FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs

Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, and Changlim Lee, *Yonsei University;* Mohammad Alian, *UIUC;* Myoungjun Chun, *Seoul National University;* Mahmut Taylan Kandemir, *Penn State University;* Nam Sung Kim, *UIUC;* Jihong Kim, *Seoul National University;* Myoungsoo Jung, *Yonsei University*

https://www.usenix.org/conference/osdi18/presentation/zhang

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

FLASHSHARE: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs

Jie Zhang[1], Miryeong Kwon[1], Donghyun Gouk[1], Sungjoon Koh[1], Changlim Lee[1],
Mohammad Alian[2] , Myoungjun Chun[3], Mahmut Taylan Kandemir[4],
Nam Sung Kim[2], Jihong Kim[3], and Myoungsoo Jung[1]

Yonsei University[1],
Computer Architecture and Memory Systems Laboratory,
University of Illinois Urbana-Champaign[2], Seoul National University[3], Pennsylvania State University[4]
http://camelab.org

Abstract

A modern datacenter server aims to achieve high energy efficiency by co-running multiple applications. Some of such applications (e.g., web search) are latency sensitive. Therefore, they require low-latency I/O services to fast respond to requests from clients. However, we observe that simply replacing the storage devices of servers with Ultra-Low-Latency (ULL) SSDs does not notably reduce the latency of I/O services, especially when co-running multiple applications. In this paper, we propose FLASHSHARE to assist ULL SSDs to satisfy different levels of I/O service latency requirements for different co-running applications. Specifically, FLASHSHARE is a holistic cross-stack approach, which can significantly reduce I/O interferences among co-running applications at a server without any change in applications. At the kernel-level, we extend the data structures of the storage stack to pass attributes of (co-running) applications through all the layers of the underlying storage stack spanning from the OS kernel to the SSD firmware. For given attributes, the block layer and NVMe driver of FLASHSHARE differently manage the I/O scheduler and interrupt handler of NVMe. We also enhance the NVMe controller and cache layer at the SSD firmware-level, by dynamically partitioning DRAM in the ULL SSD and adjusting its caching strategies to meet diverse user requirements. The evaluation results demonstrate that FLASHSHARE can shorten the average and 99^{th}-percentile turnaround response times of co-running applications by 22% and 31%, respectively.

1 Introduction

Datacenter servers often run a wide range of online applications such as web search, mail, and image services [8]. As such applications are often required to satisfy a given Service Level Agreement (SLA), the servers should process requests received from clients and send the responses back to the clients within a certain amount of time. This requirement makes the online applications latency-sensitive, and the servers are typically (over)provisioned to meet the SLA even when they unexpectedly receive many requests in a short time period. However, since such events are infrequent, the average utilization of the servers is low, resulting in low energy efficiency with poor energy proportionality of contemporary servers [28, 17].

To improve utilization and thus energy efficiency, a server may run an online application with offline applications (e.g., data analytics workloads), which are latency-insensitive and are often throughput-oriented [26, 30, 29]. In such cases, it becomes challenging for the server to satisfy a given SLA for the online application because co-running these applications further increase I/O service latency. We observe that device-level I/O service latency of a high-performance NVMe solid state drive (SSD) contributes to more than 19% of the total response time of online applications, on average. To reduce the negative impact of long I/O service latency on response time of online applications, we may deploy Ultra-Low-Latency (ULL) SSDs based on emerging memory, such as Z-NAND [36] or 3D-Xpoint [15]. These new types of SSDs can accelerate I/O services with ULL capability. Our evaluation shows that ULL SSDs (based on Z-NAND) can give up to 10× shorter I/O latency than the NVMe SSD [14] (cf. Section 2).

These ULL SSDs offer memory-like performance, but our in-depth analysis reveals that online applications cannot take full advantage of ULL SSDs particularly when a server co-runs two or more applications for higher utilization of servers. For example, the 99^{th} percentile re-

sponse time of *Apache* (i.e., online application) is 0.8 *ms*. However, the response time increases by 228.5% if the server executes it along with a *PageRank* (i.e., offline application). A reason behind this offset of the benefits of memory-like performance is that server's storage stack lacks understanding of criticality of user's I/O requests and its impact on the response time or throughput of a given application.

In this paper, we propose FLASHSHARE, a holistic cross-stack approach that enables a ULL SSD device to directly deliver its low-latency benefits to users and satisfy different service-level requirements. Specifically, FLASHSHARE fully optimizes I/O services from their submission to execution to completion, by punching through the current server storage stack. To enable this, FLASHSHARE extends OS kernel data structures, thereby allowing users to dynamically configure their workload attributes (for each application) without any modification to their existing codes. FLASHSHARE passes these attributes through all components spanning from kernel to firmware and significantly reduces inter-application I/O interferences at servers when co-running multiple applications. The specific stack optimizations that this work performs can be summarized as follows:

• *Kernel-level enhancement.* At the kernel-level, there are two technical challenges in exposing the pure performance of ULL SSDs to users. First, the Linux multi-queue block layer (blk-mq) holds I/O requests in its software/hardware queues, introducing long latencies. Second, the current standard protocol of the NVMe queuing mechanism has no policy on I/O prioritization, and therefore, a request from an offline application can easily block an urgent I/O service requested by an online application. FLASHSHARE carefully bypasses the latency-critical requests to the underlying NVMe queue. In addition, our NVMe driver pairs NVMe submission and completion queues by being aware of the latency criticality (per application).

