataset on 4 machines. We pick 1000 labeled nodes to compute the embeddings and use neighborhood sampling (fan-out:25,10). The nodes are associated with feature vectors of size 100, and there are 16 hidden dimensions. At layer 0 (2-hops), there are 188339 nodes and at layer 1 (1-hop) there are 24703 nodes. Pulling features along with graph structure would incur 71.84 MB of network traffic. On the other hand, the activation matrix is of size input $\times$ hidden dimension. P^3 only needs to transfer the partial activations from 3 other machines, thus incurring just 5 MB ($3 \times 24703 \times 16$). Hence, by distributing the activation computation of the layer that holds the largest number of features, P^3 is able to drastically reduce network communication.

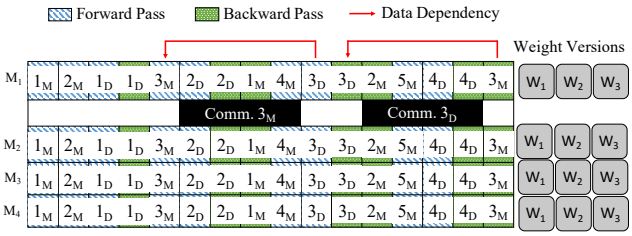


Figure 6: We employ a simple pipelining mechanism in P^3, inspired by the work in PipeDream [49]. This allows P^3 to effectively hide communication latencies by overlapping communication with computation.

3.3 Pipelining

Although P^3's push-pull parallelism based GNN computation graph creation and execution incurs less network communication compared to existing GNN frameworks, it needs to communicate more times: P^3 needs to push the graph structure of layer 1, pull partial activations in the forward pass and finally push the gradients in the backward pass. Further, since P^3 focuses on distributed settings, data copy is necessary between CPU and GPU. As a result, the computation is stalled during communication unless they are overlapped using pipelining techniques. Note that current GNN frameworks (e.g., DGL) already overlap computation and communication—while the CPU is busy creating the computation graph, the GPU is used to execute an already prepared mini batch.

In P^3, we employ a simple pipelining mechanism, inspired by PipeDream's pipeline parallelism [49]. Due to the approach we take in P^3 to enable push-pull parallelism, namely switching between model and data parallelism at layer 1, P^3 needs to execute four phases per minibatch: a model parallel

API	Description
`partition(graph, feat, topo_part_fn, ft_part_fn)`	Partition graph and input features *independently* using `topo_part_fn` and `ft_part_fn` respectively.
`scatter(graph, feat, udf)` → `msg`	Generates message `msg` by combining `src_ft`, `edge_ft`, and `dst_ft`.
`gather(msg, udf)` → `a_ft`	Computes aggregated neighbourhood representation from incoming messages.
`transform(v_ft, a_ft)` → `t_ft`	Computes partial output representation from partial input representation; requires sync.
`sync(t_ft, op='sum')` → `sync_ft`	Accumulates partial representations using user-defined arithmetic operation.
`apply(v_ft, sync_ft)` → `ft`	Computes output representation from input representation.

Table 2: The simple P-TAGS API exposed by P^3. Developers can use this API to scale up new GNN architectures.

phase in the forward pass, a data parallel phase in the forward pass, a data parallel phase in the backward pass and finally a model parallel phase in the backward pass. This provides us the opportunity to schedule 3 minibatches of computation before a data dependency is created between phases. Thus, we overlap computations between these, as shown in fig. 6. As shown, in the forward pass, the data parallel phase of minibatch 3 (denoted as 3_D) has a data dependency on the model parallel phase (3_M) in the forward pass. Hence, when phase 3_M starts communication, we schedule two forward and two backward passes from other minibatches. This 2 forward, 2 backward static scheduling strategy allows us to avoid stalls. We currently use static pipeline scheduling—while a profiling based methodology to identify a pipeline schedule may provide benefits, we defer it to a future work.

Bounded Staleness The main challenge with using pipelining as described above is the introduction of *staleness* which can be characterized by *pipeline delay*: the number of optimizer steps that have passed between when a weight is read for computing gradient and when that computed gradient is used for updating the weight. This delay is bound by the number of minibatches in the pipeline at any time and also exists in prior work [49, 52]. For P^3, this delay is fixed and bound to three, resulting in weight updates of the form:

$$w_{t+1} = w_t - \alpha \cdot \nabla f(w_{t-3}) \tag{3}$$

where, w_t is weight values after t optimizer steps, ∇f is the gradient function, α is learning rate and w_{t-3} the weight used in forward and backward passes. While unbounded stale gradient updates can negatively affect the statistical efficiency of the network, preventing convergence and producing models with lower accuracy, bounded delay enables P^3 reach target accuracy in the same number of epochs as data parallelism.

Memory Overhead While P^3's peak memory footprint is relatively on par to data parallelism, stashed weights can result in additional memory overhead. Presently, GNN models typically contain only a couple of layers of small DNN models, and therefore even with weight stashing the overhead is relatively small. This however may change in future as GNN models become larger and complex. P^3's memory overhead can be further reduced by leveraging prior work aimed at decreasing memory footprint of training DNN models [33, 34].

3.4 Caching

The use of *independent* partitioning for the graph structure and features allows P^3 to employ a simple caching scheme that can reduce the already minimal communication overhead. This is based on the observation that depending on the graph and the size of the features, either the graph or the features may be accommodated in less machines than what is available. By default, the features and the graph are partitioned without duplication across all available machines. However, when host memory is available, P^3 uses a simple greedy approach to utilize all the available free memory by caching the partitions of the graph and/or features on multiple machines using a user-defined setting. In its current state, we do simply caching, where we try to fit the input in the minimum number of machines, and create copies (caches) of partitions on other machines. We assume homogeneous machines, which is typically the standard in DNN/GNN training [35]. We believe that there are opportunities to design a better caching scheme, and plan to explore it in the future.

