ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs

Shushu Yi, *Peking University and Zhongguancun Laboratory;* Xiurui Pan, *Peking University;* Qiao Li, *Xiamen University;* Qiang Li, *Alibaba;* Chenxi Wang, *University of Chinese Academy of Sciences;* Bo Mao, *Xiamen University;* Myoungsoo Jung, *KAIST and Panmnesia;* Jie Zhang, *Peking University and Zhongguancun Laboratory*

https://www.usenix.org/conference/atc24/presentation/yi-shushu

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

ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs

Shushu Yi[1,2], Xiurui Pan[1], Qiao Li[3], Qiang Li[4]
Chenxi Wang[5], Bo Mao[3], Myoungsoo Jung[6,7], Jie Zhang[1,2]
National Key Laboratory for Multimedia Information Processing,
School of Computer Science, Peking University[1], Zhongguancun Laboratory, Beijing, China[2]
Xiamen University[3], Alibaba[4], University of Chinese Academy of Sciences[5], KAIST[6], Panmnesia[7]
https://www.chaselab.wiki

Abstract

All-flash array (AFA) is a popular approach to aggregate the capacity of multiple solid-state drives (SSDs) while guaranteeing fault tolerance. Unfortunately, existing AFA engines inflict substantial software overheads on the I/O path, such as the user-kernel context switches and AFA internal tasks (e.g., parity preparation), thereby failing to adopt next-generation high-performance SSDs.

Tackling this challenge, we propose ScalaAFA, a unique holistic design of AFA engine that can extend the throughput of next-generation SSD arrays in scale with low CPU costs. We incorporate ScalaAFA into user space to avoid user-kernel context switches while harnessing SSD built-in resources for handling AFA internal tasks. Specifically, in adherence to the lock-free principle of existing user-space storage framework, ScalaAFA substitutes the traditional locks with an efficient message-passing-based permission management scheme to facilitate inter-thread synchronization. Considering the CPU burden imposed by background I/O and parity computation, ScalaAFA proposes to offload these tasks to SSDs. To mitigate host-SSD communication overheads in offloading, ScalaAFA takes a novel data placement policy that enables transparent data gathering and in-situ parity computation. ScalaAFA also addresses two AFA intrinsic issues, metadata persistence and write amplification, by thoroughly exploiting SSD architectural innovations. Comprehensive evaluation results indicate that ScalaAFA can achieve 2.5× write throughput and reduce average write latency by a significant 52.7%, compared to the state-of-the-art AFA engines.

1 Introduction

The last decade has witnessed all-flash arrays (AFA) [36, 41, 48, 59, 63, 71] increasingly adopted as buffer layers in high-performance computing systems and datacenters [18,49]. These systems have proven critical in speeding up numerous I/O-intensive scenarios, including big data analysis, scientific computing, and machine learning [1, 17, 39, 57, 62, 66]. In comparison to traditional storage media like hard disks, AFAs capitalize on the advantages of solid-state drives (SSDs) such as enhanced throughput, latency, and power efficiency. AFAs bundle multiple SSDs into an array to improve storage capacity at scale, thereby providing a large and uniform storage space. Additionally, AFAs tackle flash errors by integrating data redundancy mechanisms.

Considerable efforts have been expended in academia and industry to make AFAs more practical [4, 26, 36, 65, 68]. For instance, Linux software RAID, known as *mdraid* [4], has been developed to exploit multi-core processors for concurrent parity preparation. Building on mdraid, two-phase write schemes [26, 36, 65, 68] have revolutionized the write path. These systems use *replication* (e.g., RAID 10) as a stepping stone to *striping* (e.g., RAID 5 or 6) for fast I/O processing and update data out-of-place to accelerate small writes.

However, with the continual advancement in SSD technology, current AFA implementations risk becoming a bottleneck for future storage systems that will leverage next-generation SSDs. For instance, Samsung PM1743 PCIe 5.0 SSDs can deliver up to 13 GB/s I/O bandwidth [10], a stark contrast to the majority of AFA engines [4, 36] that were originally designed for slower storage interfaces (i.e., SATA) and top out at a maximum throughput of 500 MB/s. To elucidate the performance issue in AFA, we conduct an experiment with a state-of-the-art two-phase write AFA engine, *FusionRAID* [36] (cf. § 3.1 for details). During replication, FusionRAID only achieved 4.8 GB/s write throughput, a mere 36.9% of the ideal performance. This is attributed to the storage software stack (e.g., user-kernel context switches and tedious block layers) consuming 54.4% of the CPU cycles, whilst the CPU stall time for SSD I/O only accounts for 20.8%.

SPDK [70], one of the most popular user-space storage frameworks, presents an encouraging approach to lighten the storage software stack. However, directly incorporating existing AFA solutions into SPDK poses significant challenges. First, existing AFA solutions rely on locks to facilitate concurrent multi-thread access [4, 71], whereas SPDK operates based on a lock-free principle. Furthermore, SPDK cannot mitigate the intrinsic shortcomings of existing AFA schemes (e.g., two-phase write). For instance, the "conversion" process from replication to striping involves reading replicated data from SSDs, computing parities, and storing them back in a space-efficient layout, leading to intensive background I/O and computation, which ultimately results in notable performance degradation in I/O-intensive scenarios (evidenced by an 88.5% throughput reduction as shown in Figure 3b). In addition, out-of-place updates necessitate extra mapping tables to track data locations. The persistence of this metadata is CPU-intensive. Lastly, the write amplification triggered by replication can significantly shorten the lifetime of SSDs.

In response to these challenges, we introduce *ScalaAFA*[1], an innovative user-space AFA engine designed to maximize the performance of future SSD arrays while maintaining low CPU overhead. In accordance with SPDK's lock-free principle, ScalaAFA substitutes the traditional locks with an SPDK-compatible message-passing-based permission management scheme for concurrent multi-thread accesses. It also minimizes the synchronization overhead with a lightweight storage space abstraction and batch processing method. Recognizing the CPU load imposed by conversion, we propose an inventive data placement policy that curtails background I/O and offloads parity computation to the embedded resources of the SSDs transparently. In the face of frequently updated mapping tables, ScalaAFA leverages a hardware-based crash consistency mechanism, significantly reducing software overhead. Finally, to temper the damage of write amplification on SSD lifetime, ScalaAFA suggests harnessing the SSD-internal high-endurance write buffer to accommodate redundant writes. Evaluation results demonstrate that ScalaAFA surpasses leading AFA engines, delivering 2.5× write throughput and reducing average write latency by 52.7%.

Our core **contributions** can be summarized as follows:

• *Constructing a lock-free AFA engine in user space:* An in-depth analysis of the existing AFA engines reveals that the primary hurdles for integrating AFA engines into user space stem from the complex lock mechanisms for concurrent multi-thread accesses. To overcome this impediment, we propose to manage write permission with a lightweight storage space abstraction and efficiently grant/retrieve these permissions in batches with a message-passing scheme. This solution conforms to the lock-free principle of user space designs and facilitates thread-level parallelism at a low cost. Consequently, it increases the write throughput by 58.4% while decreasing the average latency by 45.2%. To the best of our knowledge, this is the first study that successfully incorporates an AFA engine into user space.

• *Offloading conversion to SSDs:* In two-phase write AFAs, data chunks belonging to the same stripe are dispersed across different SSDs after replication. Offloading conversion to SSDs is non-trivial, as it requires the host to transfer data

[1] ScalaAFA is accessible at https://github.com/ChaseLab-PKU/ScalaAFA.

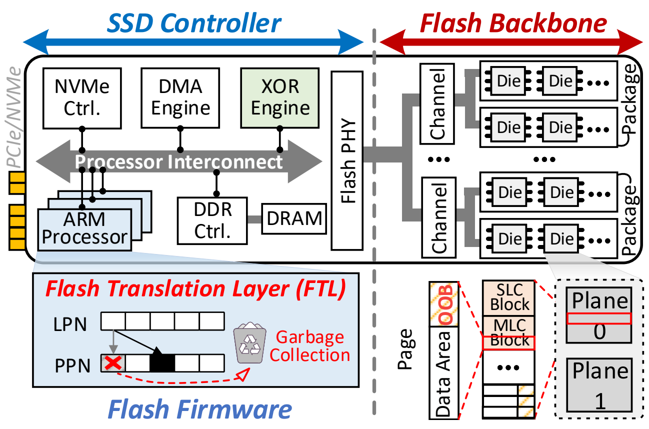


Figure 1: Details of SSD internal.

chunks to the target SSD manually, which imposes extra I/O and huge CPU burdens. We propose an innovative data placement policy that can transparently gather data chunks into the target SSDs. Specifically, when the AFA engine generates redundant writes for fault tolerance, we redirect the redundant replicas of the same stripe to the SSD where the parity will be stored. By leveraging the SSD built-in XOR engine to calculate the parity codes from the local replicas, we eliminate the need of host involvement (i.e., computation and data transfer). This design further improves the write throughput by 36.9% and reduces average latency by 36.1%.

• *Optimizing two-phase write with holistic designs:* Prior work has revealed that two-phase write AFA can outperform traditional AFA (e.g., RAID 5) especially when serving small write requests. Nevertheless, how to conceal the inherent drawbacks of two-phase write (i.e., metadata persistence and write amplification) is still unsolved. Notably, we propose to tackle these challenges by thoroughly exploiting SSD architectural innovations. We suggest a hardware-based crash consistency mechanism that employs the out-of-band areas of SSDs to persist the metadata with minor overhead. Moreover, we take a comprehensive design that accommodates transient data within the SSD-internal durable buffers and avoids flushing them to vulnerable flash cells. This design succeeds in reducing the impact of write amplification by 38.6%.

