ScalaCache: Scalable User-Space Page Cache Management with Software-Hardware Coordination

Li Peng and Yuda An, *Peking University;* You Zhou, *Huazhong University of Science and Technology;* Chenxi Wang, *University of Chinese Academy of Sciences;*
Qiao Li, *Xiamen University;* Chuanning Cheng, *Huawei;*
Jie Zhang, *Peking University and Zhongguancun Laboratory*

https://www.usenix.org/conference/atc24/presentation/peng

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

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

ScalaCache: Scalable User-Space Page Cache Management with Software-Hardware Coordination

Li Peng[1], Yuda An[1], You Zhou[3], Chenxi Wang[4], Qiao Li[5], Chuanning Cheng[6], Jie Zhang[1,2]
National Key Laboratory for Multimedia Information Processing,
School of Computer Science, Peking University[1]
Zhongguancun Laboratory, Beijing, China[2], Huazhong University of Science and Technology[3]
University of Chinese Academy of Sciences[4]
Xiamen University[5], Huawei[6]
https://www.chaselab.wiki

Abstract

Due to the host-centric design principle, the existing page cache management suffers from CPU consumption, communication costs, and garbage collection (GC) interference. To address these challenges, we propose ScalaCache, a scalable user-space page cache with software-hardware coordination. Specifically, to reduce the host CPU overhead, we offload the cache management into computational storage drives (CSDs) and further merge the indirection layers in both the cache and flash firmware, which facilitates lightweight cache management. To further boost scalability, we build a lockless resource management framework that allows multiple CSD internal cores to manage the cache space concurrently. ScalaCache also aggregates the computing power of multiple CSDs to deliver scalable I/O performance. Moreover, ScalaCache reduces communication costs by trimming the I/O control path while mitigating GC interference via a GC-aware replacement policy, thereby enhancing its efficiency and performance stability. Our evaluation results reveal that ScalaCache exhibits 5.12× and 1.70× bandwidth improvements, respectively, compared to kernel page cache and the state-of-the-art user-space one. ScalaCache is open source and available at https://github.com/ChaseLab-PKU/ScalaCache.

1 Introduction

The storage software stack, primarily consisting of a page cache management module (called *page cache manager*) [43, 54, 73] and an I/O engine [22, 42, 88], has been widely adopted in diverse computing domains such as databases, cloud computing, and high-performance computing [8, 30, 83], as it narrows down the performance gap between processors and storage devices. The page cache manager shortens the I/O latency by buffering hot data with the high temporal locality in main memory [40, 61], while the I/O engine employs a rich queue mechanism to concurrently access data residing in the underlying storage media [3, 9].

Storage media (e.g., NAND flash) have undergone significant technology shifts. For example, the emerging flash-based solid state drives (SSDs) have surpassed 14 GB/s I/O bandwidth [2, 17, 52], whereas the kernel storage software stack (cf. Figure 1a), unfortunately, fails to follow up on the performance boost. Specifically, its page cache manager, residing in the kernel space, necessitates user-kernel context switches and global locking, which slows down the I/O services [30, 100]. On the other hand, the kernel I/O engine employs an intricate interrupt service routine (ISR) to handle massive I/O responses, which further prolongs the I/O latency [71, 84].

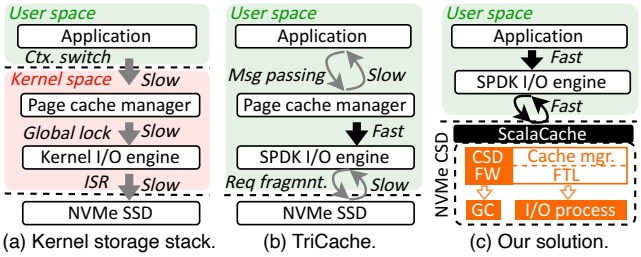


Figure 1: Comparison of representative storage stacks.

To address the aforementioned limitations, recent works have explored moving the storage stack into user space [22, 99]. In particular, the SPDK I/O engine [88] integrates a polling-based NVMe driver into user space and further employs asynchronous concurrency and lock-free technologies to eliminate the kernel I/O engine overheads. Building atop SPDK, *TriCache* [15] customizes a lock-free page cache manager in the user space (cf. Figure 1b), where multiple *cache manager threads* are used for cache management, improving I/O performance by 3.84× over the kernel page cache.

However, these page cache managers exclusively reside on the host, overlooking storage taxes levied by the *host-centric page cache manager design*. Specifically, this unilateral design imposes three sources of taxes. *(1) CPU tax*: These page cache managers heavily tax the host CPU resources, depriving applications of precious computing resources [35, 79]. For example, TriCache dedicates multiple host CPU threads per SSD to undertake index operations and cache replacement, reflecting its heavyweight. *The accumulated CPU consumption is exacerbated as the number of SSDs scales up, high-*

lighting the poor scalability. *(2) Communication tax*: The host-centric page cache managers necessitate an I/O engine to communicate with SSDs. Unfortunately, the costly ISR, context switching, and global locking in the kernel I/O engine impede this communication. While TriCache adopts the lightweight SPDK I/O engine, it allocates cache manager threads to communicate with both applications and SSDs. This tripartite structure [32] prolongs the I/O path compared to the traditional producer-consumer model [11], exhibiting poor efficiency. In other words, *cache manager threads sit on the critical I/O path*. Moreover, to harness their parallelism, I/O requests are split and distributed among them (i.e., fragmentation), worsening the manager-SSD communication issue. Our experiment shows that the queuing latency due to communication accounts for 77.74% of the total I/O latency. *(3) Interference tax*: Multiple software layers of the storage stack (e.g., I/O engine) sit between the host-centric page cache manager and SSDs, hindering the cache manager from detecting SSD internal activities such as GC. *Such software isolation leads to interference between GC and regular I/O requests, compromising performance stability*.

To address the aforementioned challenges, our key insight is that the host memory buffer (HMB) feature in NVMe [3,24] and the computational storage drives (CSDs) with ample resources [28,87] present a promising opportunity to offload the page cache manager, which can overcome these taxes. The HMB feature allows SSDs to directly manage cached data in main memory while also ensuring rapid data accessibility for applications. The task offloading scheme eliminates redundant functionalities between cache management and CSD firmware (e.g., index operations), making the cache manager lightweight enough to fit into CSDs. Consequently, the taxed host CPUs are freed up for applications' use. This in-storage processing solution can also enhance scalability without host CPU reliance while slashing communication and interference taxes via coordinating cache management and CSD internals.

Thus, we propose *ScalaCache*, a user-space page cache manager with software-hardware coordination (cf. Figure 1c). Specifically, towards successful offloading to reduce the CPU tax, ScalaCache tightly integrates a lightweight cache manager into the CSD firmware, which consolidates their indirection layers (i.e., cache indexing and SSD FTL [37, 89]) to simplify the redundant address translations. To further enhance scalability, ScalaCache builds a lockless resource management framework in CSDs, allowing multiple embedded cores to manage the cache concurrently. It further aggregates the computing power of multiple CSDs to deliver scalable I/O performance without relying on the host CPU. To mitigate the communication tax, it allows applications to directly access the cache manager and CSDs by integrating the cache manager into CSDs, thereby removing the manager-SSD communication inherent to the tripartite structure. Additionally, cache replacement, which fetches or evicts non-contiguous pages, compounds manager-SSD communication due to the NVMe command requirement for contiguous addresses. To alleviate this, ScalaCache bundles multiple pages into a single NVMe command, which causes only one communication. For the interference tax, the cache interface (i.e., pin interface [63]) situates page writebacks on the critical I/O path, where GC unintentionally stalls these writebacks. ScalaCache mitigates this by introducing a GC-aware replacement policy, which preferentially reclaims clean pages to prevent GC from delaying writebacks. Our evaluation results reveal that ScalaCache improves bandwidth by 5.12× and 1.70× compared to the kernel page cache and TriCache, respectively.

The main **contributions** can be summarized as follows:

• *User-space cache with software-hardware coordination:* ScalaCache reaps the benefits of both user-space design and software-hardware coordination. Specifically, our design incorporates the user-space SPDK I/O engine and follows its design principles like lockless, resulting in a 78.13% latency reduction. Coordination further slashes communication and interference taxes. The offloading scheme trims the I/O path by allowing applications to access CSDs directly and alleviates communication burdens by enhancing NVMe commands with cache activity awareness, which packs multiple I/O requests into one command for batch processing. Moreover, by exposing GC state to the page cache manager, we customize a GC-aware replacement policy to mitigate GC disruption.

• *Customizing lightweight cache management in CSDs:* We propose *FusionFTL* to address challenges of delegating cache management to CSDs, which struggle with management burdens. FusionFTL is a lightweight index structure that combines the cache index and the FTL mapping table, facilitating efficient address translation. We further incorporate cache operations (i.e., fetches and evictions) into I/O processing, allowing the CSD to handle multiple cache operations simultaneously in a single I/O request, eliminating the traditional need for two separate I/O requests. These lightweight designs enable successful cache offloading, which in turn reduces host CPU usage by 77.75%.

• *Enabling concurrent I/O processing for CSDs:* Accessing critical resources (e.g., cache space) by multiple CSD cores may pose a potential lock penalty. To address this issue, we build a lockless resource allocation framework within CSDs. Specifically, we assign various resources of CSDs to each core based on demand. As these resources are private to each core, cores can access them concurrently, which enhances scalability. To further extend scalability across multiple SSDs, we build a parallel processing model that organizes CSDs into a CSD array, which aggregates all their computing power.

2 Background

2.1 SSD Architecture

Figure 2 depicts a typical SSD architecture, which encompasses a storage backend and a computation frontend [16, 97].