3.5 P^3 API

P^3 wraps its independent partitioning strategy, pipelined push-pull parallelism and caching in a simple API, which developers can use to speed up new GNN architectures. The API, shown in table 2, consists of the following six functions :

- `partition` is a user-provided composite function which independently partitions graph topology and input features across machines. This step is essential to balance load and reduce communication.
- `scatter` uses a user-provided message function defined on each edge to generate a message by combining the edge representation with the representations of source and destination vertices.
- `gather` uses a user-provided commutative and associative function (e.g. sum, mean) defined on each vertex to compute the neighborhood representation by aggregating incoming messages.
- `transform` is a user-provided composite function defined on each vertex to generate partial representation by applying zero or more element-wise NN operations[4] (e.g. add),

[4]Element-wise NN operation operates on elements occupying the same index position within respective tensors.

```
class GraphSAGE(nn.module):
    def __init__(in_ft, out_ft):
        fc_self = fc_neigh = Linear(in_ft, out_ft)
    # Generates message
    def scatter_udf(s_ft, e_ft, d_ft): return s_ft
    # Aggregates messages
    def gather_udf(msg): return mean(msg)
    # Computes partial activation; requires sync.
    def transform(v_ft, a_ft):
        return fc_self(v_ft) + fc_neigh(a_ft)
    # Computes vertex representation
    def apply(v_ft, t_ft):
        return ReLU(t_ft)
    def forward(graph, feat):
        graph['m'] = scatter(graph, feat, scatter_udf)
        graph['n_p'] = gather(graph['m'], gather_udf)
        graph['n_p'] = transform(feat, graph['n_p'])
        return apply(feat,sync(graph['n_p'],op='sum'))
```

Listing 1: Using P^3's P-TAGS API to implement GraphSAGE.

followed by at most one non-element-wise NN operation (e.g. convolution) on vertex features and the aggregated neighborhood representation.

- `sync` accumulates partial representation (generated by the `transform` API) over the network using the user-provided arithmetic operation.
- `apply` is a user-provided composite function defined on each vertex to generate representation by applying zero or more element-wise and non-element-wise NN operations on vertex features and input representation.

Listing 1 outlines how GraphSAGE [24] can be implemented in P^3. Using our API, the developer composes `forward` function—function which generates output tensors from input tensors. Generated computational graph (see §3.2.1) and representation computed in the previous layer (or input vertex features partitioned along feature dimension if the first layer is being trained) are inputs to the `forward` function. For every layer in the GNN model, each vertex first aggregates the representation of its immediate neighborhood by applying element-wise mean (see `gather_udf`) over the incoming source vertex representation (see `scatter_udf`). Next, vertex's current representation and aggregated neighborhood representation are fed through a fully connected layer, element-wise summed (see `transform`) and passed through the non-linear function ReLU (see `apply`), which generates the representation used by the next layer. If this is the last layer, the generated representation is used as the vertex embedding for downstream tasks.

While training the first layer, the input representation is partitioned along the contracting (feature) dimension and evenly spread across machines, which results in the output representation generated by non-element-wise operators requiring synchronization. Notably, element-wise operations can still be applied without requiring synchronization. Since `transform` feeds the partitioned input representation through a fully connected layer, a non-element wise operator, its output representation must be synchronized before applying other downstream operators. `sync` accumulates partial representation over the network and produces the output representation which can be consumed by `apply`. Input representation in all layers except the first are partitioned along the batch dimension, and therefore the corresponding output representations do not require synchronization; thus `sync` is a no-op for all layers except the first.

4 Implementation

P^3 is implemented on Deep Graph Library (DGL) [1], a popular open-source framework for training GNN models. P^3 uses DGL as a graph propagation engine for sampling, neighborhood aggregation using message passing primitives and other graph related operations, and PyTorch as the neural network execution runtime. We extended DGL in multiple ways to support P^3's pipelined push-pull based distributed GNN training. First, we replaced the dependent graph partitioning strategy—features co-located with vertices and edges—in DGL with a strategy that supports partitioning graph structure and features independently. We reuse DGL's k-hop graph sampling service: for each minibatch a sampling request is issued via an Remote Procedure Call (RPC) to local and remote samplers. These samplers access locally stored graph partitions and return sampled graph—topology and features—to the trainer. Unlike DGL, sampling service in P^3 only returns the sampled graph topology and does not require input features to be transferred. Second, trainers in P^3 execute the GNN model using pipelined data and model parallelism. Each minibatch is assigned a unique identifier, and placed in a work queue. The trainer process picks minibatch samples and its associated data from the front of the queue,minibatch and applies neural network operations. P^3 schedules 3 concurrent minibatches using 2 forward, 2 backward static scheduling strategy (§3.3) to overlap communication with computation. Before the training mode for a minibatch can be switched from model to data parallelism, partial activations must be synchronized. To do so, we extended DGL's `KVStore` to store partial activations computed by trainers. `KVStore` uses RPC calls to orchestrate movement of partial activation across machines, and once synchronized, copies accumulated activation to device memory and places a pointer to the associated buffer in the work queue, shared with the trainer process. PyTorch's `DistributedDataParallel` module is used to synchronize weights before being used for weight update.

5 Evaluation

We evaluate P^3 on several real-world graphs and compare it to DGL and ROC, two state-of-the-art GNN frameworks that support distributed training. Overall, our results show that:

- P^3 is able to improve performance compared to DGL by up to 7× and ROC by up to 2.2×; and its benefits increase with graph size.

Graph	Nodes	Edges	Features
OGB-Product [29]	2.4 million	123.7 million	100
OGB-Paper [29]	111 million	1.6 billion	128
UK-2006-05 [10, 11]	77.7 million	2.9 billion	256
UK-Union [10, 11]	133.6 million	5.5 billion	256
Facebook [19]	30.7 million	10 billion	256

Table 3: Graph datasets used in evaluating P^3. Features column shows the number of features *per* node.