2 Background

2.1 SSD Internal

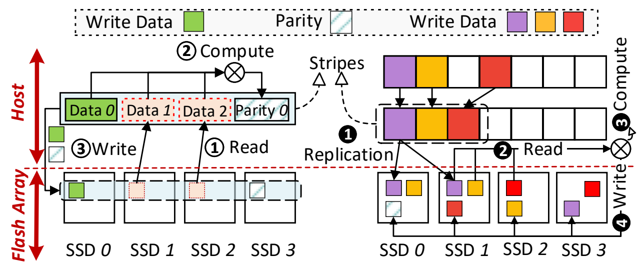
Baseline architecture. Figure 1 depicts a common architecture of modern SSDs [3, 73]. The SSD is comprised of multiple embedded processors (e.g., ARM), a DDR DRAM controller, and specialized processing elements, such as DMA and *XOR* engines [15, 16, 67]. The XOR engine bolsters the device's reliability by calculating parity codes for data stored in the SSD. These processors are linked to a flash backbone through the flash physical layer (PHY). The flash backbone consists of 4 to 16 channels, each connecting to several flash packages. Each flash package encloses multiple flash dies, each comprised of 2 to 4 flash planes. A single plane can be divided into thousands of flash blocks. The flash blocks can be categorized as *single-level-cell (SLC)* and *multiple-level-cell (MLC)* based on the number of data bits stored in a flash cell [11, 28]. The SLC blocks offer shorter I/O latency, extended endurance, and reduced capacity, while the MLC blocks provide larger capacity but exhibit longer I/O latency and reduced endurance. To optimize SSD lifetime, capacity, and performance, SSD manufacturers use SLC blocks as the write buffer to accommodate small writes and employ MLC blocks as storage backend [32,34]. A flash block contains hundreds of flash pages, each divided into data and *out-of-band (OOB)* areas. The data area stores data whilst the OOB area stores metadata, such as error correction codes (ECC) [33]. Note that the size of OOB area in NAND flash is typically greater than what metadata requires (e.g., tens of bytes reserved per flash page [16, 29]).

Flash firmware. NAND flash only supports out-of-place updates due to its physical attributes [54]. To shield users from flash intrinsic, the flash firmware internally constructs an indirection layer, known as *flash translation layer (FTL)* [20, 58], to remap incoming write requests to new flash pages and invalidate the stale flash pages (cf. Figure 1). FTL maintains the mapping information between the request logical address (i.e., logical page number, LPN) and the flash physical address (i.e., physical page number, PPN) in the SSD-internal DRAM. The flash firmware also persists this mapping information in the OOB area, enabling it to be recovered after system corruption or reboot. Flash page invalidation due to write requests can significantly diminish the available SSD capacity for users. To tackle this problem, the firmware performs garbage collection (GC) to reclaim invalid pages. Specifically, it selects a victim block and moves all valid pages in it to an erased block. Subsequently, the victim block is erased and reused.

2.2 All-Flash Array

All-flash array (AFA) is a storage organization that groups multiple SSDs as a single logical unit to aggregate their capacity and throughput while providing fault tolerance. Existing AFA designs can be classified into two types, stripe write and two-phase write, based on how data is written to SSDs.

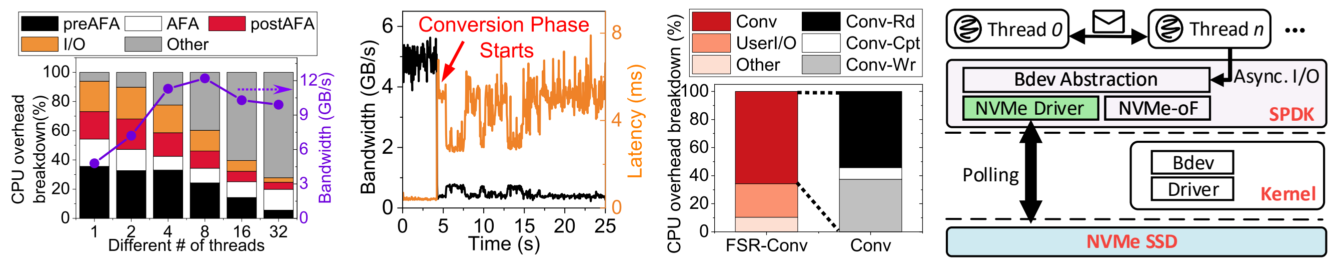
Stripe write AFA. Stripe write AFA orchestrates data as *stripes*. Each stripe consists of k data chunks and m parity chunks with a fixed size (e.g., 64 KB). The parity chunks are calculated with a specific error-correction code (ECC) algorithm (e.g., Reed-Solomon codes [64]). The $k+m$ chunks are distributed and stored in $k+m$ SSDs (we call this *striping layout*). If a few data chunks are lost due to SSD failures, the same number of parity chunks can be used to recover the lost data chunks. Therefore, a $k+m$ AFA can tolerate up to m SSD corruptions. Figure 2a shows how a stripe write AFA serves write requests. When a write request arrives, it first slices the request into multiple data chunks. If the number of data chunks is less than k, the write request is called *partial write*. Partial write, unfortunately, is unfriendly to stripe write AFA. The AFA engine must fetch the missing data chunks belonging to the same stripe from SSDs (①). Then, the host CPU calculates the parity chunks for this stripe (②). Finally, both the data and parity chunks will be written to the storage devices (③). This procedure is called *read-construct-write*. Partial writes introduce extra data reads and intensive parity updates, which significantly delay the I/O completion time.



(a) Stripe write AFA. (b) Two-phase write AFA.

Figure 2: Write paths of all-flash arrays (k=3 and m=1).

Two-phase write AFA. To tackle the challenge imposed by partial writes, prior work proposes two-phase write AFA (or log-structured write AFA) [26, 36, 65, 68], which employs replication as the prelude of striping to absorb small write requests. Two-phase write AFA [36] exhibits superiority over stripe write AFA [4] in terms of both throughput and latency (e.g., 51.2% throughput improvement and 33.9 % average latency reduction for 64 KB sequential write, cf. § 6.2), two of the most crucial metrics when employing AFAs as the buffer layers in datacenters [17, 39, 57, 62]. Figure 2b shows the procedure of two-phase writes. Specifically, two-phase write AFA writes data chunks in two phases: *replication phase* and *conversion phase*. In the replication phase, the data chunks are replicated into $m+1$ copies and distributed across $m+1$ SSDs (❶), which provides the same fault tolerance as stripe write AFA with $k+m$ SSDs. When the size of replicated data exceeds a given watermark (e.g., 5% of the AFA capacity), user I/O requests are delayed and the conversion phase starts. In particular, the host reads all data chunks of the same stripe from SSDs (❷) and computes the parity chunks (❸). Afterward, these data and parity chunks are written back to SSDs and stored in the space-efficient striping layout (❹). Finally, the space previously used to store replicated data chunks will be recycled. Note that all the updates in two-phase write AFA are out-of-place. Therefore, when updating data chunks, two-phase write AFA only needs to replicate the updated data to new spaces and invalidate the stale data, which avoids the tedious read-construct-write procedure of stripe write.



(a) Analysis of FSR-Rep. (b) Performance of FusionRAID. (c) Overhead analysis. (d) Architecture of SPDK.

Figure 3: Deep analysis of two-phase write AFA and key insights.

3 Preliminary Study

3.1 Challenges

While two-phase write has demonstrated its superiority over stripe write in terms of both throughput and latency, we observe that the designs of existing two-phase write AFA engines are still the bottleneck when adopting high-performance SSDs. To illustrate this, we reproduce FusionRAID [36], one of the state-of-the-art two-phase write AFA engines, and set up an experiment with 4+1 ($k+m$) Samsung 980 Pro SSDs [2] to analyze its write performance. We use fio [13] and perf [23] to evaluate the I/O performance and capture CPU cycles of the key functions in the storage stack, respectively.

Challenge in replication phase. Figure 3a shows the 64 KB sequential write throughput and CPU overhead breakdown with different numbers of I/O threads in the replication phase of FusionRAID (*FSR-Rep*). We categorize the overhead into five parts: `preAFA` is the time of submitting I/O requests from user space to FSR-Rep through context switches and block layers, while `AFA` represents the time consumed by FSR-Rep; `postAFA` is the time of sending I/O requests to SSDs through the NVMe driver; `I/O` is the time of serving I/O requests in SSDs; Lastly, `Other` summarizes other software overheads (e.g., CPU spins for synchronization among the I/O threads and kernel worker threads [63, 71]).

FSR-Rep achieves 4.8 GB/s and 7.2 GB/s write throughput with 1 and 2 threads, which are only 36.9% and 55.4% of the ideal case (i.e., $(k+m)/(m+1)*S$, where S is the peak throughput of a single SSD, that is, 5.2 GB/s for Samsung 980 Pro). This is because the frequent user-kernel context switches and the tedious storage stack (i.e., `preAFA` and `postAFA`) consume tremendous CPU ticks. For example, with one thread, `I/O` only accounts for 20.8%, while `preAFA` and `postAFA` consume 54.4%. One way to improve the throughput is employing more CPU resources. For example, FSR-Rep achieves 12.2 GB/s with 8 threads. More than 8 threads instead degrades the throughput because inter-thread synchronization (i.e., `Other`) dominates the CPU overhead [63, 71]. However, allocating 8 threads per 4+1 AFA is still infeasible for most existing storage servers, which equip hundreds of SSDs but limited CPU resources. For instance, PowerStore 500T [9] holds up to 97 SSDs while only equipping two 24-core CPUs. To summarize, *the software overhead imposed by the tedious storage stack has become a hindrance to achieving high performance with limited CPU resources* ***(Challenge 1)***.