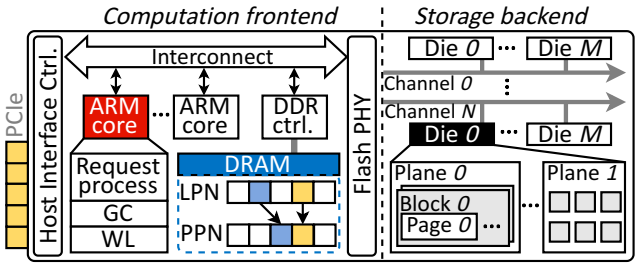
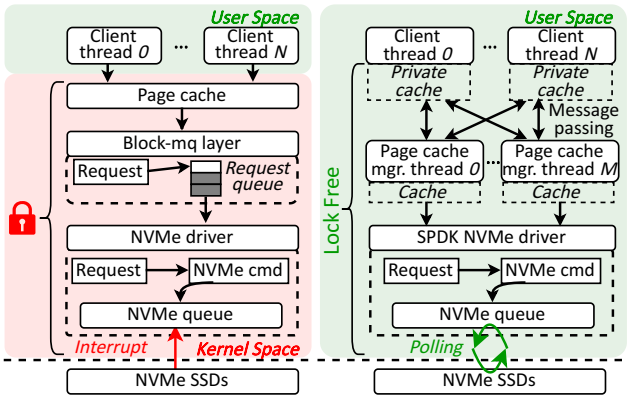


Figure 2: SSD internal architecture.

Storage backend. The storage backend primarily contains multiple flash dies, which are organized as 4∼16 flash channels [49]. Each flash die consists of 2∼4 planes. A flash plane comprises hundreds to thousands of blocks, each containing hundreds of pages. This organization capitalizes on the parallelism of flash, thereby delivering superb I/O throughput [95]. The flash intrinsic necessitates flash block erasure before data can be overwritten [86]. To address this, the SSD constructs an indirection layer called the flash translation layer (FTL) [65,81], which maintains a mapping table to record the mapping between the host logical address (i.e., LPN) and flash physical address (i.e., PPN). For overwrites, SSD allocates new PPNs from pre-erased blocks, updates the mapping table, and then invalidates the stale flash pages. When overwrites exhaust free blocks, garbage collection (GC) is triggered to reclaim used blocks [86], which entails selecting victim blocks based on the wear-leveling (WL) strategy, migrating all valid pages to free blocks, and then erasing the victim blocks. Note that GC invokes extensive flash reads and writes, which significantly delay incoming I/O requests [64,82,96].

Computation frontend. The computation frontend consists of a host interface controller, an embedded DRAM, and multiple ARM cores. Specifically, DRAM serves to store the FTL mapping table, while the ARM cores offer the necessary computing power to execute the aforementioned mechanisms (e.g., FTL, GC, and WL). These cores handle I/O requests from the host interface controller, forward them to the storage backend through the flash physical layer (PHY) [69], and subsequently respond to the host. In addition, the host interface controller connects to the host via a high-performance communication protocol (e.g., NVMe [3]), which enhances I/O parallelism through a rich queue mechanism. The latest version of NVMe protocol also releases several features to enhance storage functionality. A notable one is the host memory buffer (HMB) [3,24], which allocates a portion of host memory to the SSD to buffer data. This feature is beneficial for consumer-grade and cost-efficient enterprise SSDs, as it enables these products to buffer internal metadata (e.g., FTL mapping table) on the host, thereby reducing the heavy capacity demand on the SSD internal DRAM [25,65]. To manage the buffered metadata, the SSD interacts with the HMB region via extra DMA, which incurs a minor latency (typically a few microseconds) [24]. Moreover, to ensure the integrity of the metadata, SSDs employ vendor-specific fault-tolerance



(a) Kernel storage software stack. (b) User-space storage software stack.

Figure 3: Typical kernel and user-space software stacks.

mechanisms. Note that the default HMB feature limits the maximum memory allocation to 128 MB [42], which constrains further advancements.

In-storage processing (ISP). In-storage processing [28], as a new computing paradigm that offloads data-intensive tasks directly to the underlying SSDs, diminishes the inefficiencies imposed by data movements and eases the computational burden on the host. Computational storage drives (CSDs), a typical ISP implementation, integrate multiple powerful ARM cores and augment DRAM capacity [6,26,50,74–76,92]. For example, a recent high-performance SSD controller boasts in excess of 10 ARM cores and 8 GB memory [6,50]. These substantial computing and memory resource enhancements empower CSDs to efficiently process the offloaded tasks while simultaneously serving numerous I/O requests. Many prior works [12,39,60] have explored the feasibility of ISP to accelerate complicated applications, including large-scale graph processing and recommendation inference [51,80]. Moreover, several studies [21,62] propose to offload intricate file systems for direct access. Nevertheless, research endeavors focusing on the storage tax associated with the page cache manager remain in their infancy [96].

2.2 Kernel Storage Software Stack

The kernel storage software stack (cf. Figure 3a) includes the I/O engine [42] and page cache management module [43].

Kernel I/O engine. The kernel I/O engine encompasses a multi-queue block layer (blk-mq) and an NVMe driver. The block layer buffers I/O requests from the page cache management module in the request queues, where they undergo scheduling to optimize performance before being passed to the NVMe driver. The driver places these requests into the NVMe queues and informs the underlying SSD of their arrival. Upon request completion, the SSD sends a message signaled interrupt (MSI) to the host, which triggers the interrupt service routine (ISR) [10,78] to finish the post-processing of the requests. Note that context switching and ISR within the

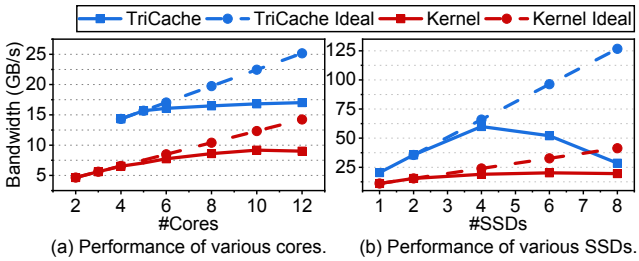


Figure 4: Performance analysis.

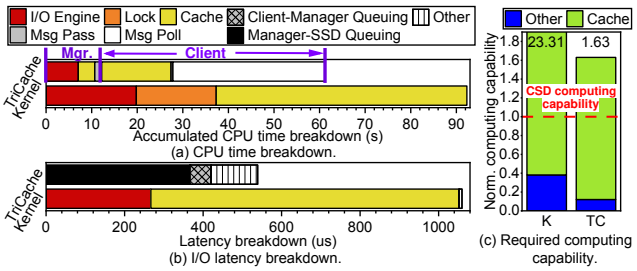


Figure 5: Breakdowns and required computing capability.

interrupt processing lead to increased I/O latency [84].

Kernel page cache management module. This kernel module, a typical host-centric cache management design, accommodates hot data at page granularity (i.e., 4 KB) to facilitate fast data retrieval when requested. For simplicity, we use the terms "page cache manager" and "page cache management module" interchangeably. To enable a rapid lookup of the buffered page location, the page cache manager employs a radix tree index [7] to map the LPNs to their memory addresses (i.e., page frame address). Upon successful lookup of the page frame address from the index structure (i.e., a cache hit), it reads or overwrites the buffered data based on the I/O request [94]. Otherwise, a cache miss triggers the kernel I/O engine to fetch missing pages from the SSDs. When cached pages occupy excessive memory, the page cache manager frees up memory by evicting cold pages back to the SSD. Note that the global lock mechanism in this module, which serializes cache access and management in a multi-threaded scenario, can severely hinder performance, especially when employing high-performance SSDs [58, 61]. In addition, the sole software design on the host side fails to detect the GC activity in the underlying SSD, leading to blocked page eviction [96].

2.3 User-space Storage Software Stack

To address the kernel space design issues, prior works [15, 22, 88, 99] explore the user-space storage stack (cf. Figure 3b).

SPDK I/O engine. SPDK [88] is a high-performance user-space I/O engine customized for NVMe SSDs that encapsulates several strategies to enhance performance. Specifically, it integrates the NVMe driver into the user space. This key shift allows applications to directly access the SSDs without kernel involvement, effectively obviating the context switching overhead. It also embraces a polling-based model for I/O completion, ensuring prompt processing of I/O responses to minimize I/O latency [71, 84]. In addition, it adopts a lock-free design principle throughout the entire I/O engine, which significantly improves the parallelism of I/O processing [66, 100]. Lastly, SPDK incorporates techniques like asynchronous concurrency to further enhance performance.

TriCache. The prior work TriCache [15] proposes a user-space page cache manager built on top of SPDK. To be specific, TriCache inherits the host-centric page cache manager design, which utilizes multiple CPU threads, referred to as *page cache manager threads*, to conduct cache management. Each page cache manager thread manages an exclusive portion of the logical address space, which is partitioned based on a page-grained hash algorithm. Based on its logical address, an I/O request from an application thread (i.e., *client thread*) is split and distributed to different page cache manager threads to harness thread-level parallelism via message passing. The page cache manager thread serves these requests from the internal index if the requested pages are buffered. Otherwise, the page cache manager thread retrieves these missing pages from the SSD via SPDK. This tripartite structure extends the I/O control path as client threads interact with SSDs through page cache manager threads first. Additionally, the page-grained address space partition leads the page cache manager thread to fetch pages with non-contiguous addresses, preventing SPDK from merging small I/O requests into large ones (i.e., request fragmentation), thereby increasing the manager-SSD communication overhead. This communication overhead also extends to page evictions. To accelerate lookups, TriCache adds a private cache index within each client thread to buffer page frame addresses, avoiding additional client-manager communication unless a private cache miss occurs.