- We find that P^3 can achieve graceful scaling with number of machines and that it matches the published accuracy results for known training tasks.
- Our caching and pipelining techniques improve performance by up to 1.7×, with benefits increasing with more caching opportunities.

Experimental Setup: All of our experiments were conducted on a GPU cluster with 4 nodes, each of which has one 12-core Intel Xeon E5-2690v4 CPU, 441 GB of RAM, and four NVIDIA Tesla P100 16 GB GPUs. GPUs on the same node are connected via a shared PCIe interconnect, and nodes are connected via a 10 Gbps Ethernet interface. All servers run 64-bit Ubuntu 16.04 with CUDA library v10.2, DGL v0.5, and PyTorch v1.6.0.

Datasets & Comparison: We list the five graphs we use in our experiments in table 3. The first two are the largest graphs from the OGB repository [29]—OGB-Products [29], an Amazon product co-purchasing network, and OGB-Papers [29], a citation network between papers indexed by Microsoft Academic Graph [57]—where we can ensure correctness and validate the accuracy of P^3 on various tasks compared to the best reported results [4]. The latter three—UK-2006-05 [10, 11], a snapshot of .uk domains, UK-Union [10, 11], a 12-month time-aware graph of the same and Facebook [19], a synthetic graph which simulates the social network—are used to evaluate the scalability of P^3. We selected these due to the lack of open-source datasets of such magnitude specifically for GNN tasks. The two OGB graphs contain features. For the remaining three, we generate random features ensuring that the ratio of labeled nodes remain consistent with what we observed in the OGB datasets. Together, these datasets represent some of the largest open-source graphs used in the evaluation in recent GNN research[5]. We present comparisons against DGL [1, 61] and ROC [36], two of the best performing open-source GNN frameworks that support distributed training—to the best of our knowledge—at the time of our evaluation. However, due to the limitations imposed by ROC at the time of writing, specifically its support for only full-batch training and the availability of GCN implementation only, we compare against ROC only when it is feasible to do so and use DGL for the rest of the experiments. While DGL uses the METIS partitioner as the default, we change it to use hash partitioning in all the evaluations unless specified. This is due to two reasons. First, hash is the only partitioner that can handle all the five graphs in our datasets without running out of memory. Second, METIS incurs significant computational overheads that often exceed the total training time (see §2).

Models & Metrics: We use four different GNN models: S-GCN [63], GCN [17, 41], GraphSAGE [24] and GAT [59], in the increasing order of model complexity. These models represent the state-of-the-art architectures that can support all GNN tasks (§2). Unless mentioned otherwise, we use a standard 2-layer GNN model for all tasks. We enable sampling (unless stated) for all GNN architectures because it represents the best case for our comparison system and one of standard approaches to scaling. The sampling approach we adopted, based on recent literature [24], is a (25, 10) neighborhood sampling where we pick a maximum of 25 neighbors for the first hop of a node, and then a maximum of 10 neighbors for each of those 25. Both GraphSAGE and GCN use a hidden-size of 32. For the GAT model, we use 8 attention heads. Minibatch size is set to 1000 in all our experiments. We use a mix of node classification and link prediction tasks where appropriate for the input. Graph classification tasks are usually done on a set of small graphs, hence we do not include this task. We report the average *epoch time*, which is the time taken to perform one pass over the entire graph, unless otherwise stated. We note that for training tasks to achieve reasonable accuracy, several 100s or even 1000s of epochs are needed. In the experiment evaluating the accuracy attained by the model, we report the total time it takes to achieve the best reported accuracy (where available). For experiments that evaluate the impact of varying configurations (e.g., features), we pick a middle of the pack dataset in terms of size (OGB-Paper) and GNN in terms of complexity (GraphSAGE).

5.1 Overall Performance

We first present the overall results. Here, we compare DGL and P^3 in terms of per epoch time. For P^3, we disable caching (§3.4) so that it uses the *same amount of memory* as DGL for a fair comparison. Note that enabling caching only benefits P^3, and we show the benefits of caching later in this section. We train all the models on all the graphs, and report the mean time per epoch. The results are shown in table 4.

We see that P^3 outperforms DGL across the board, and the speedups range from 2.08× to 5.43×. The benefits increase as the input graph size increases. To drill down on why P^3 achieve such superior performance, we break down the epoch time into its constituents: embedding computation graph creation time (indicated as DAG), data copy time and the computation time which is the sum of the forward pass time, backward pass time and update time (§2). Clearly, P^3's independent partitioning strategy and the hybrid parallelism significantly reduces the time it takes to create the computation graph, which dominates the epoch time. We see a slight increase in data copy and compute times for P^3 due to the

[5] Larger industry datasets have been reported (e.g., [65, 68]) but they are unavailable to the public.