Challenge in conversion phase. The performance becomes worse when the conversion phase starts (*FSR-Conv*). We continuously send 64 KB sequential write requests to FusionRAID with one I/O thread. Figure 3b shows the write throughput and average latency over time. When the conversion phase starts, write throughput drops from about 4.8 GB/s to 550 MB/s, and the latency increases from 0.4 ms to 5.9 ms. We break down the CPU overhead of FSR-Conv into three parts, which are shown in Figure 3c. `Conv` is the time consumed by the conversion in the background. `UserI/O` is the time of serving I/O requests sent by the user (i.e., replication of 64 KB sequential write). Lastly, `Other` includes other software overheads in FSR-Conv (e.g., CPU spins for synchronization among the I/O thread and kernel worker threads). `Conv` accounts for 65.7% while `UserI/O` only consumes 23.9% of the CPU resources. We further categorize `Conv` into `Conv-Rd`, `Conv-Cpt`, and `Conv-Wr`, which represent the overheads of the aforementioned read, compute, and write operations (i.e., ❷, ❸, ❹ in Figure 2b), respectively. As shown in Figure 3c, the read and write (i.e., `Conv-Rd` and `Conv-Wr`) overheads dominate the conversion phase, while `Conv-Cpt` only accounts for 8.3%. To sum up, *in the conversion phase, the background tasks (i.e., I/O and computation) significantly degrade the performance of user I/O* ***(Challenge 2)***.

Intrinsic issues of two-phase write. Apart from the aforementioned performance penalties, two-phase write AFA engines impose two more challenges.

First, *two-phase write introduces extra metadata, which increases crash consistency cost* ***(Challenge 3)***. Specifically, to support out-of-place updates, the AFA engine maintains extra mapping tables in host memory to record where data is actually stored. As these mapping tables are updated frequently, persisting them imposes a huge burden on the host (e.g., by writing an undo or redo log [27] before every update).

Second, *two-phase write causes significant write amplification* ***(Challenge 4)***. Assume that 1$\times$ data needs writing to a

4+1 two-phase write AFA. In the replication phase, the AFA engine duplicates the data by 2× to provide guaranteed fault tolerance. In the conversion phase, it reads 1× data out for parity computation and writes 1× data and 0.5× parity back to SSDs. The total write amplification is 3.5×. These extra writes significantly shorten the lifetime of SSDs.

3.2 Key Insights

User-space storage stack. The user-space storage framework, such as Storage Performance Development Kit (SPDK) [70], is a promising solution to address ***Challenge 1***, that is, the software overheads in the storage stack. Figure 3d shows the architecture of SPDK. SPDK employs a block device abstraction called *Bdev* to perform the same functions as the block device layer in the kernel. SPDK also implements a user-space, asynchronous, polling-based *NVMe driver*. Therefore, users can directly access NVMe SSDs in user space without trapping into the intricate kernel. These designs achieve a significant acceleration of the storage stack. However, it is non-trivial to directly integrate the existing AFA designs into SPDK. This is because traditional AFA engines rely on multiple complicated locks to facilitate thread-level parallelism [4, 63, 71], which violates the lock-free principle of SPDK. *Our key insight is that the SPDK-compatible message-passing mechanism can be the alternative to locks for supporting concurrent multi-thread access.*

SSD-internal hardware resources. It is non-trivial to solve ***Challenges 2∼4*** with purely software solutions. For example, the read, compute, and write operations in the conversion phase heavily rely on the host CPU. Even if we can optimize parity computation with sophisticated algorithms, the major overheads (i.e., `Conv-Rd` and `Conv-Wr` in Figure 3c) remain unavoidable. In addition, solely software solutions are helpless for mitigating the impact of write amplification (on SSD endurance) due to the SSDs' agnostic about the characteristics of data (e.g., when it will be invalidated). Fortunately, holistic designs can be a promising solution to address these issues. *Our key insight is that the available resources in SSDs can overcome the software constraints in the existing AFA engines, making them fit for next-generation storage.*

4 ScalaAFA Overview

Inspired by the above analysis and key insights, we propose *ScalaAFA*, a high-performance AFA engine built from holistic designs, which overcomes all the aforementioned challenges by **embracing user-space storage stack** and **exploiting SSD-internal hardware resources**. Figure 4 illustrates the architecture of ScalaAFA. Note that, ScalaAFA focuses on improving the write performance of AFA engines and follows the conventional designs of read path [36]. Prior work [63, 71] has proved the scalability and high performance of read operations in available AFA solutions.

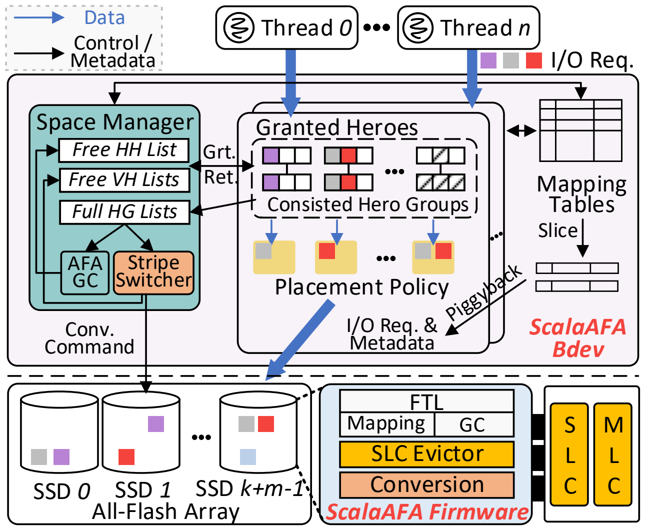


Figure 4: Architecture of ScalaAFA.

For ***Challenge 1***, we adopt SPDK [70] to take advantage of its high-performance NVMe driver and lightweight storage stack. Considering the lock-free principle of SPDK, ScalaAFA replaces the conventional lock mechanism with an SPDK-compatible message-passing scheme to avoid write collision among threads. Specifically, ScalaAFA first abstracts storage space with a novel data structure named *hero* [2]. Afterward, ScalaAFA employs a *space manager* to manage the write permissions of all the storage spaces (i.e., heroes). These permissions are granted to threads via message passing in batches. Threads can only serve write requests with the heroes where permissions are already granted.

For ***Challenge 2***, a promising solution to relieve the CPU burden is to offload the parity computation to the storage, similar to the prior in-storage processing approaches [37, 43, 53, 56]. Note that parity computation only consumes minor CPU resources (cf. `Conv-Cpt` in Figure 3c). Such computation can be easily taken with available on-device hardware (e.g., the SSD-embedded XOR engines), which is originally designed to enable SSD-internal parity computation. For instance, our prototype takes only 20 μs and 16 mW dynamic power to calculate a 64 KB parity chunk from 6 data chunks (cf. § 6.1 for details). However, this method cannot address the background read and write issues in the conversion phase, which have become the major bottleneck in two-phase write AFA (cf. `Conv-Rd` and `Conv-Wr` in Figure 3c). This is because two-phase write AFA scatters data chunks across different SSDs in the replication phase. The conversion phase requires the host CPU to copy all the corresponding data chunks to the target SSD before offloading parity computation to that SSD. Tackling this issue, we propose a novel *placement policy*, which can transparently gather data chunks of the same stripe to the SSD where parity chunks will be stored. After-

[2]Hero is the name of the I-shape piece in Tetris.

ward, in the conversion phase, ScalaAFA only needs to send a command through the *stripe switcher* to the SSD. Once receiving these commands, the *conversion* module in each SSD is in charge of computing parity chunks on device (without collecting data from other SSDs) and storing the parity locally (without writing to other SSDs).

For ***Challenge 3***, one feasible solution is to persist the mapping tables in host-side battery-backed DRAM [36]. However, it increases monetary costs and cannot provide enough fault tolerance (i.e., data cannot be located if the DRAM fails). It is also impractical to store them in SSD internal battery-backed DRAM as its limited capacity only keeps up to tens of MB of data persistent [14]. To address this, we propose to persist these mapping tables in the OOB area of SSDs. In particular, considering OOB is scattered across different physical pages of SSDs (cf. Figure 1), ScalaAFA first reconstructs the mapping tables into a segmentable data structure. It then slices and piggybacks the mappings to SSDs via write requests. Finally, ScalaAFA persists the sliced mappings in OOB when data is written to the data area. Note that writing data and its metadata in the data and OOB areas can be done by a single flash program operation, the cost of which is negligible.

For ***Challenge 4***, while our novel placement policy has eliminated extra I/O in the conversion phase, the replication phase still causes $m+1$ times write amplification. We alleviate its damage to SSD endurance by fully utilizing the durable SLC buffer in SSDs. Specifically, ScalaAFA avoids writing redundant replicas from SLC buffer back to the vulnerable MLC blocks. This is because these replicas will be invalidated in the conversion phase soon (cf. § 2.2). To this end, ScalaAFA deploys *SLC evictor*, which gives a low priority to these replicated data when selecting victims to be evicted.