3 Motivation and Challenges

3.1 Preliminary Study

We examine a macro-benchmark (i.e., `mds` [53]) to analyze the aforementioned storage software stack designs: the kernel storage software stack (`Kernel`) and TriCache (`TriCache`). Please refer to Section 6 for the detailed experiment setup. We evaluate their throughput by varying the number of CPU cores and SSDs. For TriCache, we dedicate two CPU cores as page cache manager threads for each SSD. Both `Kernel` and `TriCache` are compared with their ideal cases (`Kernel Ideal` and `TriCache Ideal`) that are mathematically estimated by scaling the bandwidth of `Kernel` and `TriCache` perfectly linearly with the number of cores or SSDs.

Analysis of kernel software stack. As depicted in Figure 4a, `Kernel` degrades performance by 36.76% compared to `Kernel Ideal` when using 12 cores. It also fails to achieve scalability on multiple SSDs, as evidenced by a 52.54% performance gap when utilizing 8 SSDs (cf. Figure 4b). To figure

out the reason behind the performance gap, we analyze the accumulated CPU time when using 8 cores and one SSD (cf. Figure 5a), decomposing it into the time spent in the kernel I/O engine (`I/O Engine`), lock contention (`Lock`), and the kernel page cache manager (`Cache`). The analysis reveals that `I/O Engine` and `Lock` consume 21.45% and 18.98% of the total CPU time, respectively, indicating that they limit performance. The latency breakdown (cf. Figure 5b) shows that `I/O Engine` and `Cache` account for 25.22% and 74.07% of the total I/O latency, respectively, which also indicates their inefficiency.

Analysis of TriCache. While `TriCache` exhibits 89.13% higher throughput than `Kernel` through the lockless design and the user-space implementation, its performance cannot scale as the number of cores increases (cf. Figure 4a). In particular, when using 12 cores, `TriCache` degrades the performance by 32.33% compared with `TriCache Ideal`. To figure out the root cause of the tremendous performance gap, we analyze the breakdown of the accumulated CPU time and I/O latency, respectively, when testing with 8 client threads.

As shown in Figure 5a, we categorize the CPU time of page cache manager threads into the SPDK (`I/O Engine`), cache management (`Cache`), and message passing (`Msg Pass`), while dividing the CPU time of the client threads into client private cache (`Cache`), message passing (`Msg Pass`), and polling the message queues (`Msg Poll`). The breakdown reveals that `Msg Pass` accounts for a considerable 10.08% of the manager's CPU time due to receiving extensive client requests. To handle these requests, the manager threads consume 30.5% CPU time for cache management. Moreover, the request fragmentation (cf. Section 2.3) forces the page cache manager threads to issue 6.72× more NVMe commands than the number of I/O requests in the workload. The increased NVMe commands further amplify the CPU overhead of `I/O Engine` (cf. Section 6.3). These communication costs decrease the page cache manager's processing capability, which in turn forces the client threads to spend 68.14% CPU time waiting for page cache manager responses (i.e., `Msg Poll`).

For the latency breakdown (cf. Figure 5b), we decompose total I/O latency into the queuing latency for client-manager communication (`Client-Manager Queuing`), the queuing latency for manager-SSD communication (`Manager-SSD Queuing`), and others (`Other`). `Client-Manager Queuing` originates where page cache manager threads process requests sequentially in the message queue, as each request must wait for its predecessors to be handled. Additionally, requests also suffer from `Manager-SSD Queuing` when awaiting the page cache manager to forward them to the SSD. These queuing latencies dramatically increase when massive requests need to be processed due to request fragmentation. `Client-Manager Queuing` and `Manager-SSD Queuing` account for 10.02% and 67.72% of the total latency, respectively, revealing that the communication tax inherent to the tripartite structure leads to considerable performance degradation.

In multi-SSD testing, the performance gap between `TriCache` and `TriCache Ideal` becomes more severe (e.g., 77.51% when employing 8 SSDs), which is shown in Figure 4b. This gap can be attributed to the heavy CPU occupancy introduced by page cache manager threads (e.g., occupying 16 cores when using 8 SSDs). Such CPU occupancy diminishes the computing resources available for client threads, which in turn impairs the overall performance. The poor scalability highlights the unavoidable host CPU tax that stems from the host-centric page cache manager design.

3.2 Challenges and Key Insight

Through prior analysis, we summarize the challenges arising from host-centric page cache manager designs as follows:

• **CPU consumption:** Kernel page cache and TriCache both consume excessive computing resources to execute cache operations. While TriCache embeds a private cache in client threads to minimize this overhead, the page cache manager threads are still heavyweight, as they impose a substantial CPU load for the I/O engine and address translation. Moreover, the CPU reliance of these host-centric designs is exacerbated as the SSD count scales up, posing a significant challenge to scalability.

• **Communication cost:** The ponderous kernel I/O engine impedes efficient communication between the kernel page cache and SSDs. Additionally, the inherent tripartite structure of TriCache necessitates frequent communication between client and page cache manager threads, as well as between page cache manager threads and SSDs, making the page cache manager threads sit on the I/O critical path.

• **GC interference:** The GC activity in SSDs inadvertently blocks cache management, diminishing the stability of I/O services. Unfortunately, with multiple software layers isolating the page cache manager from SSDs, these host-centric designs lack the necessary knowledge to mitigate the GC interference [82, 96].

Existing host-centric designs struggle with these challenges. However, as our analysis shows, sole software optimizations are insufficient to eliminate these taxes if they adhere to the host-centric design principle. Our key insight is that offloading the page cache manager into CSDs offers a promising opportunity to address these challenges. Specifically, the HMB feature enables CSDs to manage the host memory directly, making it feasible to undertake cache management. The offloading scheme not only eliminates host CPU reliance but also inherently achieves scalability with multiple CSDs by aggregating compute resources within each CSD. Note that CSD internal mechanisms similar to cache indexing (e.g., the FTL mapping table) can potentially facilitate a lightweight page cache manager design. Meanwhile, software-hardware coordination can mitigate communication and interference taxes by coordinating cache management and request handling within CSDs.

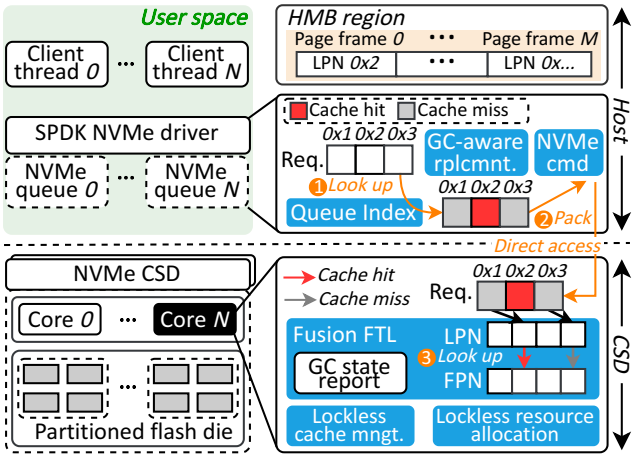


Figure 6: Overview of ScalaCache.

We further analyze the computing capability demands of these page cache manager designs to assess the feasibility of this offloading (cf. Figure 5c). Specifically, we evaluate Dhrystone million instructions per second (DMIPS) [31, 90], a unified computing capability metric suitable for processors with different ISAs (e.g., host x86 CPU and CSD ARM processor), needed by their cache management. This evaluation is under the same configuration as outlined in Section 3.1. The results indicate the kernel page cache (`K`) is highly demanding on the CSD (i.e., 22.31×), while TriCache (`TC`) shows promising potential for offloading. However, TriCache still requires 63% more computing capability than CSD supplies [50, 93], indicating the need for a more lightweight design.

4 Overview of ScalaCache

To mitigate the heavy storage taxes analyzed above, we propose *ScalaCache*, a software-hardware coordination for user-space page cache manager, which achieves lightweight and scalable cache management with efficiency and stability. Serving as a block cache [15, 98], ScalaCache builds a logical address-based cache to accelerate data access to storage devices. Figure 6 shows the overview of ScalaCache.

Lightweight. ScalaCache exploits the HMB feature to enable cache management for the underlying CSD. However, directly delegating this management to the CSD is impractical due to the intricate computational and memory demands. Notably, both the page cache manager and CSD employ an indirection layer to figure out data location (i.e., address translation). Consequently, we propose a high-performance and memory-efficient index structure called *FusionFTL*, which tightly integrates the cache index structure into the FTL mapping table. FusionFTL allows the fast lookup of both page frame and flash addresses in a single memory shot. It also consolidates the metadata of the page cache manager and the CSD, significantly reducing memory overhead. To further reduce the computational demand, page eviction and fetching are incorporated into the CSD internal request processing. Specifically, the offloaded page cache manager can evict and load pages in a single I/O request rather than two separate flush and fetch operations in traditional cache manager designs, thus mitigating the processing overhead. Furthermore, to prevent potential CSD overloading due to numerous requests, we add *Queue Index* within each client thread to buffer page frame addresses for cached pages. As a result, client threads can rapidly access the Queue Index for cached pages, significantly reducing the number of requests to the CSD.

Scalability. To prevent the potential lock issue from constraining scalability, we design a lockless resource allocation framework for CSDs. Specifically, we partition resources (e.g., the FTL mapping table and the cache space) by CSD core, which allows concurrent access without lock-based synchronization. This partitioning extends to the NAND flash, which leverages the hardware isolation offered by the flash die to negate inter-core interference. Note that the host CPU reliance of existing page cache manager designs limits the scalability across multiple SSDs. By offloading cache management, we remove such CPU reliance and further enhance scalability via computing power aggregation. Specifically, we build a parallel processing model to enable multiple CSDs to autonomously handle their requests, thereby leveraging the aggregated computing power to maintain scalability.