Graph	Model	DGL				P^3				Speedup
		Epoch	DAG	Copy	Compute	Epoch	DAG	Copy	Compute	
OGB-Product	SGCN	4.535	4.051	0.233	0.251	1.019	0.256	0.364	0.399	4.45
	GCN	4.578	3.997	0.253	0.328	1.111	0.248	0.372	0.491	4.12
	GraphSage	4.727	4.056	0.258	0.413	1.233	0.245	0.361	0.627	3.83
	GAT	5.067	4.164	0.271	0.632	1.912	0.248	0.379	1.285	2.65
OGB-Paper	SGCN	9.059	7.862	0.436	0.761	2.230	0.447	0.605	1.178	4.06
	GCN	9.575	8.117	0.461	0.997	2.606	0.457	0.619	1.530	3.67
	GraphSage	9.830	8.044	0.441	1.345	3.107	0.451	0.597	2.059	3.16
	GAT	10.662	8.094	0.462	2.106	5.138	0.462	0.652	4.024	2.08
UK-2006-05	SGCN	6.435	5.682	0.279	0.474	1.481	0.259	0.416	0.806	4.34
	GCN	7.023	6.146	0.282	0.595	1.509	0.252	0.408	0.849	4.65
	GraphSage	7.085	6.005	0.272	0.808	1.880	0.259	0.395	1.226	3.77
	GAT	8.084	6.378	0.330	1.376	3.379	0.234	0.472	2.673	2.39
UK-Union	SGCN	11.472	10.168	0.401	0.903	2.379	0.353	0.597	1.429	4.82
	GCN	12.523	10.815	0.444	1.264	2.864	0.343	0.624	1.897	4.37
	GraphSage	12.481	10.452	0.435	1.594	3.395	0.368	0.619	2.408	3.68
	GAT	13.597	10.693	0.480	2.424	5.752	0.371	0.652	4.729	2.36
Facebook	SGCN	22.264	19.765	0.627	1.872	4.102	0.509	0.907	2.686	5.43
	GCN	24.356	20.673	0.760	2.923	5.624	0.494	1.010	4.120	4.33
	GraphSage	23.936	19.756	0.755	3.425	6.298	0.554	1.027	4.717	3.80
	GAT	24.872	19.472	0.758	4.642	8.439	0.623	0.953	6.863	2.95

Table 4: P^3 is able to gain up to $5.4\times$ improvement in epoch time over DGL. The gains increase with graph size. The table also provides a split up of epoch time into its constituents: computation graph creation (DAG), data copy, and compute. The compute time is the sum of the forward pass, backward pass and update.

need for *pushing* the graph structure, and the overheads associated with additional CUDA calls necessary to push the activations (§3). We remind the reader that caching/replication for P^3 is disabled for this experiment, and enabling it would reduce the data copy time. However, P^3's aggressive pipelining is able to keep the additional overheads in forward pass to a minimum. We also notice that as the model complexity increases, the dominance of computation graph creation phase reduces in the overall epoch time as the forward and backward passes become more intensive.

5.2 Impact of Sampling

In the last experiment, we enabled aggressive sampling, which is a common strategy used by existing GNNs to achieve scalability and load balancing. However, sampling affects the accuracy of the task, and the number of epochs it is necessary to achieve the best accuracy. Further, some GNN architectures may not support sampling, or require more samples (compared to the (25, 10) setting we used). To evaluate how P^3 performs when the underlying task cannot support sampling, we repeat the experiment by disabling sampling. Everything else remains the same. Figure 7 shows the result.

Without sampling, we note that the largest graphs (UK-Union and Facebook) cannot be trained in our cluster. This is because the computation graphs exhaust the memory in the case of DGL and the only way to solve it is to enable sampling. Additionally, for the more complex model (GAT), DGL struggles to train on all datasets. Thus, we do not report

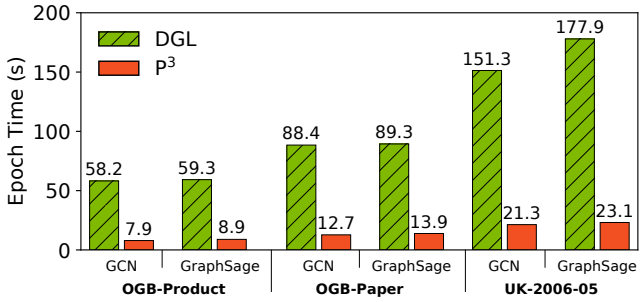
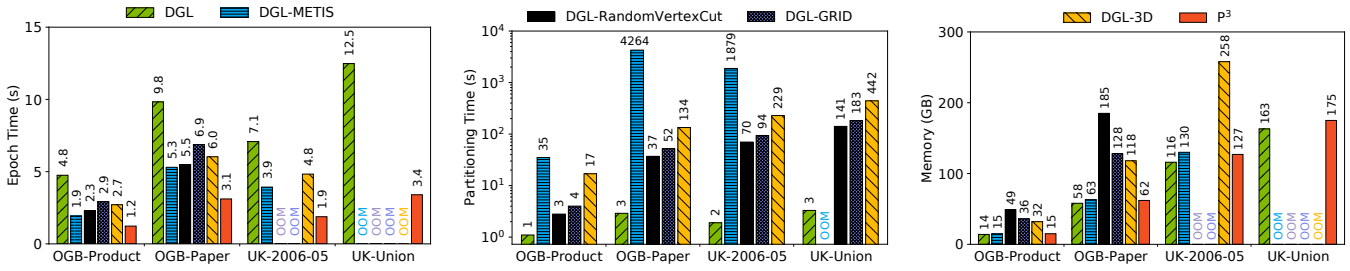


Figure 7: Without sampling, DGL struggles to train complex models and larger graphs. P^3's benefits increase up to 7.69 $\times$.

the results on these two large graphs and for GAT. Even otherwise, we note that P^3's benefits increase compared to the sampled case, with speed ups ranging from $6.45\times$ to $7.69\times$. This clearly indicates the benefits of pulling only the graph structure across the network.

5.3 Impact of Partitioning Strategy

Here, we investigate how different partitioning strategies affect the training time. DGL only supports edge-cut partitioning (using METIS [38]) by default, so we implemented four different partitioning schemes: hash, which is the same partitioner used by P^3, RandomVertexCut [22, 23] and GRID [12, 22] which are vertex-cut partitioners, and 3D, which is the 3-d partitioner proposed in [69]. We train one model, GraphSAGE in DGL with different partitioning strate-



(a) P^3 is able to gain up to 3.8× speed up in terms of epoch time compared to DGL.

(b) Schemes except random hash incur significant overheads in partitioning time (in log scale).