5 Design Details of ScalaAFA

5.1 Storage Space Abstraction

Figure 5 shows the storage space abstraction of ScalaAFA. From the user's perspective, ScalaAFA functions as a standard block device, which exposes a continuous space, referred to as *user address space*, and enables random writes as well as in-situ updates. The user-written data will be sequentially logged into an intermediary space, named *AFA address space*. Contiguous k chunks (also named k *slots*) in AFA address space are orchestrated as a *stripe*. Each stripe documents the storage locations of the k data chunks and their corresponding m parity chunks in the SSDs (i.e., in which SSD and the SSD logical address in that SSD). ScalaAFA partitions the storage space of SSDs into two virtual areas: *normal area* and *transient area*. The former is used to store long-term data (e.g., data chunks after conversion) while the latter accommodates transient data (e.g., replicated data chunks). Note that the size of the transient area is variable and determined by the total capacity of the SLC buffers (cf. § 2.1) within SSDs.

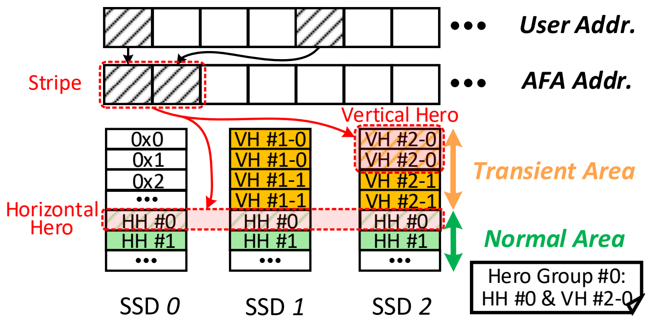


Figure 5: Space abstraction of ScalaAFA (k=2 and m=1).

ScalaAFA organizes the two areas as sets of *horizontal hero* (HH) and *vertical hero* (VH), respectively. In the normal area, $k+m$ chunks with the same SSD logical address from $k+m$ SSDs are grouped as an HH (e.g., `HH #0` in Figure 5), while contiguous k chunks in the same SSD make up a VH in the transient area (e.g., `VH #2-0` in `SSD 2`). Further, ScalaAFA composes one HH and m VHs from m different SSDs as a *hero group* (HG), such as the `HG #0` that consists of `HH #0` and `VH #2-0` (we will describe the rationale of HG in § 5.3). With the abstraction of HG, ScalaAFA can track the storage locations of data and parities by associating stripes with HGs.

5.2 Enable Lock-free Multi-Thread Access

ScalaAFA supports multiple I/O threads to access it concurrently. Similar to conventional block devices, ScalaAFA does not provide sequential consistency [12]. Thus, there is no need to prevent I/O threads from accessing the same user address simultaneously. However, to prohibit these threads from mapping different user addresses to the same AFA address or mapping different AFA addresses to the same SSD storage location (i.e., HG), ScalaAFA has to manage the write permission of AFA address space and SSD storage space. ScalaAFA achieves this with a user-space-friendly (i.e., lock-free) message-passing scheme.

Considering the communication overhead, we first simplify address mappings to reduce the need for communicating. Specifically, we fix the mappings between stripes and HHs. One stripe is bound with one HH that has the same offset. For example, the first stripe in the AFA address space is bound with the first HH in the normal area (i.e., `HH #0` in Figure 5). Therefore, we only need to assure that different user addresses will not be mapped to the same HH and VHs (i.e., the same HG) without considering the intermediary AFA address. To this end, the *space manager* is responsible for granting and retrieving the write permissions of heroes. Figure 6 gives an example of our solution.

Grant. Initially, all write permissions of heroes belong to the space manager (i.e., *Free HH List* and *Free VH Lists*). When an SPDK I/O thread is set, it asks the space manager for VHs and HHs. Then, the space manager grants heroes in batches

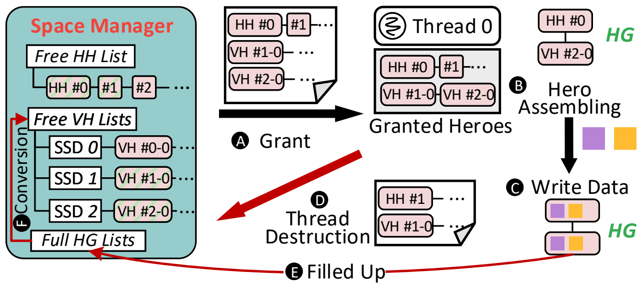


Figure 6: An example of write permission management.

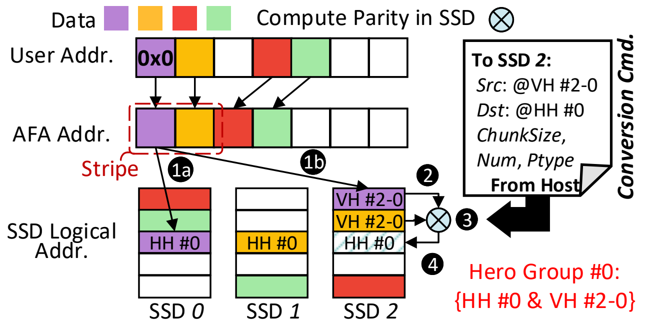


Figure 7: Procedure of conversion offloading (k=2 and m=1).

(e.g., 1024 $* m$ VHs from m SSDs and 1024 HHs at once, Ⓐ). Afterward, if an I/O thread wants to write data chunks to a certain user address, it first assembles granted HH and VHs as an HG (Ⓑ). Then, the data will be written to the HG (Ⓒ), and the user address will be mapped to the corresponding stripe that is bound with the HH. If heroes are run out, the I/O thread will ask the space manager for more grants.

Retrieve. The space manager can retrieve heroes in four ways: (1) When an I/O thread is being destructed, it returns unused heroes to the space manager (Ⓓ); (2) When an HG is filled up by an I/O thread, it will be inserted into a list (i.e., *Full HG Lists*) associated with this thread (Ⓔ). When the CPU is idle or heroes are running out (i.e., the conversion phase starts), the space manager scans the list of each I/O thread and conducts conversion for HGs in the list. Note that, after conversion phase, data and parity chunks are stored in the more space-efficient striping layout (cf. § 2.2). Therefore, the saved space (heroes) can be recycled (Ⓕ); (3) If no more heroes can be granted, the space manager broadcasts a message to recall unused heroes from all I/O threads. The more often a thread requests heroes over a period of time, the fewer heroes will be recalled from that thread; (4) Out-of-place write invalidates stale data and diminishes available storage space. The space manager executes AFA-level GC [36] to recycle these spaces and reuse heroes.

5.3 Evolve the Write Path

Replication phase with placement policy. With the aforementioned space abstraction, we can place data chunks in heroes to transparently gather them in the SSDs where conversion will be executed. Figure 7 illustrates this process. In the replication phase, if one data chunk is written to the stripe, it will be replicated $m+1$ times. One of them (named *primary copy*) will be stored in HH while the other m copies (named *backup copies*) are written to m VHs. Taking Figure 7 as an example, the user sends a write request to `0x0` and the data chunk is logged in the first slot of the stripe, which is mapped with `HG #0`. Therefore, in the replication phase, the primary copy is written to the first slot of `HH #0` (1a), and the backup copy is placed in the first slot of `VH #2-0` (1b). Repeating this process, data chunks belonging to the same stripe will be transparently gathered in VHs (e.g., the chunks in `VH #2-0`).

Conversion phase with offloading. Afterward, when the host is idle or VHs are running out, ScalaAFA starts conversion phase and offloads the conversion tasks to SSDs where the VHs locate (e.g., `SSD 2`). To make the SSDs aware of this, ScalaAFA extends the NVMe command set with a *conversion command*. We include the following information in the new command. *Src*: offset of the VH in SSD; *Dst*: the logical address where parity chunk will be stored; *ChunkSize*: the size of chunks; *Num*: the number of data chunks in one stripe (i.e., k); *Ptype*: the type of parity chunk that will be computed. *Ptype* is used to determine the parity computing method. For example, in Reed-Solomon codes, *Ptype* is the row number of the encoding vector in Vandermonde matrix [64]. Once receiving the command, the SSD executes conversion locally. Specifically, the SSD controller reads *Num* chunks with *ChunkSize* from *Src* (❷) to its internal DRAM. It then computes the parity chunk according to *Ptype* (❸) and writes the parity chunk to *Dst* (❹). Finally, the SSD cleans the data in VH (e.g., `VH #2-0`) by marking the flash pages as invalid. Note that, after ❹, the HH (e.g., `HH #0`) has stored data and parity chunks in striping layout. Thus, ScalaAFA avoids extra writes for scattering chunks (cf. § 2.2). By doing these, ScalaAFA successfully reduces the host-SSD I/O in conversion phase thereby mitigating the performance degradation in this phase.