Efficiency and stability. To reduce communication costs, we trim the I/O path by removing page cache manager threads from the I/O critical path. Specifically, with cache management delegated to CSDs, client threads can directly access the cache via the SPDK I/O engine, avoiding additional communication with page cache manager threads. Furthermore, to mitigate the communication of fetching or evicting non-contiguous pages, we pack multiple page information into a single NVMe command, which only incurs one communication. To improve stability, we first expose the GC state inside the CSD to client threads by piggybacking it with NVMe commands. Subsequently, we design a GC-aware replacement policy in client threads to prioritize the reclamation of clean pages, preempting the stalling of dirty page writebacks.

Building upon all key components of ScalaCache, we describe the detailed I/O path of the entire cache management from top to down. In particular, when the client thread initiates an I/O request, it searches the target pages within its internal Queue Index (❶ in Figure 6). If the client thread locates the pages successfully (i.e., cache hits), the client can access them directly. Otherwise, to fetch the missing pages, it packs corresponding retrieval operations into a single NVMe command and then forwards the command to the offloaded cache management within the CSD (❷). Upon receiving the NVMe command, the CSD leverages FusionFTL to perform rapid address translation for these requested pages (❸). For those cached pages, the CSD returns their page frame addresses to the client thread. Otherwise, the CSD reads the missing pages from NAND flash and buffers them in free page frames.

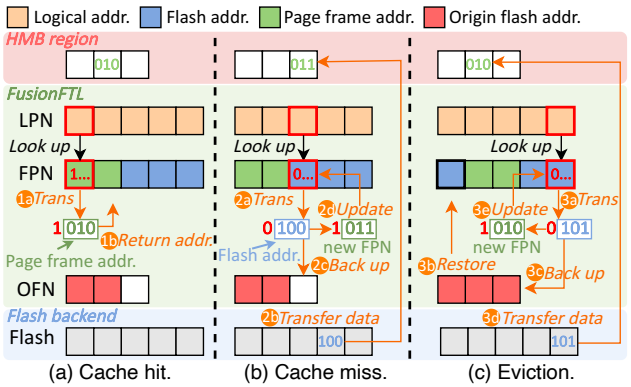


Figure 7: Details of FusionFTL.

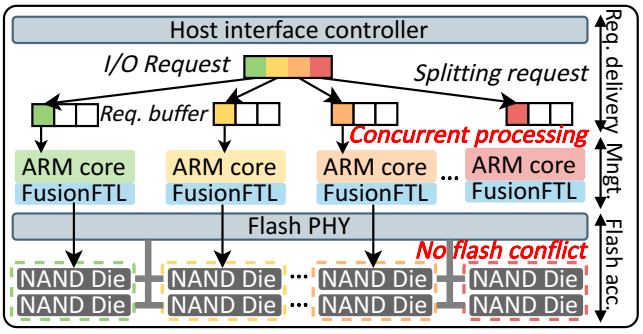


Figure 8: Concurrent I/O processing within CSD.

5 ScalaCache Design

5.1 Cache Management Offloading

Cache space allocation. Our key insight is that the HMB feature allows CSDs to directly manage pre-allocated host memory. However, the current SPDK lacks support for this feature, and the default HMB feature restricts the allocated memory to 128 MB. To address these limitations, we enhance SPDK with HMB support and extend its initialization to allocate adequate memory. Specifically, the host uses the SPDK memory API [88] to allocate sufficient DMA-able user-space memory (e.g., more than 96GB) as cache space based on the demand of the CSD. Memory addresses are conveyed to the CSD, allowing it to manage cache via DMA.

FusionFTL. Figure 7 shows the details of FusionFTL. Noting that both the page frame address (for cached data) and the flash address (for uncached data) describe the data residing location, we combine these two addresses into a unified address referred to as *Fusion Page Number* (FPN). This unification allows us to merge the FTL mapping table and the cache index into a compact index structure (i.e., FusionFTL) that maps LPN to FPN. The most significant bit of the FPN is used to distinguish between page frame and flash addresses (i.e., 1 for page frame address and 0 for flash address). To revert FPNs of cached pages to their original flash addresses upon page eviction, we incorporate the *original flash address* (i.e., OFN) to record the flash address for each cached page.

To enable lock-free cache management, we shift the cache access interface to the pin/unpin interface [5, 34, 63], which employs reference count to track page frame access. Based on the reference count, the page cache manager can safely evict pages not in use by any client thread, thus eliminating locks existing in kernel page cache that avoid evicting pages being accessed. This interface also eliminates unnecessary writebacks for dirty pages that are recently accessed [30, 43], which not only eases communication with the CSD but also lightens its load. After shifting this interface, the CSD handles pin and unpin requests from client threads.

For unpin requests, the CSD looks up the target pages in FusionFTL to decrease their reference counts. For pin requests, the CSD looks up the FusionFTL first. If a cache hit occurs, as depicted in Figure 7a, the CSD directly translates LPNs to the page frame addresses (1a), updates the reference counts, and returns the addresses to the client thread (1b). Otherwise, the CSD converts LPNs to flash addresses (2a and 3a) and directly fetches flash pages. If sufficient free page frames are available (cf. Figure 7b), the CSD buffers the fetched flash pages into these frames (2b) and then updates the FusionFTL, which involves recording the original flash addresses in the OFNs (2c) and then writing new page frame addresses into the FPNs (2d). Afterward, the CSD returns the page frame addresses to the client thread.

When no free page frames exist, the CSD evicts cached pages to reclaim page frames (cf. Figure 7c). FusionFTL employs a clock replacement policy [72] to evict pages with zero reference counts. While clean pages are simply evicted without writeback, their FPNs require restoring to the original flash addresses, ensuring correct flash page retrievals for future requests. Hence, the CSD updates their FPNs with the OFNs (3b). Conversely, dirty pages must be written back to the flash before eviction. Thus, their OFNs are discarded, and their FPNs are updated with new flash addresses. Afterward, the reclaimed frames are reused for new pages, as described earlier (3c∼3e). Moreover, given that GC migrates flash pages, we revise GC to fit FusionFTL. If the migrated page is already cached, the CSD updates its OFN with the new flash address, while the uncached page follows the default GC processing, updating its FPN with the new flash address.

The FusionFTL is a lightweight index structure that offers several benefits over traditional approaches. First, it offers efficient lookups. The firmware requires just one memory access to determine cache hit or miss, which is more efficient than existing hash tables [89]. Second, the memory consumption imposed by FusionFTL is negligible, as it removes the redundant memory consumption caused by the FTL mapping table and cache index. Its memory overhead, comprising OFN and replacement policy data structures (e.g., reference count), is less than 0.30% of the cache size (e.g., 12MB for a 4GB cache). Lastly, unlike existing software stacks requiring separate eviction and fetch requests, FusionFTL can evict and fetch multiple pages in a single I/O command.

5.2 Concurrent I/O Processing inside CSD

After offloading the page cache manager, the entire I/O path within the CSD involves request delivery, cache management, and flash access. However, relying solely on lightweight cache management (i.e., FusionFTL) is insufficient to enhance the overall I/O performance. This limitation arises from the contention among different CSD cores over critical resources in the I/O path (e.g., FusionFTL, free page frames, and flash), resulting in performance bottlenecks. To fully unleash the computing capability of each CSD core, we redesign the I/O path for highly concurrent request processing (cf. Figure 8).

To fully utilize each CSD core, the host interface controller needs to balance the load among the cores effectively. To this end, ScalaCache splits the logical address space into many sub-address spaces by stripe (e.g., 128KB stripe) and assigns them to cores in a round-robin manner. Thus, the host interface controller first splits an I/O request into multiple sub-requests based on the sub-address space division and address range of the request. These sub-requests are further delivered to cores for processing. This fine-grained sub-request division facilitates a balanced load distribution, obviating the need for an additional load-balancing algorithm. To ensure efficient sub-request delivery without lock, we allocate a lock-free request buffer between the host interface controller and each core, allowing each core to independently retrieve sub-requests. Note that existing SSDs [27, 68] already adopt a similar lock-free request buffer [55, 56]. However, this design only allocates one buffer in one-core SSDs, restricting its applicability to multi-core CSDs. We extend the lock-free design as described above while ensuring negligible memory overhead. For example, a 4-core CSD requires only a negligible 16KB memory for four request buffers.

Since each core exclusively handles sub-requests from its own logical sub-address space, the same FPN in FusionFTL is not concurrently updated by multiple cores, obviating the need for locks to serialize updates. However, the free page frame allocation potentially encounters locking issues when multiple cores attempt to acquire free frames simultaneously. To address this, we allocate all page frames evenly to each core. Consequently, each core can manage its free page frames independently based on its respective replacement policy data structure, thereby removing locking concerns during both free frame allocation and cache replacement. Interference may also arise when cores access the same flash die simultaneously, which causes flash accesses to block each other, thereby prolonging the overall access latency. To avoid such interference, the aforementioned partition is extended to the flash, which is divided based on its inherent isolation (i.e., flash dies). Hence, each core can access the flash with hardware isolation and maintain low access latency.

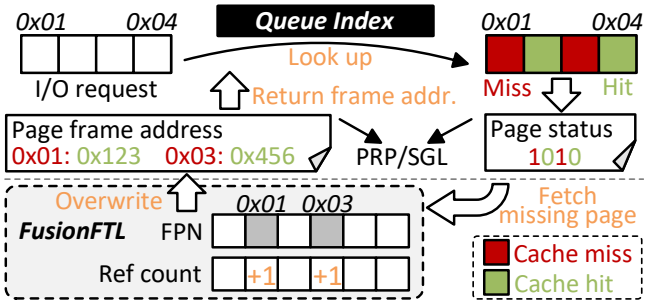


Figure 9: Coordination between Queue Index and FusionFTL.