(c) Vertex-cut schemes incur significant memory overheads due to the need to replicate data.

Figure 8: P^3's random hash partitioning outperforms all schemes, including the best strategy in DGL (METIS).

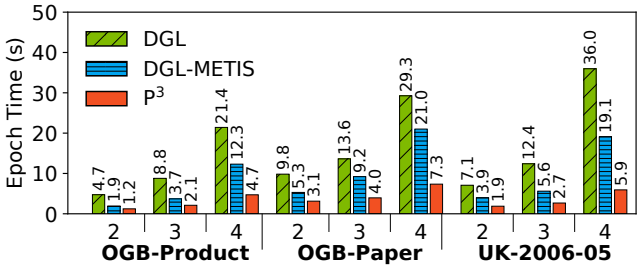


Figure 9: P^3's benefits increase with increase in layers in the GNN. With more layers, partitioning schemes do not help.

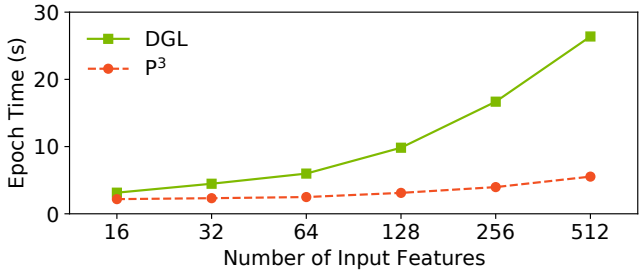


Figure 10: P^3's benefits increase as feature sizes increase, depicting the advantage of not moving features over the network.

gies, and compare against P^3 with its random hash partitioner. We report the average epoch time in fig. 8a.

We notice that P^3's random hash partitioning outperforms all schemes, even the best strategy in DGL (METIS), and the speedups for P^3 ranges from 1.7× (against METIS) to 3.85× (against random hash). The RandomVertexCut, GRID and 3D partitioners run out of memory for larger graphs. The only partitioning scheme that works for the Facebook graph is the random hash partitioner, so we omit it in this experiment. It may be tempting to think that an intelligent partitioner (other than hash partitioner) may benefit DGL. However, this is not true due to two reasons. First, partitioners incur preprocessing time as shown in fig. 8b. We see that METIS incurs the most time, and the overhead is often more than the total training time. It also cannot support large graphs. Other strategies may seem reasonable, but fig. 8c proves otherwise. This figure shows the memory used by various partitioning strategies. It can be seen that vertex cut schemes (RandomVertexCut, GRID, 3D) need to replicate data, and hence incur significant memory overhead. In contrast, not only does P^3's independent partitioning strategy outperform the best performing strategy in DGL (METIS) in terms of memory and epoch time, but it also incurs almost no preprocessing cost.

5.4 Impact of Layers

In this experiment, we evaluate the effect of the number of layers in the GNN. To do so, we pick GraphSAGE and create three different variants of the model, each having different number of layers, from 2 to 4. We then train the model using DGL and P^3. Sampling is enabled in this experiment, as DGL is unable to train deeper models (more layers) even on small graphs without it. The results are shown in fig. 9. We see that P^3's benefits increase with increase in the number of layers, outperforming DGL by up to 6.07×. This is because as the network becomes deeper, the computation graph also grows larger. Further, we see that as the network becomes deeper, the benefits of intelligent partitioning strategies (METIS) start to diminish compared to random hash partitioning. This is due to existing partitioning schemes being optimized for the first hop neighborhood. P^3 is not impacted by either due to its independent partitioning of graph and features and the hybrid parallelism in executing the GNN.

5.5 Impact of Features

To evaluate the impact of feature size on training performance, we vary the number of node features for OGB-Paper dataset from 16 to 512. Since the dataset originally had 128 features, we either prune or duplicate them to obtain the desired number of features. We use GraphSAGE model with sampling for training and report the average epoch time in fig. 10.

We clearly see the benefits of P^3's hybrid parallelism based execution. DGL's performance degrades with the increase in the number of features. This is expected, because to create the computation graph, DGL needs to pull the features, and with more features it incurs more network traffic. In contrast, since P^3 only needs to use network to get the activations, its performance incurs minimal degradation—the epoch time

Graph	Partitions Cached		Memory(GB)	Epoch(s)	Speedup
	Graph	Features			
OGB-Product	1	1	15	1.233	-
	4	4	68	0.724	1.703
OGN-Paper	1	1	62	3.107	-
	4	4	252	1.896	1.639
Facebook	1	1	158	6.298	-
	1	4	362	4.646	1.356
UK-2006-05	1	1	127	1.880	-
	2	2	262	1.524	1.233
UK-Union	1	1	175	3.395	-
	2	1	345	2.748	1.235

Table 5: By caching partitions of graph structure and features independently, P^3 is able to achieve up to 1.7× more performance.

only doubles when the number of features increase by a factor of 32. Here, P^3 outperforms DGL by 4.77×.

5.6 Microbenchmarks

Impact of Caching: In this experiment, we evaluate the benefits of P^3's caching (§3.4). Like in table 4, we use GraphSAGE for training on four graph datasets, but cache the partitions of the graph and features on multiple machines as memory permits. It is interesting to note that for some graphs, it is possible to replicate the structure on multiple machines (e.g., UK-Union) but not features and vice-versa (e.g., Facebook). This shows that independently partitioning the structure and features makes it possible to do caching which was otherwise not possible (i.e., DGL cannot leverage our caching mechanism). Here, P^3 is able to achieve up to 1.7× better performance, and the improvement increases with more caching opportunities. Moreover, caching extends training speedup of P^3 over DGL from 3.6× (in table 4) to 5.23× (here).