5.4 Persist the Metadata

Mapping table. As shown in Figure 8, ScalaAFA maintains two mapping tables in host memory: *AFA Mapping Table (AMT)* and *Hero Group Mapping Table (HGMT)*. AMT maps user addresses to AFA addresses, which records a 32-bit *AFA chunk number* for each chunk in user address space. HGMT translates AFA addresses (i.e., stripes) to SSD storage locations (i.e., HGs). Since stripes and HHs are one-to-one mapped (cf. § 5.2), ScalaAFA only needs to record the mapping information between stripes and VHs. To this end, each row in HGMT represents a stripe and maintains a pointer to a VH list that records the m VHs of this HG. Each item in the VH list contains two components: an 8-bit *SSD ID* to record which SSD the VH locates in and a 32-bit *VH number*, which

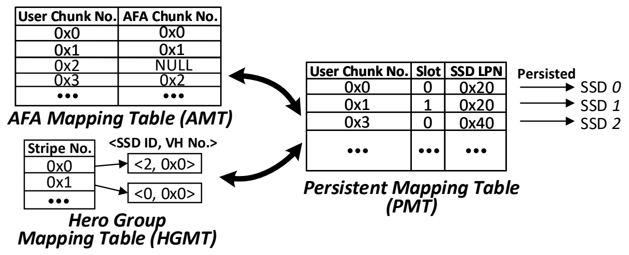


Figure 8: Mapping tables in ScalaAFA.

is the offset of the VH in that SSD. If a stripe is in striping layout (i.e., after conversion), its VHs have been recycled (cf. § 5.3) and the VH list is NULL. Note that, in ScalaAFA, the chunk size is 64 KB by default. Therefore, to create a 6+3 AFA with 2TB SSDs, ScalaAFA needs only about 928 MB (0.005%) memory to maintain AMT and HGMT.

Persisting the mapping table. As two-phase write AFA has to update its mapping tables frequently, persisting such metadata, in turn, introduces huge software overheads (e.g., by persisting an undo/redo log [27] before every update). One possible solution to tackling this challenge is to maintain the metadata in SSD-internal OOB. However, OOB is scattered across different physical pages and the size of OOB in each flash page is tens of bytes (cf. § 2.1). It is difficult to store the entire AMT and HGMT in one OOB.

To better utilize the constrained OOB, we first convert AMT and HGMT to a segmentable mapping table, called *persistent mapping table (PMT)*. As shown in Figure 8, each entry in PMT corresponds to a chunk in the user address space (i.e., *user chunk number*). It contains an 8-bit *slot number* and a 32-bit *SSD LPN*. The former represents the slot in the stripe where the chunk is placed. The latter records the SSD logical address of the chunk's primary copy (i.e., the copy placed in HH, cf. § 5.3). For example, `user chunk 0x0` is placed in `slot 0` of the first stripe in AFA address space, and its primary copy is stored in `0x20` of `SSD 0` (cf. Figures 7 and 8). Since logical pages are one-to-one mapped to physical pages in SSD, we slice PMT based on *SSD LPN* and store it in the corresponding OOB. Specifically, we piggyback the user chunk number and slot number in write requests and send them to SSDs with a 32-bit timestamp. When writing data to flash physical pages, the SSD controller persists the metadata to OOB via the same program operations (i.e., without extra write). To guarantee fault tolerance of PMT, we also compute parity codes for metadata. Specifically, in the conversion phase, the conversion module reads both data and metadata from the same physical page with a single read operation. It then computes parity codes for them simultaneously. Finally, the parity codes of data and metadata will be written back together and stored in the same page (cf. § 5.3). Note that our solution only utilizes 72 bits *spare* space of OOB (cf. § 2.1) per chunk to persist our customized metadata, which does not affect other OOB-based functions (e.g., ECC).

Recovery of mapping table. Take Figure 8 as an example to depict how we recover AMT and HGMT. For simplicity, we assume the chunk size, flash page size, and transient area size of each SSD are 64, 4, and 128 KB. First, we scan OOB to recover PMT. However, this process can be time-consuming as it needs to scan all flash pages. We accelerate this process by adopting periodical checkpoints [72] and only scanning the un-checkpointed metadata. From PMT, we know that the primary copy of `user chunk 0` is stored at `0x20` of `SSD 0`, which is the first chunk of the normal area (i.e., `HH #0` in Figure 7). Therefore, `user chunk 0` is stored in the first stripe of AFA address space. Considering the slot number of `user chunk 0` is `0`, the chunk is stored in the first chunk of AFA address space, i.e., its AFA chunk number is `0`. Repeating this, we can restore AMT. As `user chunks 0` and `1` are stored in `SSD 0` and `1`, respectively, we know that their parity chunk is stored in `SSD 2` (i.e., SSD ID). We then read the OOB from `0x20` of `SSD 2` and verify if it is the parity. If so, this stripe is in the striping layout and the corresponding item in HGMT is NULL. Otherwise, we need to scan the transient area of `SSD 2` and match the OOB with `user chunks 0` and `1` to locate their VH (i.e., VH number). Thereby, we recover the HGMT.

5.5 Reduce the Impact of Write Amplification

While our conversion offloading has eliminated extra I/Os in conversion phase (cf. § 5.3), the replication phase still causes $m+1$ times write amplification, which significantly shortens the SSD lifetime. Our key insight is that backup copies (i.e., data in VHs) in two-phase write are transient and will be invalidated soon in the conversion phase. Thus, there is no need to flush them from the durable SLC buffer to the vulnerable MLC blocks. Unfortunately, SSDs are unaware of which data are transient. Tackling this issue, we propose a holistic design. Specifically, when writing backup copies to SSDs, ScalaAFA marks them as transient data by flagging one bit in the write requests. Once receiving these requests, SSD records the transient tags. Afterward, when the SLC buffer is full, the SLC evictor selectively evicts SLC blocks by considering the transient tags. To be specific, the SLC evictor will evict the SLC block with the highest evicting score, which can be calculated with the following formula:

$$score = factor * num\_invalid - num\_transient$$

$num\_invalid$ and $num\_transient$ are the numbers of invalid and transient pages in this block, while $factor$ is a constant. We set it as 1 empirically in this paper. If one block has more invalid pages, we evict it in higher priority, because it generates fewer writes in this eviction. On the other hand, if one block contains more transient pages, we evict it in lower priority. Since these pages will be invalidated soon, there is no need to flush them to MLC blocks immediately.

5.6 Implementation

ScalaAFA employs a daemon thread to play the role of space manager. When an AFA is created, we initialize the daemon thread and bind it to a fixed CPU core via *spdk_env_thread_launch_pinned()*. All of our modifications to NVMe protocols are based on NVMe Base Specification 2.0c [5] and NVMe Command Set Specification 1.0c [6]. The conversion command is implemented as an NVMe IO command. The transient tag uses one reserved bit in NVMe write command. We piggyback the user chunk number, lot number, and timestamp to SSDs and program them in OOB with the support of NVMe Protection Information feature [6]. We develop ScalaAFA in SPDK v22.05 [8] with 6K LOC. The modification to SSD firmware is implemented in a popular SSD emulator [47] with 1K LOC.

6 Evaluation

6.1 Experimental Setup

Methodology. We use Femu [47], a QEMU-based SSD emulator, to evaluate our holistic designs. We set up the QEMU virtual machine to run on Linux v5.11.0 with 32 CPU threads, 32 GB DRAM, and up to 8 NVMe SSDs. We configure the emulated SSDs as high-performance storage devices with 7500 MB/s and 4890 MB/s peak read and write bandwidths, respectively. We set the size of the SLC buffer to 4 GB. These configurations match with commercial high-end SSD products [2]. In addition, our simulator employs an XOR engine in the SSD controller, which is the same as the state-of-the-art SSD devices [15, 16, 67]. We get configurations of the XOR engine from Xilinx xc7a200t FPGA with 200 MHz clock. It takes 20 μs and 16 mW dynamic power to calculate a 64 KB parity chunk from 6 data chunks. The conversion of a stripe in SSD costs 103 μs in total. The key configurations in our experiments are listed in Table 1.

AFA platforms. We compare ScalaAFA with five other popular AFA engines. (1) `mdraid` [4]: the default stripe write AFA engine implemented in Linux kernel; (2) `ScalaRAID` [71]: a state-of-the-art stripe write AFA engine, which mitigates the software overheads of mdraid with fine-grained locks and improves its performance to some extent; (3) `stRAID` [63]: another stripe write AFA engine, which alleviates the software overheads in mdraid with a run-to-complete I/O processing scheme; (4) `RAID5F` [7]: an ***incomplete*** stripe write AFA engine in SPDK, which has no crash consistency and ***can only serve RAID 5 full-stripe I/O*** (i.e., the I/O size is equal to the stripe size). We consider the performance of `RAID5F` is close to the ideal case of software-only stripe write AFA designs; (5) `FusionRAID` [36]: a state-of-the-art two-phase write AFA engine; (6) `ScalaAFA`: the user-space AFA engine that includes all the designs proposed in this paper. We set the chunk size in all the AFA platforms as 64 KB.

Host System		Femu		Software	
CPU	Intel Xeon 5320	**Virtual machine**	32 CPU threads	**Linux kernel**	v5.11.0
	1×26 Core / 2.2 GHz		32 GB DRAM		
	with hyper-threading	**Flash**	8 Channel / 12 Die /	**fio**	v3.30
Mem.	8×64 GB / DDR4		1 Plane / 352 Block /	**perf**	v5.11
SSD	Samsung 980 Pro		512 Page / 4 KB	**mdadm**	v4.1
	R/W: 7000/5200 MB/s	**Bw.**	R/W: 7500/4890 MB/s	**SPDK**	v20.05

Table 1: System configurations.