5.3 Cache Access on the Host-side

Merely offloading the cache manager cannot sufficiently ensure high I/O performance, as in such a hardware-only design, numerous requests can overload underlying CSDs. Even with cache hits in FusionFTL, requests might suffer from a long queuing delay within CSDs (cf. Section 6 for detailed evaluation results). Our solution is to add *Queue Index* to the SPDK driver to buffer mappings from LPNs to page frame addresses of cached pages in the client threads. Specifically, we adopt a lightweight hash table [89] to construct an index structure on top of each NVMe queue. This hash table occupies memory that is less than 3.25% of the cache space size, which is similar to other index structures [15, 33, 70]. As described in Section 5.2, for pin commands, the offloaded cache manager increments the reference count of requested pages and then returns the page frame addresses to the host. These addresses are buffered in the Queue Index for fast subsequent access without repetitive lookups in FusionFTL, which also reduces the load inside the CSD.

Figure 9 depicts the workflow after incorporating Queue Index. Specifically, when processing an I/O request, the client thread checks the Queue Index for target LPNs. Once a cache hits, it readily accesses the data by referring to the buffered page frame addresses. Note that when multiple client threads access the same page frame, consistency is ensured by the CPU cache, which is aligned with existing caches [15, 43]. Otherwise, it issues a pin command to the CSD, which returns the page frame address. In addition, to maintain fair cache allocation, each client thread owns an equal portion of cache capacity. When the number of cached pages exceeds its capacity limit, the client evicts cold pages based on the clock replacement policy. These page frame addresses are packaged into an unpin command, which is subsequently issued to the CSD to release these pages. We will describe the specific pin and unpin commands shortly. Our pre-partitioned design matches the cache designs in current applications such as databases and graph processing [23, 45, 57, 63], where threads are assigned a predetermined cache size based on a load-balance algorithm. Therefore, ScalaCache does not require a redundant algorithm for cache allocation. Moreover, to meet varying cache size demands, ScalaCache allows applications to customize the cache size for different client threads.

5.4 Coordination between Host and CSDs

Trimmed communication. Section 3 highlights that current page cache manager designs heavily burden CPU resources due to intense communication. ScalaCache trims the communication through cache offloading, which removes the communication between cache manager threads and SSDs. Further, to allow client threads to directly access the offloaded cache, we incorporate pin and unpin commands into NVMe commands. However, these commands, which transfer massive amounts of information, are not well supported by the NVMe protocol. Specifically, when client threads send LPNs of the missing pages through a pin command, these non-contiguous LPNs (cf. Figure 9) create substantial CSD traffic, as NVMe commands necessitate contiguous addresses. After completing a pin command, the CSD returns multiple page frame addresses, but the NVMe commands lack sufficient reserved bits for this address transfer [3]. Similarly, during eviction, the unpin command also faces increased communication, which sends multiple non-contiguous addresses of evicted pages.

Note that when issuing the pin and unpin commands, the client thread no longer needs the address information structure (i.e., the physical region page (PRP) list or the scatter-gather list (SGL) [3]) to convey the data buffer address of requests, as the CSD transfers data directly into page frames rather than the data buffer. In addition, in SPDK, to avoid repeated memory allocations, each NVMe command is pre-allocated with these structures. We utilize these structures to transmit the aforementioned information, as shown in Figure 9. Specifically, for pin requests, the client thread stores the page hit or miss status into a bitmap (e.g., 0 and 1 indicate a cache hit and miss, respectively) within the structure. Thus, based on the bitmap, the CSD fetches the missing pages and then returns their page frame addresses by overwriting these structures. For evictions via unpin commands, the client also populates the structure with the LPNs of evicted pages, thus circumventing the limitations of the NVMe command. To further reduce communication overhead, when transferring a few pages (e.g., only one page), the client writes page frame addresses directly into the reserved bits in the NVMe command. Through these optimizations, the client and CSD communicate only once to pin and unpin multiple discontinuous pages, thus greatly reducing communication overhead.

GC-aware replacement policy. For better performance, the client thread aggressively uses its full cache capacity by pinning all cached pages in memory. By doing so, fetching new pages necessitates releasing pages first via the unpin command, and the cache manager in the CSD subsequently writes back these pages, placing page writeback in the critical path of the fetching page. However, GC can significantly delay the page writeback, as the SSD cannot serve new I/O requests during GC, thus extending the fetching latency. Therefore, to minimize the GC interference on page eviction, we propose a GC-aware replacement policy. Specifically, the CSD shares its internal GC state with the client thread by appending the state to a reserved bit in the NVMe CQ response (i.e., 0 for non-GC state and 1 for GC state). Thus, the client thread can customize its replacement policy based on GC activity. When GC occurs, the replacement policy prioritizes the eviction of clean pages (i.e., pages that do not require writeback), preventing page evictions from being blocked. That is, the policy first targets clean pages for eviction using the clock algorithm and only evicts dirty pages if clean ones are unavailable.

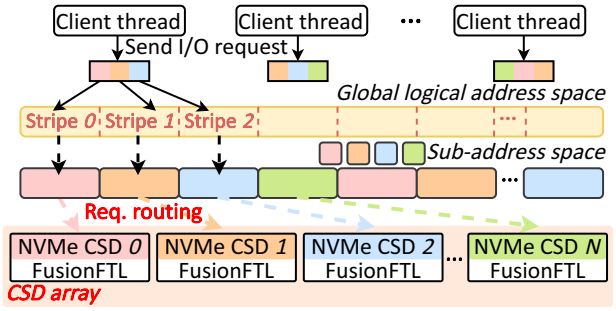


Figure 10: Page cache manager built on the CSD array.

5.5 Concurrent Cache Built on a CSD Array

The host-centric cache manager over multiple SSDs heavily taxes the CPU resources, depriving client threads of available computing resources. This constraint not only restricts the available CPU threads for applications but also impedes applications from exploiting the full performance of multiple SSDs. This issue is exacerbated as the SSD count increases, highlighting the poor scalability. In contrast, ScalaCache offloads the cache manager to CSDs, freeing up the host CPU resources for applications. In addition, to fully capitalize on the aggregated computing capability of CSDs, we design a parallel processing model that organizes multiple CSDs into a CSD array (cf. Figure 10). Specifically, we finely partition the global address space of the CSD array, whose size is equal to the aggregated capacity of all CSDs, into multiple stripes (e.g., 64KB). These stripes are then allocated in a round-robin manner to different sub-address spaces, each designated for a specific CSD. When I/O requests are received, the sub-address space to which these requests belong can be found based on their address range, and thus they are routed to the appropriate CSDs and processed concurrently. The fine-grained division enables multiple CSDs to handle requests independently and simultaneously, enhancing scalability across multiple CSDs.

5.6 Implementation

We build ScalaCache atop SPDK v22.01.2 [88] and NVMe Base Specification 2.0c [3]. To develop our CSD hardware, we adopt FEMU, a popular QEMU-based SSD emulator in academia [18, 36–38]. The host-side and CSD-side parts take 3.2K and 3.1K LOC, respectively. Specifically, we customize

Host system		FEMU		Software	
CPU	AMD EPYC 9654	**VM**	24 Core / 128 GB DRAM	**Linux kernel**	6.8
	96 Core / 2.4 GHz	**Flash**	Rd./Prog.: 18/35 us		
Mem.	768 GB DRAM		8 Channel / 4 Die / 1024 Block / 512 Page / 4 KB	**SPDK**	22.01.2
				Cache size	18.75%

Table 1: System configurations.

Trace	Req cnt. (Mops)	Avg req size (KB)	Data size (GB)	Hit ratio	Randomness	Hotness [19] (reuse dis. (GB))
webmail	7.80	4	29.74	0.96	0.22	0.21
online	5.70	4	21.80	0.94	0.26	0.62
webusers	5.70	4.22	22.90	0.71	0.30	0.40
prn_0	5.59	11.09	59.09	0.89	0.77	0.81
usr_0	2.24	22.66	48.37	0.96	0.89	1.05
src2	3.37	34.19	109.97	0.90	0.95	0.47
T1205	0.33	160.10	50.47	0.61	0.89	1.45
T2982	1.06	65.55	66.02	0.67	0.97	2.59
proj_1	23.64	34.42	775.93	0.78	0.87	1.02
mds	2.85	36.56	99.33	0.76	0.91	0.50

Table 2: The characteristics of examined workloads.

FEMU to emulate multi-core CSDs and facilitate the parallel I/O path described in Section 5.2. Additionally, to reconcile the differing memory addresses used by client threads (virtual memory addresses to access page frames) and CSDs (DMA addresses to transfer data), we extend the HMB initiation process. During HMB setup, the host provides both the DMA and virtual memory addresses of the HMB region to the CSD. Thus, the CSD returns the virtual memory address to the client threads during processing pin commands while employing DMA addresses for data transfer.

Software and firmware implementation. The Queue Index, pin/unpin interface, and partitioning strategy (i.e., stripe) are implemented in user-space software. These user-space designs not only improve cache management by eliminating context switches and lock mechanism but also allow the SPDK integration to enhance the efficiency of the I/O engine. Our firmware designs, including cache manager offloading, FusionFTL, pin/unpin command batching, and GC reporting further customize lightweight, scalable, and stable cache management. Note that these techniques can be integrated into the CSD firmware without any hardware modification.