Impact of Pipelining: Here we evaluate the benefits of pipelining in P^3 (§3.3). To do so, we use P^3 to train GraphSAGE on four different datasets twice; first with pipelining enabled and then with it disabled. Figure 11 shows that pipelining effectively overlaps most of the communication with computation, and that P^3 is able to extract 30-50% more gains.

GPU Utilization: Figure 12 depicts the peak GPU utilization while training GraphSAGE model on OGB-Product dataset during a five second window for DGL and P^3. Here, the GPU utilization is measured every 50 milliseconds using the `nvidia-smi` [3] utility. We observe that the peak GPU utilization for both DGL and P^3 are similar (≈ 28%). This is due to the nature of GNN models, they perform sparse computations that fail to leverage peak hardware efficiency across all cores[6]. However, we see that during the duration of this experiment, DGL is able to keep the GPU *busy*—keep at least one of the many GPU cores active—for ≈ 20% of the time only. For the remaining ≈ 80%, GPU resources are blocked on the network and thus their utilization drops to zero. On the other hand, P^3 is able to keep the GPU *busy* for ≈ 85% of the time. As a result, it is able to complete 4 epochs of training in the five second duration, compared to 1 in the case of DGL.

[6]Improving peak utilization of accelerators such as GPUs by leveraging the sparsity of the workloads is outside the scope of this work.

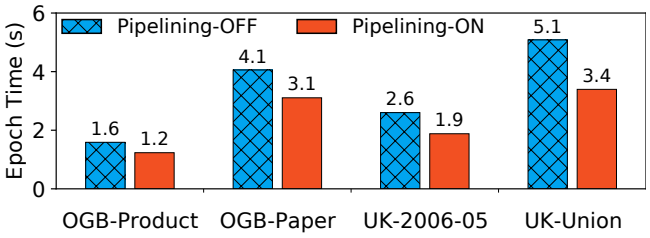


Figure 11: Pipelining boosts P^3's performance by up to 50%.

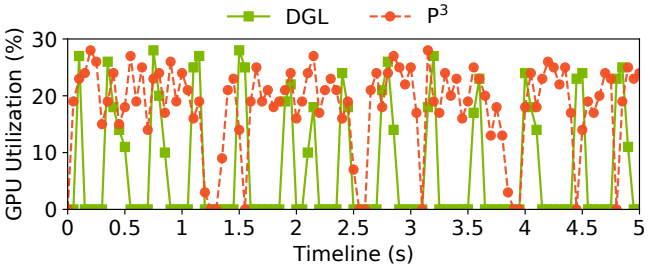


Figure 12: P^3 is able to keep the GPU busy for significantly more time (≈ 85%) compared to DGL (≈ 20%).

5.7 P^3's Scaling Characteristics

Here we evaluate the strong scaling properties of P^3. We again choose the OGB-Paper dataset and train GraphSAGE model on it. To understand the scaling properties, we vary the number of machines, there by varying the number of GPUs used by P^3 and DGL. We report the average throughput (the number of samples processed per second) in fig. 13.

P^3 exhibits near linear scaling characteristics; its throughput doubles when the number of machines (and hence the number of GPUs) are doubled. In contrast, DGL's throughput remains nearly the same as the number of machines increase. This is mainly because GPU resources in DGL are constrained by data movement over network, while P^3 is able to effectively eliminate this overhead using its proposed techniques. As the number of machines continue to grow, we expect P^3 to exhibit less optimal scaling. In P^3, each machine needs to pull activations from all other machines, and this grows linearly with the number of machines resulting in increased data movement that may adversely affect performance. This is a fundamental problem in model parallelism, and hence existing mitigation techniques are directly applicable to P^3.

5.8 Accuracy

Here, we evaluate the correctness of our approach in P^3. To do so, we train GraphSAGE model with sampling, but this time on the OBG-product graph (the smallest graph in our datasets). The best accuracy reported on this graph is approximately 78.2% using about 50 epochs [4]. Due to the lack of published accuracy results for larger graphs, we were unable to repeat this experiment for large graphs in our dataset. We run both DGL and P^3 on this dataset until we obtain the same accuracy

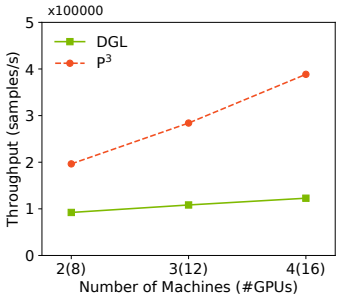


Figure 13: P^3 exhibits graceful and near linear scaling.

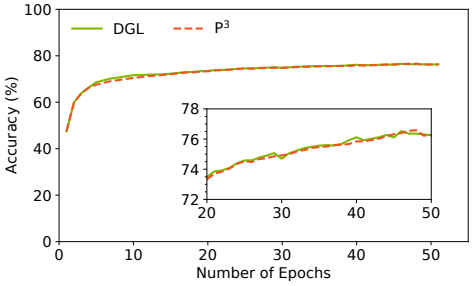


Figure 14: P^3 achieves the same accuracy as DGL, but much faster.

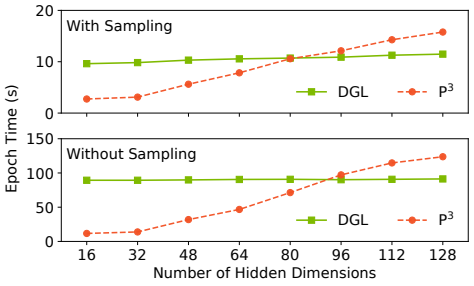


Figure 15: As the number of hidden dimensions increases, benefits of P^3 decreases.