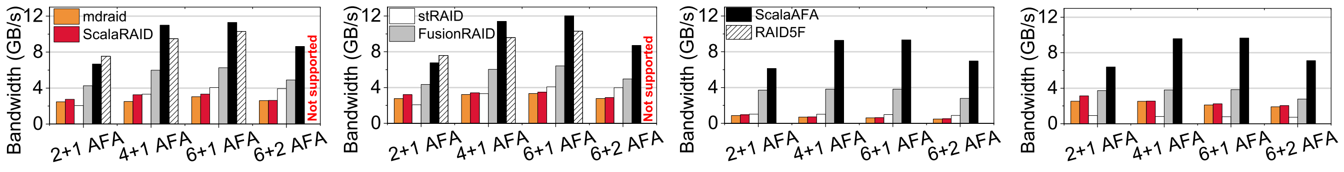
Trace	Wr. cnt. (Kops)	Rd. cnt. (Kops)	Avg. wr. len. (KB)	Avg. rd. len. (KB)	Data wr. (GB)	Data rd. (GB)
proxy0	12135.4	383.5	4.6	8.3	53.8	3.1
prn	7753.0	9066.3	10.4	22.5	76.8	194.5
src2	2201.1	1171.3	23.2	54.8	48.8	61.2
CFS	1173.0	3304.1	12.6	8.7	14.1	27.3
DAP	475.3	610.4	97.2	62.1	44.1	36.2
webmail	6381.9	1413.8	4.0	4.0	24.3	5.4

Table 2: Characteristics of real workloads.

Workloads. We evaluate ScalaAFA with various benchmark suites and applications. Specifically, we measure the performance of different AFA engines by employing fio v3.30 [13] to execute microbenchmarks. By default, we use a single I/O thread to generate asynchronous I/O requests. We also consider the multi-thread scenarios in scalability testing (cf. § 6.2). To reflect the impacts of block devices on system performance, we evaluate ScalaAFA with block I/O traces [45]. We select six representative workloads from both industries (Microsoft MSRC [38] and MSPC [55]) and academia (FIU [61]) including both write-dominated (e.g., proxy0) and read-write mixed (e.g., DAP) traces with varied I/O sizes. Table 2 summarizes the key characteristics of the selected workloads. We also conduct a comparison on RocksDB [24], a popular KV store, with `db_bench` [25], which demonstrates the end-to-end performance improvement brought by our designs.

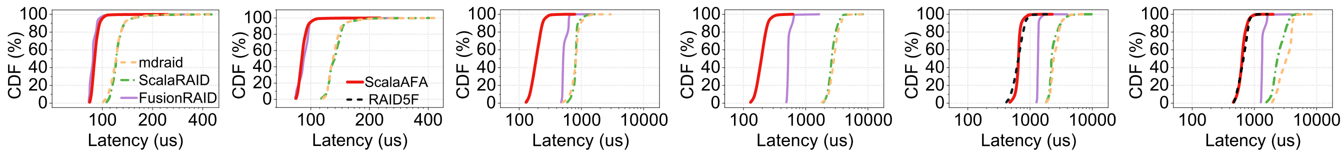
6.2 Overall Performance

Throughput. We compare the write throughput of different AFA engines, which is shown in Figure 9. We set the I/O depth to 32. `ScalaRAID` slightly outperforms `mdraid`. This is because `ScalaRAID` aims at mitigating the overheads of lock mechanism. However, such overheads are not severe when employing only one I/O thread (cf. § 3.1). `stRAID` also achieves 17.9% higher full-stripe write throughput than `mdraid`, on average, thanks to its run-to-complete I/O processing scheme. As shown in Figure 9c, `mdraid`, `ScalaRAID` and `stRAID` all have poor performance in 64 KB random write accesses because stripe write AFA are unfriendly to partial write. Since `mdraid` and `ScalaRAID` opportunistically aggregate multiple contiguous write requests as a full-stripe write by employing a DRAM cache [4, 63], they achieve higher performance for 64 KB sequential write than `stRAID` (cf. Figure 9d). `FusionRAID` outperforms `stRAID` by 3.6× in 64 KB random write, on average. This is because it absorbs partial write requests with replication. Also, it updates data chunks in an out-of-place way, which avoids the tedious read-construct-



(a) Full-stripe random write. (b) Full-stripe sequential write. (c) 64 KB random write. (d) 64 KB sequential write.

Figure 9: Comparison of write throughput on microbenchmarks.

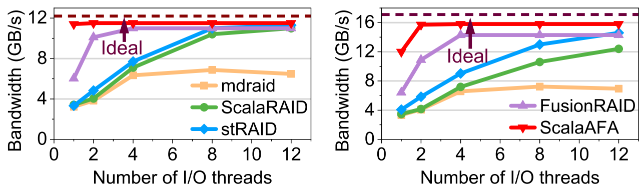


(a) 4 KB sequential. (b) 4 KB random. (c) 64 KB sequential. (d) 64 KB random. (e) Full-stripe sequential. (f) Full-stripe random.

Figure 10: Comparison of write latency CDF on microbenchmarks.

write procedure. ScalaAFA outperforms FusionRAID in all types of I/O access patterns. For example, in 64 KB sequential write, ScalaAFA further improves the write throughput by 1.6×, 2.4×, 2.4×, and 2.5× in 2+1, 4+1, 6+1, and 6+2 AFAs, respectively. This is because the key techniques in ScalaAFA, such as the user-space design and conversion offloading, significantly reduce the CPU burdens. RAID5F outperforms ScalaAFA by 12.4% in 2+1 AFA, but has worse performance (17.4% and 13.1%) in 4+1 and 6+1 AFAs. This is because conversion overhead is minor in 2+1 AFA whereas becoming severe as the number of SSDs increases. With our hardware/software co-designs, ScalaAFA not only offloads parity computation to SSDs but also eliminates the penalty of data preparation (cf. §5.3).

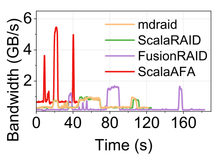
Latency. Figure 10 shows the cumulative distribution function (CDF) of all the tested AFA engines in terms of latency. We omit stRAID in this comparison for the implementation bugs in its open-sourced codes. We select 4+1 AFAs in this test. We set the I/O depth to 1 for 4 KB write while it is 32 for both 64 KB and full-stripe write. There is no visible difference between ScalaRAID and mdraid in the 4 KB write scenario. However, ScalaRAID reduces the average and 99th percentile latency by 22.9% and 15.0% for full-stripe random write, where the lock overhead is not negligible. FusionRAID outperforms mdraid in terms of both average and tail latencies. For example, it reduces the average and 99th percentile latency to 66.1% and 48.2%, respectively, for 64 KB sequential write. ScalaAFA shows significantly better CDF profiles in all scenarios. Compared with FusionRAID, ScalaAFA further reduces the average and 99th percentile latency by 52.7% and 41.9% for full-stripe write, respectively. This is because ScalaAFA not only benefits from the high-performance user-space storage stack but also mitigates performance degradation caused by conversion with SSD architectural innovations. Although RAID5F is considered an ideal software-only AFA solution, ScalaAFA even achieves 4.8% shorter 99th percentile tail latency than RAID5F thanks to our exploration of hardware resources (e.g., conversion offloading).



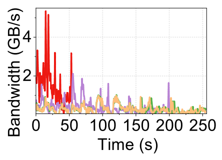
(a) 4+1 AFA. (b) 6+1 AFA.

Figure 11: Comparison of scalability.

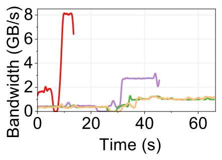
Scalability. We further measure the scalability of different AFA engines by varying the number of I/O threads from 1 to 12, which is shown in Figure 11. For mdraid, ScalaRAID, and stRAID, we employ the same number of worker threads as I/O threads (as suggested by prior work [63, 71]). We also plot the ideal throughput of two-phase write AFA engines (i.e., $(k+m)/(m+1)*S$, where S is the peak bandwidth of a single SSD). mdraid gains limited benefits from extra threads. It achieves peak performance when employing 8 I/O threads. This is because lock overhead caused by multi-thread access dominates the time cost of the storage stack [71]. Benefited from the lower software overhead (i.e., fine-grained locks [71] and run-to-complete I/O processing scheme [63]), ScalaRAID and stRAID are more scalable than mdraid. They achieve 11.0 GB/s and 11.3 GB/s, respectively, in 4+1 AFA with the cost of 12 I/O threads. In comparison, ScalaAFA can achieve 11.4 GB/s with only one I/O thread, which is almost the same as the ideal case for 4+1 AFA. The minor performance difference between ScalaAFA and the ideal case comes from the background tasks within SSDs (e.g., conversion). For 6+1 AFA, FusionRAID achieves only 6.4 GB/s with one I/O thread, 37.5% of the ideal performance. This



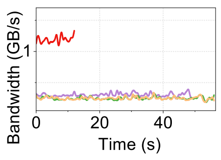
(a) proxy0.



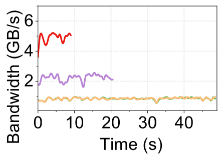
(b) prn.



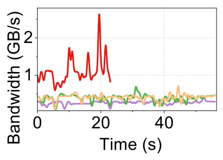
(c) src2.



(d) CFS.

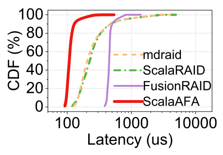


(e) DAP.

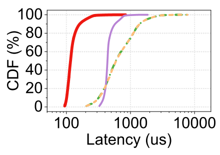


(f) webmail.