6 Evaluation

6.1 Experimental Setup

Methodology. We allocate the FEMU virtual machine with 24 cores and 128GB DRAM. The NVMe CSDs in FEMU are set for 18us read latency and 35us write latency, which matches high-performance SSDs [61, 100]. We match the CPU frequency of the emulated CSD cores to prior ISP works [21,62,93] for accurate computing capability emulation. In addition, cache memory is set to 18.75% of SSD capacity, consistent with prior studies [8,61]. We use SPDK perf [4] and Linux perf [1] to measure the performance of cache manager designs and capture the CPU usage of their key functions. The detailed configurations are listed in Table 1.

Platforms. We compare ScalaCache against its hardware-only variant and the existing cache managers. (1) `ScalaCache`: A software-hardware coordinated user-space cache manager that includes all our proposed designs. (2) `Hardware`: A vanilla version of ScalaCache, which simply offloads the cache management to the CSD without the host-side design (i.e., designs in Section 5.3). (3) `TriCache` [15]: A state-of-the-art user-space cache manager atop the SPDK I/O engine. Due to the lack of the block interface required by the storage stack testing tools, we reproduce TriCache based on its open-source repository [77]. (4) `Kernel`: The cache manager implemented in the Linux kernel space [43]. Unless otherwise specified, we set up 4 CSD cores in both `Hardware` and `ScalaCache`.

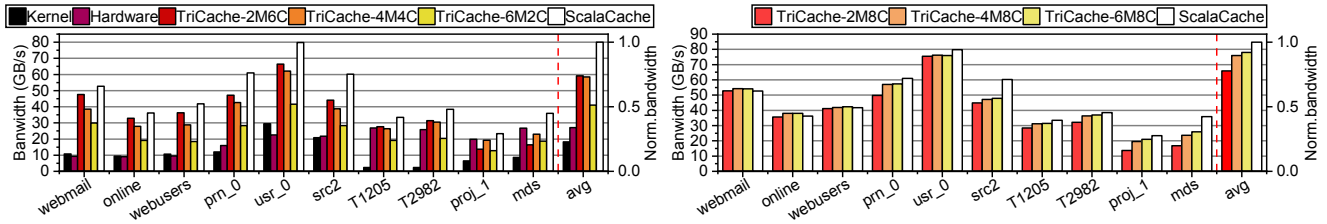
Workloads. We evaluate ScalaCache with multiple block I/O workloads, including MSR, FIU, and Tencent trace [53,67,98]. These workloads cover read-intensive, write-intensive, and mixed scenarios with requests ranging from 4 to tens of KB. We profile their cache behavior by running them with 8 client threads on `Kernel`. Table 2 lists their key characteristics.

6.2 Overall Performance

Bandwidth. Figure 11a illustrates the bandwidth comparison when using 8 host CPU cores and a CSD. `TriCache` varies cache manager thread numbers, running remaining cores as client threads (e.g., `TriCache-2M6C` with 2 manager threads and 6 client threads), while other cache manager designs use all cores for client threads. The I/O depth of each client thread is set to 32. Due to its lockless and user-space design, `TriCache-2M6C` outperforms `Kernel` by 3.84×. Furthermore, in `proj_1` and `mds` workloads, `TriCache-4M4C` outperforms `TriCache-2M6C`, indicating that more cache manager threads are required to enhance performance. In other words, its cache manager thread is heavyweight and becomes the bottleneck. `ScalaCache` outperforms all other cache manager designs in all workloads. It outperforms `Kernel` by 5.12× while outperforms `Hardware` by 1.95× as Queue Index accelerates cache lookups and avoids overloading the CSD. For `TriCache-2M6C` and `TriCache-6M2C`, `ScalaCache` outperforms them by 35.30% and 94.78%, respectively, as it frees up taxed CPU for more client threads by manager offloading. It also benefits from the lightweight FusionFTL and current processing, fully exploiting the CSD computing capability.

We also relax the number of cores in `TriCache`, that is, it uses fixed (8) cores for client threads while allocating extra cores as cache manager threads. Despite this, `ScalaCache` still outperforms it (cf. Figure 11b). `ScalaCache` outperforms `TriCache-2M8C` and `TriCache-4M8C` by 28.68% and 11.53%, respectively, as more cache manager threads in `TriCache` increase the communication cost, limiting the performance improvement. In contrast, `ScalaCache` removes the communication cost. We analyze this in Section 6.3.

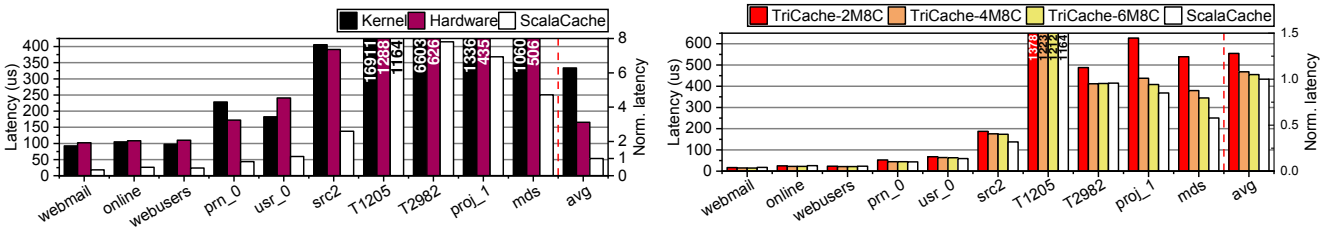
I/O latency. Figure 12a shows the average latency comparison with fixed (8) CPU cores. `ScalaCache` achieves a 78.13% latency reduction compared to `Kernel` while reducing la-



(a) Bandwidth comparison with fixed (8) host CPU cores.

(b) Bandwidth comparison with fixed (8) client threads.

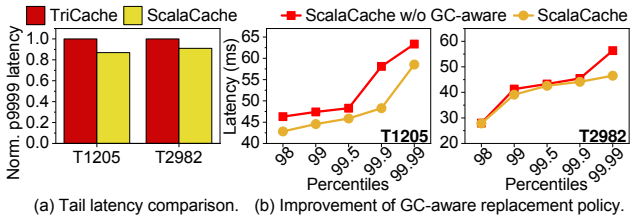
Figure 11: Bandwidth comparison.



(a) Latency comparison with fixed (8) host CPU cores.

(b) Latency comparison with fixed (8) client threads.

Figure 12: Latency comparison.



(a) Tail latency comparison. (b) Improvement of GC-aware replacement policy.

Figure 13: Tail latency comparison.

tency by 56.07% over `Hardware` since it buffers page frame addresses in client threads to shorten the cache hit path. Latency results of `TriCache` are omitted since it has fewer client threads issuing I/O requests than `ScalaCache`, leading to an unequal I/O load. A fair comparison with an equal number of clients is shown in Figure 12b. `ScalaCache` lowers latency by up to 53.50%, 33.97%, and 27.33% compared to `TriCache-2M8C`, `TriCache-4M8C`, and `TriCache-6M8C`, respectively. This efficiency stems from lightweight FusionFTL and efficient communication, which boost its processing capability and consequently diminish queuing latency.

GC impact. We compare the tail latency between `ScalaCache` and `TriCache` in two representative write-intensive workloads, `T1205` and `T2982`. Due to the GC-aware replacement policy, `ScalaCache` reduces the number of GC-affected requests by 8.19% on average, which further leads to 11% reduction in the 99.99^{th} latency (cf. Figure 13a). This reduction is unattainable with host-centric cache manager designs like `TriCache`. To further explore the effectiveness of this policy, we evaluate the tail latency of `ScalaCache` with and without GC awareness. Figure 13b reveals that the GC-aware replacement policy can reduce 99.9^{th} latency by 17.44% in `T1205` and 99.99^{th} latency by 16.76% in `T2982`, respectively. This reduction shows that this software-hardware coordinated fashion can alleviate GC impact by preventing GC from stalling page writebacks.

6.3 Performance Analysis

Figures 14 and 15 illustrate the I/O latency and CPU time breakdown for two representative workloads, `webmail` and `mds`. Both manager designs use 8 client threads, with varying manager threads for `TriCache`. `TriCache` (`T`) is decomposed based on the previously described methodology (cf. Section 3), while `ScalaCache` (`SC`) is categorized into the queuing latency of communication between client threads and FusionFTL (`Client-FusionFTL Queuing`) and inevitable latency (`Other`) that is unrelated to the cache (e.g., accessing flash).

Latency breakdown. `ScalaCache` achieves lower latency than `Kernel` due to the efficient cache manager and I/O engine. Compared to `TriCache`, `ScalaCache` removes `Client-Manager Queuing` due to the trimmed I/O path. In `mds`, compared to the counterpart in `TriCache` (i.e., `Manager-SSD Queuing`), `ScalaCache` reduces `Client-FusionFTL Queuing` by 47.09%. This is because numerous NVMe commands in `TriCache` due to request fragmentation overload its cache manager threads, which prolongs the queuing latency, while `ScalaCache` optimizes communication through bundling multiple missing pages into a single NVMe command. It also benefits from the lightweight cache index structure, which accelerates the address translation. We further analyze the performance issues caused by cache offloading. Although requests suffer from the additional I/O engine, the latency penalty is slight, such as a 13.97% increase in `webmail` against `TriCache`.

CPU time breakdown. Compared to `Kernel`, `ScalaCache` decreases CPU time by 77.75%. Note that client threads in `TriCache` consume much CPU waiting for manager responses (i.e., `Msg Poll`), indicating cache manager threads become a bottleneck. Moreover, its communication cost escalates with cache manager thread counts. For instance, `I/O engine` in `T-6M` requires 1.03× more CPU time than `T-2M`,

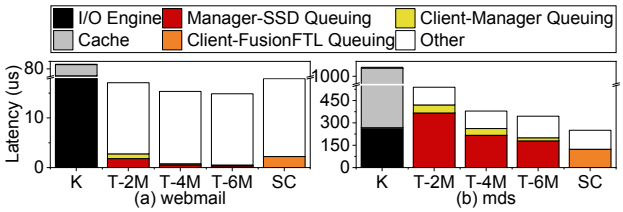


Figure 14: I/O latency breakdown.