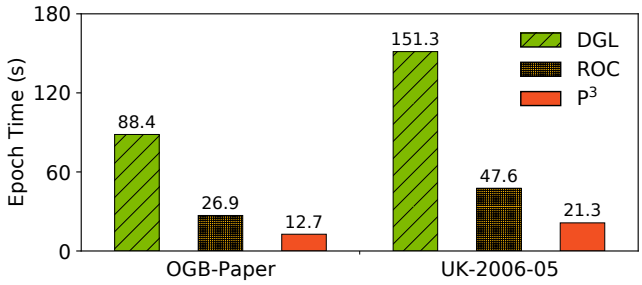


Figure 16: P^3 is able to outperform ROC by up to 2.2×, and its benefits increase with input graph size.

as reported. We show the results in fig. 14. We notice that stock DGL and P^3 both achieve the same accuracy iteration by iteration, and that they both achieve approximately 76.2% accuracy at the end of 51 iterations. P^3 is able to complete this training in 61.65 seconds, while DGL takes 236.37 seconds when using random hash partitioning of the input, and 96.3 seconds when using METIS partitioner. However, METIS takes 35.09 seconds to partition the graph, making the total training time 126.39 seconds. This experiment shows that not only is P^3 able to replicate the same accuracy as DGL thus ensuring its correctness, it is able to complete the training much faster than DGL even for the smallest of the graphs.

5.9 Comparison with ROC

Next we present comparison against ROC. Since ROC does not support sampling, we turn off sampling on all systems. At the time of evaluation, ROC only supported full batch training, and only had implementation for GCN available, so we use that as defaults for this experiment. We run 50 epochs of a 2-layer GCN on OGB-Paper and UK-2006-05 graphs. ROC uses an online partitioner that relies on moving parts of the graphs *during* the execution of the GNN. Due to this, we skip the first few epochs to allow ROC to complete its data movement and measure the average epoch time after that. The result of this experiment is shown in fig. 16.

ROC is able to process both the input graphs significantly faster than DGL due to its highly optimized graph engine. However, P^3 is able to outperform ROC, completing epochs up to 2.2× faster. We also notice that P^3's benefits increase with the size of the input graph. This is due to the fundamental differences in P^3 and ROC's design. While ROC's online partitioner is able to obtain superior partitions based on the access patterns, it still relies on moving features while training the GNN model. As the graph size increases, this results in more features being transferred across the network. In contrast, P^3's design tries to spread the computation of the bottle-necked layer across the cluster and avoids feature movement entirely. Moreover, as the number of layers increase, ROC (and DGL) would need to move exponentially more features, thereby resulting in increased network overhead.

While perusing this result, we wish to remind the reader about few caveats. Our evaluation uses 10 Gbps Ethernet interconnect which favours techniques resulting in lesser data movement. Hence, some of the observed network overheads due to feature movement for ROC (and DGL) can be minimized by using faster interconnects such as InfiniBand. Further, unlike ROC, P^3 and DGL require training data—the graph topology, features, model parameters and activations—to fit in device memory, and failure to do so results in out-of-memory error during training. On the other hand, ROC only requires training data to fit in DRAM, and leverages a cost-based memory manager to selectively move tensors between device memory and DRAM, which may affect performance.

5.10 P^3 Shortcomings

Finally, we present cases where P^3 does not provide benefits. Recall that the fundamental assumption made by P^3 is that the hidden dimensions in GNNs are typically smaller which results in the activations being significantly smaller than features. As this assumption is violated, P^3 starts losing its benefits, and may even incur performance penalties.

To illustrate this, we evaluate the impact of hidden dimensions in this experiment. We train GraphSAGE on the OGB-Product dataset, and fix the number of features to 100. For varying number of hidden dimensions, we record the average epoch time for DGL and P^3 with and without sampling enabled. Figure 15 shows the result. As we expect, the benefits of P^3 decreases as we increase the number of hidden

dimensions (thereby increasing the size of the activations), and P^3 becomes strictly worser than DGL once the hidden dimension size reach close to the feature size. We note that P^3 also incurs additional overhead due to model parallelism, due to which the exact point of transition varies depending on the characteristics of the graph. Dynamically determining whether P^3 would provide benefits in a given scenario and switching appropriately is part of our planned future work.

6 Related Work

Graph Processing Systems Several large-scale graph processing systems that provide an iterative message passing abstraction have been proposed in literature for efficiently leveraging CPUs [12, 21–23, 31, 32, 48, 50] and GPUs [39, 62]. These systems have been shown to be capable of scaling to huge graphs, in order of trillion edges [15]. However, these are focused mainly on *graph analysis and mining*, and lack support for functionalities that are crucial for GNN training, such as auto differentiation and dataflow programming.

Deep Learning Frameworks like PyTorch [5], TensorFlow [6], and MXNet [2] commonly use **Data Parallelism** [44] and **Model Parallelism** [14, 16] to speedup parallel and distributed DNN training. To scale even further, some recent works have proposed combining data and/or model parallelism with pipelining, operator-level partitioning, and activation compression [30, 42, 49, 54]. GPipe [30] and PipeDream [49] are aimed at alleviating low GPU-utilization problem of model parallelism. Both permit partitioning model across workers, allowing all workers to concurrently process different inputs, ensuring better resource utilization. GPipe maintains one weight version, but requires periodic pipeline flushes to update weight consistently, thus limiting overall throughput. PipeDream keeps multiple weight versions to ensure consistency, thereby avoiding periodic flushes at the cost of additional memory overhead. Prior works [37, 60] have even shown how to automatically find fast parallelization strategy for a setting using guided randomized search.

GNN Frameworks Driven by emerging popularity in training GNN models, several specialized frameworks [1, 20, 36, 45, 47, 65, 68] and accelerators [40] have been proposed. They can be categorized in two broad classes: systems [36, 65, 68] which extend existing graph processing systems with NN operations, and systems [1, 20, 45, 47] which extend existing tensor-based deep learning frameworks to support graph propagation operations. Both use graph partitioning as a means of scaling GNN training across multiple CPUs and/or GPUs either in a single machine or over multiple machines. Some frameworks, like AliGraph [65] and AGL [68], only support training using CPUs, while others [1, 20, 36, 45, 47] support performing training on GPUs and use CPU memory for holding graph partitions and exchanging data across GPUs.