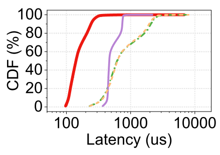
Figure 12: Comparison of write throughput on real workloads.



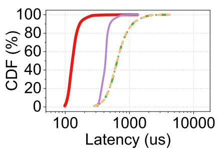
(a) proxy0.



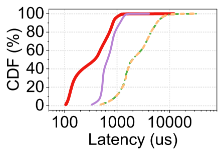
(b) prn.



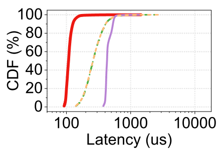
(c) src2.



(d) CFS.



(e) DAP.



(f) webmail.

Figure 13: Comparison of write latency CDF on real workloads.

is because `FusionRAID` suffers from the huge CPU burden caused by the tedious storage stack (cf. § 3.1). `ScalaAFA` improves the throughput to 12.0 GB/s and 15.7 GB/s when employing 1 and 2 I/O threads, respectively. This is because `ScalaAFA` imposes minor overhead on the host CPU, thereby allowing I/O threads to fully exploit the bandwidth of SSDs.

6.3 Analysis of Real Workloads

Throughput. Figure 12 illustrates the throughput of different AFA engines in different real workloads. We set the I/O depth to 32 in this test. Compared with `mdraid`, `FusionRAID` achieves 34.1% higher throughput in all workloads, on average. Based on `FusionRAID`, `ScalaAFA` further improves the average throughput by 2.9×, 2.2×, and 1.2× in workloads `prn`, `src2`, and `DAP`, respectively. Moreover, it shortens the completion time of all workloads to 30.0%, on average, compared with `FusionRAID`. Workload `DAP` has the largest average write request size, in which `ScalaAFA` achieves the highest average throughput (i.e., 4.8 GB/s) among all workloads. All AFA solutions exhibit the lowest throughput in `webmail`. This is because `webmail` has the smallest write request size, which cannot fully utilize the SSD-internal parallelism. However, the bandwidth of `ScalaAFA` still exceeds that of `FusionRAID` by 2.8×.

Latency. Figure 13 shows the latency CDFs in different real workloads. Compared with `mdraid`, `FusionRAID` reduces the average and 99th percentile latency by 13.8% and 59.7%, respectively. In all workloads, `ScalaAFA` achieves the best CDF profiles. For example, `ScalaAFA` reduces the average latency to 24.2% and 26.9% in `proxy0` and `prn`, compared to `FusionRAID`. `ScalaAFA` reduces the 99th percentile latency of `FusionRAID` by 59.1%, 65.1%, and 72.5% in the `src2`, `CFS`, and `webmail`, respectively. This is because ScalaAFA not only offloads conversion to SSDs that eliminates background I/O but also benefits from the lock-free user-space designs.

6.4 End-to-end Evaluation

To demonstrate the superiority of ScalaAFA on applications, we conduct an end-to-end evaluation on RocksDB [24], a popular KV store. We run RocksDB on Ext4 and BlobFS [60] file systems for kernel-space (i.e., `mdraid` and `FusionRAID`) and user-space (i.e., `ScalaAFA`) AFA engines, respectively. We use five representative benchmarks from `db_bench` including both write-dominated (e.g., `fillrandom`) and read-write-mixed (e.g., `fillseekseq`) workloads. We set the key and value sizes as 16 B and 1 KB, respectively. Figure 14 illustrates the throughput comparison of different AFA engines. Thanks to the utilization of SSD-internal resources and the high-performance storage software stack, `ScalaAFA` achieves the highest throughput in all workloads. Specifically, `ScalaAFA` outperforms `mdraid` by 41.4% in all tests, on average. Compared with `FusionRAID`, `ScalaAFA` improves the average throughput by 31.9%, 41.1%, and 23.8% in `fillrandom`, `fillseq`, and `overwrite`, respectively. The improvement decreases to 8.1% in `fillseekseq`. This is because, in this workload, lots of I/O requests are absorbed by the DRAM cache [19] of RocksDB without entering AFAs.

6.5 Benefits of Individual Techniques

In this section, we evaluate the benefits brought by each key design in ScalaAFA. We implement two new AFA engines, which incorporate only parts of ScalaAFA techniques. (1) `Scala-CO`: based on ScalaAFA, we execute the conversion in the host. Specifically, in the conversion phase, the daemon thread reads the data chunks from the SSDs, computes the parity chunks, and finally writes them back to SSDs; (2) `Scala-WR`: based on ScalaAFA, we do not set the replicated

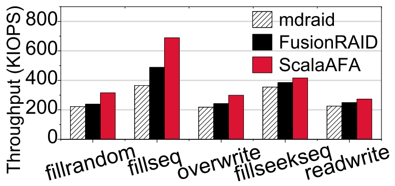


Figure 14: Comparison in RocksDB.

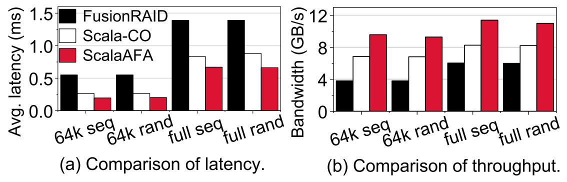


Figure 15: Sources of performance improvement.

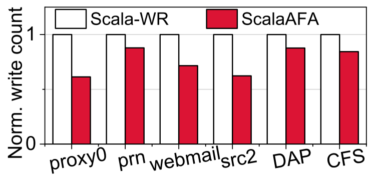


Figure 16: Write count reduction.

chunks as low priority. We use 4+1 AFA in this evaluation.

Sources of performance improvement. We compare the write throughput and latency of different AFA engines in different I/O patterns, which is shown in Figure 15.

• *Benefit of lock-free user-space design:* Compared with `FusionRAID`, `Scala-CO` reduces the average latency by 45.2% and improves the write throughput by 58.4%, for all scenarios on average. Note that, FusionRAID [36] recommends persisting metadata with host-side battery-backed DRAM, which is impractical for productions considering its monetary cost and insufficient fault tolerance. In our reproduced version, `FusionRAID` employs the same metadata persistence scheme as `ScalaAFA`. Therefore, the performance differences between `FusionRAID` and `Scala-CO` mainly come from our lock-free user-space design (i.e., message-passing-based permission management for multi-thread access).

• *Benefit of conversion offloading:* In comparison to `Scala-CO`, `ScalaAFA` reduces the average latency by 36.1%. In terms of write throughput, `ScalaAFA` improves the performance by 39.6%, 36.1%, 37.9%, and 34.0% for 64 KB sequential and random write, full-stripe sequential and random write, respectively. This is because the conversion takes up 77.6% of the CPU ticks of the daemon thread, which significantly disrupts the space manager running on the daemon thread thereby blocking user I/Os. In contrast, `ScalaAFA` achieves higher throughput in all testing scenarios thanks to the conversion offloading design.

Write amplification reduction on MLC blocks. To evaluate how much our design can mitigate the impact of write amplification, we set the size of the SLC buffer to 4 GB and count the number of writes on MLC blocks. The results are shown in Figure 16. Compared to `Scala-WR`, which performs the same as existing two-phase write engines [26, 36, 65], `ScalaAFA` can decrease write count in all workloads. For example, it alleviates the write amplification by 38.6%, 12.2%, 37.7%, and 28.4% in `proxy0`, `prn`, `src2`, and `webmail`, respectively. This is because, with our holistic write amplification reduction scheme, ScalaAFA evicts the replicated data to MLC blocks in a low priority as these data will be invalidated right after the conversion phase. Therefore, `ScalaAFA` reduces the amount of data written to the vulnerable MLC blocks.

7 Related Work and Discussion

Overheads of AFA engines. Multiple studies [31, 35, 36, 50, 51, 63, 71] have been proposed to mitigate the software overheads of AFA engines. ScalaRAID [71] and stRAID [63] partially mitigate the overhead of multi-thread access with fine-grained lock schemes and distributed data structures. FusionRAID [36] reduces the overhead of partial write with two-phase write scheme. However, the overheads of parity computation and background I/O (e.g., conversion) still exist in the prior work and impose a huge burden on the host CPU, especially in I/O-intensive scenarios. In contrast, the holistic designs in ScalaAFA not only alleviate the lock overheads but also eliminate the conversion penalty. Moreover, our hardware-based metadata persistence scheme can further accelerate AFA. EAR [50] tries to reduce the data reads/writes in conversion phase with a sophisticated flow graph matching algorithm. However, EAR still computes parities out of place and requires extra communications among storage nodes for forwarding computing results (i.e., map-reduce). Considering the lack of proactive communication capability between SSDs, it is hard to adopt EAR for AFA scenario. In comparison, ScalaAFA can generate parities locally, which completely eliminates the communications among SSDs. Besides, Much prior work [16, 21, 30, 40–42, 46, 48, 51, 69] has extensively studied the performance degradation caused by SSD GC in AFA. In contrast, ScalaAFA focuses on optimizing the general write scenario and is orthogonal to these approaches.

In-storage processing. As high-performance SSDs usually equip abundant computing resources (e.g., ARM processors and XOR engine), much prior work [22, 37, 43, 44, 52, 53, 56] proposes to offload tasks to SSDs. HolisticGNN [43] offloads GNN operators to SSDs. Cognitive SSD [53] accelerates deep learning with on-device resources. Willow [56] extends the same idea to a general-purpose framework, which allows applications to offload data-intensive operands to the underlying SSDs. ScalaAFA extends this idea by taking the system overheads of the AFA scenario into consideration.