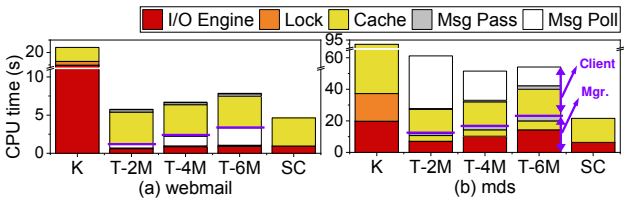


Figure 15: CPU time breakdown.

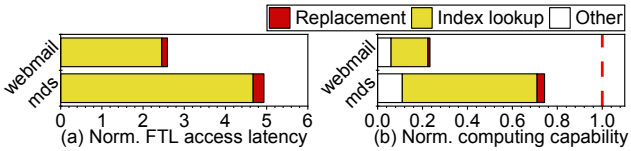


Figure 16: FTL access latency and computing capability.

as more cache manager threads exacerbate request fragmentation. In contrast, `ScalaCache` reduces CPU time by 45.67% on average compared to `TriCache`, which can be attributed to the elimination of the CPU burden imposed by cache manager threads as well as the trimmed I/O path. In addition, `I/O engine` consumes 34.58% less compared to `TriCache`, thanks to the batch NVMe command process.

FusionFTL analysis. We measure the FusionFTL access latency and the required CSD computing capability, which are normalized to the counterpart in `Kernel` and existing CSDs [50, 93], respectively. Figure 16a reveals that FusionFTL introduces 2.76× latency overhead on average, where the overhead due to the replacement policy accounts for 5.28%. Figure 16b shows that even under the intense load in `mds`, ScalaCache only consumes 74.06% of the CSD computing capability. Under the reduced load (e.g., `webmail`) due to software-hardware coordination, the computing burden is reduced to 23.26%. Consequently, the lightweight cache manager in the CSD handles pin/unpin calls for `webmail` and `mds` at 1.49 Mop/s and 0.24 Mop/s, respectively, outperforming the naive scheme (i.e., integrates the hash table directly into the CSD) by 1.14× on average. These results indicate that CSDs can effectively undertake cache offloading. Note that CSD exhibits a lower processing frequency than the request frequency in `webmail` as Queue Index filters a large fraction of requests, leaving the remaining requests to the CSD.

6.4 Scalability

Scalability with host CPU cores. Figure 17 depicts the performance scalability against varying CPU cores. Due to page limits, we show six representative results. `Hardware` is more scalable than `Kernel` in some workloads (e.g., `proj_1`) by eliminating locks, but faces scalability issues when overloaded (e.g., `webmail`). `TriCache` maintains good scalability in scenarios with small I/O requests (e.g., `webmail`), however, it struggles with large I/O requests (e.g., `src2`), where cache manager threads become the bottleneck due to the request fragmentation. Conversely, `ScalaCache` consistently shows improved scalability in all workloads. For example, the peak performance in `src2` surpasses `TriCache-2M` and `TriCache-4M` by 35.17% and 31.59%, respectively. The good scalability is due to lightweight and lockless designs, including lightweight cache management, the lockless resource allocation framework, and the concurrent I/O processing.

Scalability with multiple SSDs. Figures 18a and 18b depict the bandwidth when using 14 and 18 host CPU cores by varying SSD counts, respectively. In `TriCache`, we allocate two cache manager threads for each SSD, while in `Hardware` and `ScalaCache` we set to two cores for each CSD. `TriCache` performance decreases with more SSDs, revealing its inability to scale across SSDs due to the heavy CPU tax. As the SSD count scales up, fewer cores are available for client threads, leading to SSDs being underutilized. In contrast, `ScalaCache` and `Hardware` maintain scalability. With 18 cores and 8 SSDs, they outperform `TriCache` by 1.34× and 1.70×, respectively. By offloading cache management to CSDs, they allow the host CPU to be entirely dedicated to client threads, fully utilizing all CSDs. Moreover, the CSD array aggregates the computing power of multiple CSDs to process requests concurrently.

7 Related Work and Discussion

Cache management. Previous efforts [15, 58, 61, 85] aim to mitigate cache management overhead. FrozenHot [61] removes unnecessary management for hot data, while [58, 85] focus on locking mechanism optimization. TriCache [15] advances this by designing a lock-free cache manager. Despite these advancements, cache management continues to tax the host CPU. Unlike these host-centric designs, ScalaCache offloads the cache to the CSD, thereby eliminating such CPU overhead. Moreover, ScalaCache accelerates address translation and reduces GC disruption via hardware-software coordination, which is unattainable by these host-centric caches.

User-space storage software stack. To alleviate the overhead of the kernel storage stack, several works [14, 15, 88] shift part or the entire storage stack into user space. [20, 29, 46, 91] design user-space file systems to avoid trapping into the kernel. Differing from their contributions, ScalaCache focuses on the user-space cache manager. SPDK [88] relocates the NVMe driver to user space and employs lock-free resource allocation to enhance performance. Extending this lock-free paradigm, ScalaCache proposes a lock-free resource allocation framework in CSDs. Unlike SPDK, which concentrates on host resources, ScalaCache addresses the overlooked CSD resource allocation to mitigate interference within CSDs.

In-storage processing. Capitalizing on the rich resources

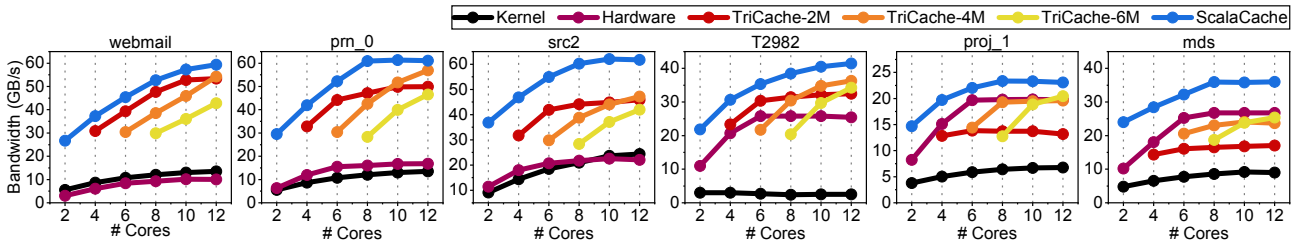
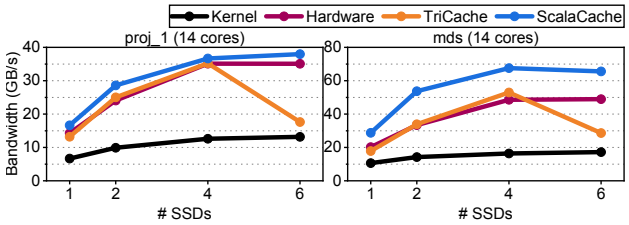
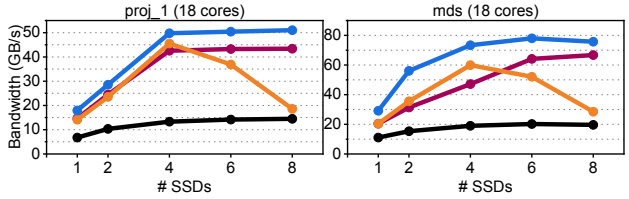


Figure 17: Scalability with host CPU cores.



(a) Performance with 14 host CPU cores.



(b) Performance with 18 host CPU cores.

Figure 18: Scalability with varying CPU cores and SSDs.

within CSDs, many studies [12, 28, 39, 60] propose to offload data-intensive tasks, such as database scans and machine learning. [21, 62, 93] delegate file systems to CSDs for high-performance direct I/O access. In contrast to their contributions, ScalaCache focuses specifically on offloading cache management. ScalaCache further exploits the aggregation of multiple CSDs to enable scalability, while they ignore the optimization opportunity to apply ISP on multiple CSDs.

Message batching. Prior studies [15, 17, 88] adopt message batching to amortize the overhead between different host threads. However, they overlook the host-device communication. Although [44] reduces the I/O engine communication via batching requests, it cannot eliminate excess NVMe commands, potentially tripling overhead. In contrast, ScalaCache reduces this by customizing NVMe commands and CSD internals, which is unachievable through sole software design.

Load balance. Most well-optimized applications have already integrated effective algorithms to balance loads between threads, which have been widely studied in prior works [23, 45, 57, 63]. A lightweight resource partitioning strategy and request slicing as what ScalaCache provides are sufficient for these applications to fully exploit the high concurrency and throughput of ScalaCache. In rare cases of ill-tuned applications, ScalaCache can integrate existing sophisticated balancing schemes [8, 41, 47] to meet load-balancing demands.

CXL interconnect. CXL [13] offers efficient fine-granule access between host memory and CXL devices, which may aid small-size metadata transfers in ScalaCache (e.g., GC reporting). However, adopting CXL into CSDs necessitates customized hardware units to support cache coherence. ScalaCache does not require such a strong coherence guarantee. For instance, cache replacements do not demand cache coherence support, as the CPU is guaranteed not to access pages being loaded or evicted by the cache management within CSD.