PyTorch-Geometric [20] and DGL [1] wrap existing deep learning frameworks with a message passing interface. They focus on designing a graph oriented interface for improving GNN programmability by borrowing optimization principles for traditional graph processing systems and DNN frameworks. However, as we show, they fail to effectively leverage the unique context of GNNs workload and thereby yield poor performance and resource underutilization.

ROC [36] is a recent distributed multi-gpu GNN training system that shares the same goal as P^3, but proposes a fundamentally different approach. It explores using a linear regression model as a sophisticated online partitioner, which is jointly-learned with GNN training workload. Unlike P^3, despite the sophisticated partitioner, ROC must still move graph structure and features over network, which as we show results in high overheads.

PaGraph [45] and NeuGraph [47] are single machine multi-gpu frameworks for training GNNs. PaGraph reports data copy to be a major bottleneck and focuses on reducing data movement between CPU and GPU by caching features of most frequently visited vertices. On the other hand, NeuGraph uses partitioning and a stream scheduler to better overlap data copy and computation. However, in distributed multi-gpu setting, we observe that network communication is a major bottleneck and accounts for a large fraction, up to 80%, of training time while data copy time only accounts for 5%. We note that the proposed techniques in PaGraph and NeuGraph are orthogonal to our work and can only benefit P^3, if applied.

Besides above mentioned system-side optimizations to alleviate scalability bottlenecks, node-wise [24], layer-wise [72], and subgraph-based [13] **sampling techniques** have been proposed. These are orthogonal to and compatible with P^3.

7 Conclusion

In this paper, we looked at the problem of scalability issues in distributed GNN training and their ability to handle large input graphs. We found that network communication accounts for a major fraction of training time and that GPUs are severely underutilized due to this reason. We presented P^3, a system for distributed GNN training that overcomes the scalability challenges by adopting a radically new approach. P^3 practically eliminates the need for any intelligent partitioning of the graph, and proposes *independently* partitioning the input graph and features. It then completely avoids communicating (typically huge) features over the network by adopting a novel *pipelined push-pull* execution strategy that combines intra-layer model parallelism and data parallelism and further reduces overheads using a simple caching mechanism. P^3 exposes its optimizations in a simple API for the end user. In our evaluation, P^3 significantly outperforms existing state-of-the-art GNN frameworks, by up to 7×.

Acknowledgements

We thank our shepherd, Chuanxiong Guo, all the anonymous OSDI reviewers and Ramachandran Ramjee for the invaluable feedback that improved this work. We also thank the contributors and maintainers of PyTorch and DGL frameworks.

References

[1] Deep Graph Library. https://www.dgl.ai/.

[2] MXNet. https://mxnet.apache.org/.

[3] NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[4] Open Graph Benchmark Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/.

[5] PyTorch. https://pytorch.org/.

[6] TensorFlow. https://www.tensorflow.org/.

[7] Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

[8] Davide Bacciu, Federico Errica, and Alessio Micheli. Contextual graph Markov model: A deep and generative approach to graph processing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 294–303, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[9] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[11] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 257–266, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[17] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, pages 3844–3852. Curran Associates, Inc., 2016.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[19] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo. Generating synthetic social graphs with darwini. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 567–577, 2018.

[20] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[21] Swapnil Gandhi and Yogesh Simmhan. An Interval-centric Model for Distributed Computing over Temporal Graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1129–1140, 2020.

[22] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, and Ion Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[24] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 1024–1034. Curran Associates, Inc., 2017.

[25] William Hamilton. Graph Representation Learning Book. https://www.cs.mcgill.ca/~wlh/grl_book/.

[26] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin*, page arXiv:1709.05584, September 2017.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[28] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[29] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv e-prints*, page arXiv:2005.00687, May 2020.

[30] 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*, pages 103–112, 2019.

[31] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[32] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: Efficient ad-hoc analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355. USENIX Association, April 2021.

[33] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 776–789. IEEE Press, 2018.

[34] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020.

[35] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 947–960, USA, 2019. USENIX Association.

[36] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198, 2020.

[37] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[38] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[39] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Kevin Kiningham, Christopher Re, and Philip Levis. GRIP: A Graph Neural Network Accelerator Architecture. *arXiv e-prints*, page arXiv:2007.13828, July 2020.

[41] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017.

[42] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv e-prints*, page arXiv:1404.5997, April 2014.

[43] Jurij Leskovec. *Dynamics of Large Networks*. PhD thesis, USA, 2008.

[44] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association.

[45] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Yu-Chen Lo, Stefano E. Rensi, Wen Torng, and Russ B. Altman. Machine learning in chemoinformatics and drug discovery. *Drug Discovery Today*, 23(8):1538 – 1546, 2018.

[47] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[49] 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 '19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 2311–2320, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321. USENIX Association, July 2020.

[53] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. Estimating node importance in knowledge graphs using graph neural networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 596–606, New York, NY, USA, 2019. Association for Computing Machinery.

[54] 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 '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[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 e-prints*, page arXiv:1909.08053, September 2019.

[57] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 243–246, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688 – 702.e13, 2020.

[59] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[60] 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 '19, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv e-prints*, page arXiv:1909.01315, September 2019.

[62] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8), February 2016.

[63] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks, 2019.

[64] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[65] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 3165–3166, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 974–983, New York, NY, USA, 2018. Association for Computing Machinery.

[67] 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 in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[68] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, August 2020.

[69] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, Savannah, GA, November 2016. USENIX Association.

[70] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5165–5175. Curran Associates, Inc., 2018.

[71] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2019.

[72] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 11249–11259. Curran Associates, Inc., 2019.