Comparison summary. Taking the discussion of related work into consideration, we summarize the key differences between prior work and our proposed ScalaAFA in Table 3.

Trade-off in ScalaAFA. Compared to traditional software-only designs, the main obstacle of implementing ScalaAFA is the modification of SSD firmware. Limited by SSD vendors, it is difficult to achieve this in off-the-shelf SSDs. However, we

Challenge	***Challenge 1***		***Challenge 2***	***Challenge 3***	***Challenge 4***
Overhead	**User-kernel ctx. switch**	**Lock-based thread sync.**	**Conversion**	**Metadata persist.**	**Write amplification**
mdraid	✘	✘	✘	✘	✘
ScalaRAID	✘	✓✘	✘	✘	✘
stRAID	✘	✓✘	✘	✘	✘
EAR	✘	✘	✓✘	✘	✘
ScalaAFA	✓	✓	✓	✓	✓

✓ = Solved; ✓✘ = Mitigated; ✘ = Ignored.

Table 3: Overall comparison across different AFA solutions.

believe in its prospects, given the significant benefits brought by our holistic designs (cf. § 6) and the emergence of in-storage processing.

8 Conclusion

Existing AFA engines fail to adopt next-generation high-performance SSDs because of the huge software overhead caused by the storage stack and AFA internal tasks. Tackling this issue, we propose a new user-space AFA engine with holistic designs, called *ScalaAFA*, which is tightly integrated into SPDK while harnessing SSD built-in resources to deliver scalable performance at low CPU costs. To be specific, ScalaAFA employs a message-passing-based permission management mechanism for concurrent access thereby conforming to the lock-free principle of SPDK. ScalaAFA offloads the conversion tasks to SSDs, which reduces the CPU burden. Lastly, ScalaAFA addresses the metadata persistence and write amplification issues by exploiting SSD architectural innovations. Evaluation results reveal that ScalaAFA improves the write throughput to 2.5× and reduces the average latency by 52.7% compared to the state-of-the-art AFA solutions.

Acknowledgement

We thank 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. Dr. Mao is supported in part by the National Natural Science Foundation of China under Grant No. U22A2027 and No. 61972325. Dr. Jung is supported in part by NRF2021R1A2C4001773, IITP 2021-0-00524, IITP 2022-0-00117, IITP RS-2023-00221040, G01230749, SRFC-IT2302-05, Samsung HiPER (G01220296), and KAIST IDEC. The corresponding author is Jie Zhang.

References

[1] Netapp inc. data ontap 8. `http://www.netapp.com/us/products/platform-os/data-ontap-8/`, 2010.

[2] Samsung 980pro nvme ssd. `https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/`, 2020.

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

[4] mdraid layer. `https://github.com/torvalds/linux/tree/master/drivers/md`, 2022.

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

[6] Nvm express nvm command set specification 1.0c. `https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0c-2022.10.03-Ratified.pdf`, 2022.

[7] Raid5f. `https://github.com/spdk/spdk/tree/master/module/bdev/raid/raid5f.c`, 2022.

[8] Spdk v22.05. `https://github.com/spdk/spdk/tree/v22.05.x`, 2022.

[9] Dell dell powerstore 500t storage array. `https://www.delltechnologies.com/asset/en-ca/products/storage/technical-support/dell-powerstore-gen2-spec-sheet.pdf`, 2023.

[10] Samsung pm1743. `https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/`, 2023.

[11] Ahmed Izzat Alsalibi, Sparsh Mittal, Mohammed Azmi Al-Betar, and Putra Bin Sumari. A survey of techniques for architecting slc/mlc/tlc hybrid flash memory–based ssds. *Concurrency and Computation: Practice and Experience*, 30(13):e4420, 2018.

[12] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[13] Jens Axboe. Flexible i/o tester. `https://github.com/axboe/fio`, 2019.

[14] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th*

Annual International Symposium on Computer Architecture (ISCA), pages 425–438, 2018.

[15] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for flash-based ssds. In *2021 USENIX Annual Technical Conference (ATC)*, pages 689–703, 2021.

[16] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channelssd subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST)*, pages 359–374, 2017.

[17] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 233–242, 2015.

[18] Zhichao Cao. *High-Performance and Cost-Effective Storage Systems for Supporting Big Data Applications*. PhD thesis, University of Minnesota, 2020.

[19] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdbkey-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*, pages 209–223, 2020.

[20] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[21] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software orchestrated flash array. In *Proceedings of International Conference on Systems and Storage*, pages 1–11, 2014.

[22] Hyeokjun Choe, Seil Lee, Hyunha Nam, Seongsik Park, Seijoon Kim, Eui-Young Chung, and Sungroh Yoon. Near-data processing for differentiable machine learning models. *arXiv preprint arXiv:1610.02273*, 2016.

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

[24] Facebook. Rocksdb. http://rocksdb.org/, 2015.

[25] Facebook. Performance benchmarks. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2021.

[26] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th annual workshop on petascale data storage*, pages 6–10, 2009.

[27] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system r database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.

[28] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of nand flash memory. In *FAST*, volume 7, pages 10–2, 2012.

[29] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[30] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST)*, pages 263–276, 2016.

[31] Brian Hickmann and Kynan Shook. Zfs and raid-z: The über-fs? *University of Wisconsin–Madison*, 2007.

[32] Seongcheol Hong and Dongkun Shin. Nand flash-based disk cache using slc/mlc combined flash memory. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 21–30, 2010.

[33] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. Flexecc: Partially relaxing ecc of mlcssd for better cache performance. In *2014 USENIX Annual Technical Conference (ATC)*, pages 489–500, 2014.

[34] Soojun Im and Dongkun Shin. Comboftl: Improving performance and lifespan of mlc flash memory using slc flash buffer. *Journal of Systems Architecture*, 56(12):641–653, 2010.

[35] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. The pitfalls of deploying solid-state drive raids. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, pages 1–13, 2011.

[36] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *19th USENIX Conference on File and Storage Technologies (FAST)*, pages 355–370, 2021.

[37] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–12, 2013.

[38] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128, 2008.

[39] Aleksandr Khasymski, M Mustafa Rafique, Ali R Butt, Sudharshan S Vazhkudai, and Dimitrios S Nikolopoulos. On the use of gpus in realizing cost-effective distributed raid. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 469–478, 2012.

[40] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing array of ssds when the storage device is no longer the performance bottleneck. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, pages 20–20, 2017.

[41] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (ATC)*, pages 799–812, 2019.

[42] Youngjae Kim, Sarp Oral, Galen M Shipman, Junghee Lee, David A Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2011.

[43] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/softwareco-programmable framework for computational ssds to accelerate deep learning service on large-scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST)*, pages 147–164, 2022.

[44] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/softwareco-programmable framework for computational ssds to accelerate deep learning service on large-scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST)*, pages 147–164, 2022.

[45] Miryeong Kwon, Jie Zhang, Gyuyoung Park, Wonil Choi, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 87–96, 2017.

[46] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, et al. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST)*, pages 339–353, 2016.

[47] 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)*, pages 83–90, 2018.

[48] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. Ioda: 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.

[49] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST)*, pages 331–346, 2023.

[50] Runhui Li, Yuchong Hu, and Patrick PC Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 148–159, 2015.

[51] Yongkun Li, Helen HW Chan, Patrick PC Lee, and Yinlong Xu. Elastic parity logging for ssd raid arrays. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2016.

[52] Shengwen Liang, Ying Wang, Cheng Liu, Huawei Li, and Xiaowei Li. Ins-dla: An in-ssd deep learning accelerator for near-data processing. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 173–179, 2019.

[53] 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 (ATC)*, pages 395–410, 2019.

[54] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[55] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, 2007.

[56] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin,

Yang Liu, and Steven Swanson. Willow: A user-programmablessd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014.

[57] Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu, and Yong Chen. Ssdup: a traffic-aware ssd burst buffer for hpc systems. In *Proceedings of the international conference on supercomputing*, pages 1–10, 2017.

[58] 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.

[59] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated raid storage in modern datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 147–163, 2023.

[60] SPDK. Blobstore filesystem. https://spdk.io/doc/blobfs.html.

[61] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. FIU traces (SNIA IOTTA trace set 390). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, 2009.

[62] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 559–571, 2020.

[63] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. Straid: Stripe-threaded architecture for parity-based raids with ultra-fast ssds. In *2022 USENIX Annual Technical Conference (ATC)*, pages 915–932, 2022.

[64] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[65] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[66] Greg Wong. Ssd market overview. In *Inside Solid State Drives (SSDs)*, pages 1–17. Springer, 2013.

[67] 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.

[68] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. Ldm: Log disk mirroring with improved performance and reliability for ssd-based disk arrays. *ACM Transactions on Storage (TOS)*, 12(4):1–21, 2016.

[69] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[70] 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, 2017.

[71] Shushu Yi, Yanning Yang, Yunxiao Tang, Zixuan Zhou, Junzhe Li, Chen Yue, Myoungsoo Jung, and Jie Zhang. Scalaraid: optimizing linux software raid system for next-generation storage. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 119–125, 2022.

[72] Chi Zhang, Yi Wang, Tianzheng Wang, Renhai Chen, Duo Liu, and Zili Shao. Deterministic crash recovery for nand flash based storage systems. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[73] 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)*, pages 121–136, 2020.