Pin/unpin interface. Multiple applications [5, 59, 63] construct their built-in caches by explicitly utilizing the pin/unpin interface to access pages. Alternatively, other applications can customize a compiler module to implicitly call this interface, which is facilitated by inserting pin/unpin calls into I/O functions (e.g., read) via LLVM instrumentation [48]. The explicit invocation supported by applications allows direct integration of ScalaCache into applications, while the implicit invocation necessitates only recompilation to insert pin/unpin calls.

8 Conclusion

We propose software-hardware coordination for the user-space cache manager, called ScalaCache. Specifically, ScalaCache offloads the cache manager to CSDs and further constructs a lightweight cache index within CSDs, reducing the CPU tax. Moreover, it reduces communication tax by trimming the I/O path while mitigating interference tax via a GC-aware replacement policy. Our evaluation reveals that ScalaCache improves bandwidth by 1.70× over prior work.

Acknowledgement

We thank our shepherd, Animesh Trivedi, and the anonymous reviewers for their constructive feedback. This work is mainly supported by the National Key Research and Development Program of China under Grant No. 2023YFB4502702, the National Natural Science Foundation of China under Grant No. 62332021, the Fundamental Research Funds for the Central Universities, Peking University, and the State Key Lab of Processors, Institute of Computing Technology, CAS under Grant No. CLQ202309. Dr. Li is supported in part by the National Natural Science Foundation of China under Grant No. 62202396. Dr. Wang is supported in part by the Innovation Funding of ICT, CAS under Grant E261110. The corresponding author is Jie Zhang.

References

[1] Linux perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2022.

[2] Marvel bravera sc5 ssd controllers. https://www.marvell.com/products/ssd-controllers/mv-ss1331-1333.html, 2022.

[3] Nvm express base specification 2.0c. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf, 2022.

[4] Spdk perf. https://github.com/spdk/spdk/blob/master/examples/nvme/perf/perf.c, 2022.

[5] Mania Abdi, Amin Mosayyebzadeh, Mohammad Hossein Hajkazemi, Emine Ugur Kaynar, Ata Turk, Larry Rudolph, Orran Krieger, and Peter Desnoyers. A community cache with complete information. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 323–340, 2021.

[6] ANANDTECH. Marvell announces new client nvme ssd controllers. https://www.anandtech.com/show/12895/marvell-announces-new-client-nvme-ssd-controllers.

[7] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[8] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[10] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[11] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[12] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 29–41, 2020.

[13] CXL. Compute express link. https://computeexpresslink.org/.

[14] 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*, pages 478–493, 2019.

[15] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 395–411, 2022.

[16] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[17] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proceedings of the VLDB Endowment*, 16(9):2090–2102, 2023.

[18] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 147–162, 2021.

[19] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[20] 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*, pages 494–508, 2019.

[21] 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 *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, 2018.

[22] 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 *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[23] Juno Kim and Steven Swanson. Blaze: fast graph processing on fast ssds. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[24] Kyusik Kim, Eunji Lee, and Taeseok Kim. Hmb-ssd: Framework for efficient exploiting of the host memory buffer in the nvme ssd. *IEEE Access*, 7:150403–150411, 2019.

[25] Yoona Kim, Inhyuk Choi, Juhyung Park, Jaeheon Lee, Sungjin Lee, and Jihong Kim. Integrated Host-SSD mapping table management for improving user experience of smartphones. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 441–456, 2023.

[26] Gunjae Koo, Kiran Kumar Matam, Te I, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231, 2017.

[27] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[28] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/Software Co-Programmable framework for computational SSDs to accelerate deep learning service on Large-Scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 147–164, 2022.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[30] 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.

[31] Jongwon Lee, Jaejun Ko, and Young-June Choi. Task offloading technique using dmips in wearable devices. In *2017 International Conference on Information Networking (ICOIN)*, pages 414–416. IEEE, 2017.

[32] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[34] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGARCH Computer Architecture News*, 45(1):449–466, 2017.

[35] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and optimization of network I/O tax in confidential virtual machines. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1–15, 2023.

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

[37] 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.

[38] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. lODA: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 263–279, 2021.

[39] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for In-Storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, 2019.

[40] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting cache files for boosted mobile performance and lifetime. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 445–459, 2022.

[41] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1156–1168, 2021.

[42] Linux. NVMe driver. https://github.com/torvalds/linux/tree/master/drivers/nvme.

[43] Linux. Page cache layer. https://github.com/torvalds/linux/tree/master/fs.

[44] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. Rail: Predictable, low tail latency for nvme flash. *ACM Transactions on Storage (TOS)*, 18(1):1–21, 2022.

[45] Hang Liu and H Howie Huang. Graphene:Fine-Grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, 2017.

[46] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 819–835, 2021.

[47] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[48] LLVM. The llvm compiler infrastructure. https://llvm.org/.

[49] Bo Mao, Suzhen Wu, and Lide Duan. Improving the ssd performance by exploiting request characteristics and internal parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):472–484, 2017.

[50] Marvell. Marvell® bravera™ sc5 ssd controllers. https://www.marvell.com/content/dam/marvell/en/public-collateral/storage/marvell-ssd-mv-ss1331-1333-product-brief.pdf.

[51] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th international symposium on computer architecture*, pages 116–128, 2019.

[52] Cayla McGinnis. Pci-sig® fast tracks evolution to 32gt/s with pci express 5.0 architecture. *News Release, June*, 7, 2017.

[53] Storage networking industry association. the snia's i/o traces, tools, and analysis (iotta) repository. http://iotta.snia.org/.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[55] OpenSSD. Cosmos-openssd. https://github.com/Cosmos-OpenSSD/Cosmos-OpenSSD.

[56] OpenSSD. Cosmos-plus-openssd. https://github.com/Cosmos-OpenSSD/Cosmos-plus-OpenSSD.

[57] Oracle. Setting the database cache. https://docs.oracle.com/cd/E22289_01/html/821-1274/setting-the-database-cache.html.

[58] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827, 2020.

[59] Neoklis Polyzotis and Yannis E Ioannidis. Speculative query processing. In *CIDR*. Citeseer, 2003.

[60] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the b+-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 69–82, 2022.

[61] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management

for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 557–573, 2023.

[62] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A cross-layered Direct-Access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154, 2020.

[63] Rocksdb. Rocksdb. https://github.com/facebook/rocksdb, 2023.

[64] Zhibing Sha, Jun Li, Lihao Song, Jiewen Tang, Min Huang, Zhigang Cai, Lianju Qian, Jianwei Liao, and Zhiming Liu. Low i/o intensity-aware partial gc scheduling to reduce long-tail latency in ssds. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–25, 2021.

[65] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 338–349, 2009.

[66] SNIA. 10 million i/ops from a single thread. https://www.snia.org/educational-library/10-million-iops-single-thread-2019.

[67] SNIA. Fiu traces. http://iotta.snia.org/traces/block-io/390.

[68] 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*, pages 1–30, 2014.

[69] Avinash Srinivasan, Jie Wu, Panneer Santhalingam, and Jeffrey Zamanski. Deaddrop-in-a-flash: Information hiding at ssd nand flash memory physical layer. *SECURWARE*, 79:2014, 2014.

[70] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. SmartCuckoo: A fast and Cost-Efficient hashing index scheme for cloud storage systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 553–565, 2017.

[71] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. *ACM Transactions on Storage (TOS)*, 18(1):1–32, 2022.

[72] Andrew S Tanenbaum. *Modern operating systems*. China-Pub-Com, 2002.

[73] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[74] TechTarget. Computational storage takes spotlight in new ngd systems ssd. https://www.techtarget.com/searchstorage/news/252459062/Computational-storage-takes-spotlight-in-new-NGD-Systems-SSD.

[75] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. Compstor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1260–1267. IEEE, 2018.

[76] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Catalina: in-storage processing acceleration for scalable big data analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 430–437. IEEE, 2019.

[77] TriCache. Tricache. https://github.com/thu-pacman/TriCache, 2022.

[78] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. *Acm Sigplan Notices*, 50(7):1–15, 2015.

[79] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 352–367, 2022.

[80] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729, 2021.

[81] Guanying Wu and Xubin He. Delta-ftl: Improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 253–266, 2012.

[82] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the gc-induced performance variability in ssd-based raids with request redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):822–833, 2018.

[83] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, et al. Spool: Reliable virtualized NVMe storage pool in public cloud infrastructure. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 97–110, 2020.

[84] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[85] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021.

[86] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[87] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-IO: A unified IO stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.

[88] 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.

[89] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). *ACM SIGMETRICS Performance Evaluation Review*, 44(1):337–350, 2016.

[90] Richard York. Benchmarking in context: Dhrystone. *ARM, March*, 2002.

[91] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. EvFS: User-level,Event-Driven file system for Non-Volatile memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[92] Cristian Zambelli, Riccardo Bertaggia, Lorenzo Zuolo, Rino Micheloni, and Piero Olivo. Enabling computational storage through FPGA neural network accelerator for enterprise ssd. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(10):1738–1742, 2019.

[93] Jian Zhang, Yujie Ren, and Sudarsun Kannan. FusionFS: Fusing I/O operations using CISCOps in firmware file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 297–312, 2022.

[94] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 13–24. IEEE, 2015.

[95] Jie Zhang and Myoungsoo Jung. Zng: Architecting gpu multi-processors with new flash for scalable data analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1064–1075. IEEE, 2020.

[96] Jie Zhang, Miryeong Kwon, Sanghyun Han, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. FastDrain: Removing page victimization overheads in NVMe storage stack. *IEEE Computer Architecture Letters*, 19(2):92–96, 2020.

[97] 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 Technologies (FAST 20)*, pages 121–136, 2020.

[98] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020.

[99] Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Zhe Yang, Youyou Lu, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang. Revisiting swapping in user-space with lightweight threading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[100] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP:In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.