• *Firmware-level design.* Even though kernel-level optimizations guarantee to issue latency-critical requests with the highest order, the ULL characteristics (memory-like performance) cannot be fully exposed to users if the underlying firmware has no knowledge of latency criticality. In this work, we redesign the firmware for I/O scheduling and caching to directly disclose ULL characteristics to users. We partition ULL SSD's embedded cache and separately allocate the cache for each I/O service based on its workload attributes. Our firmware dynamically updates the partition sizes and adjusts the prefetch I/O granularity in a fine-granular manner.

• *New interrupt services for ULL SSDs.* We observe that the current NVMe interrupt mechanism is not optimized for ULL I/O services, due to the long latency incurred by storage stack layers. We also discover that a polling method (implemented in Linux 4.9.30) consumes many CPU cycles to check the completion of I/O services, which may not be a feasible option for servers co-running two or more applications. FLASHSHARE employs a selective interrupt service routine (Select-ISR), which uses message-signaled interrupts for only offline applications, while polling the I/O completion for online interactive applications. We further optimize the NVMe completion routine by offloading the NVMe queue and ISR management into a hardware accelerator.

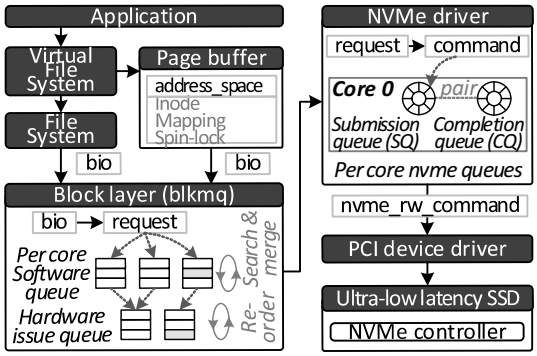


Figure 1: High-level view of software kernel stack.

We implement the kernel enhancement components in a real I/O stack of Linux, while incorporating Select-ISR and hardware/firmware modifications using a full system simulation framework [2, 21]. We also revise the memory controller and I/O bridge model of the framework, and validate the simulator with a real 800GB Z-SSD prototype. The evaluation results show that FLASHSHARE can reduce the latency of I/O stack and the number of system context switch by 73% and 42%, respectively, while improving SSD internal cache hit rate by 37% in the co-located workload execution. These in turn shorten the average and 99^{th} percentile request turnaround response times of the servers co-running multiple applications (from an end-user viewpoint) by 22% and 31%, respectively.

2 Background

2.1 Storage Kernel Stack

Figure 1 illustrates the generic I/O stack in Linux, from user applications to low-level flash media. An I/O request is delivered to a file system driver through the virtual file system interface. To improve system-level performance, the request can be buffered in the page buffer module, using an `address_space` structure, which in-

cludes the `inode` information and mapping/spin-lock resources of the owner file object. When a cache miss occurs, the file system retrieves the actual block address, referred to as Logical Block Address (LBA) by looking up `inodes` and sends the request to the underlying multi-queue block layer (blk-mq) through a `bio` structure.

In contrast to the kernel's block layer that operates with a single queue and lock, the multi-queue block layer (blk-mq) splits the queue into multiple separate queues, which helps to eliminate most contentions on the single queue and corresponding spin-lock. blk-mq allocates a `request` structure (associated to `bio`) with a simple tag and puts it in the per-CPU software queues, which are mapped to the hardware issue queues. The software queue of blk-mq merges the incoming `request` with an already-inserted `request` structure that has the same LBA, or an adjacent LBA to the current LBA. The merge operation of blk-mq can reduce the total number of I/O operations, but unfortunately, it consumes many CPU cycles to search through the software queues. From the latency viewpoint, the I/O merging can be one of the performance bottlenecks in the entire storage stack. On the other hand, the hardware issue queues simply buffer/reorder multiple requests for the underlying NVMe driver. Note that the hardware issue queue can freely reorder the I/O requests without considering the I/O semantics, since the upper-level file system handles the consistency and coherence for all storage requests.

The NVMe driver exists underneath blk-mq, and it also supports a large number of queue entries and commands per NVMe queue. Typically, each deep NVMe queue is composed of pairing a *submission queue (SQ)* and a *completion queue (CQ)*. The NVMe driver informs the underlying SSD of the arrivals and completions of I/O requests through head and tail pointers, allocated per NVMe queue. In the storage stack, every request issued by the NVMe driver is delivered to the PCI/PCIe device driver in the form of a `nvme_rw_command` structure, while the SSD dispatches them in an active manner; in contrast to other storage protocols in which a host-side controller must dispatch or transfer all data and commands, NVMe SSDs can pull the command and data stored in system memory from storage side without a host intervention. When the I/O request is completed by the SSD, it sends a *message signaled interrupt (MSI)* that directly writes the interrupt vector of each core's programmable interrupt controller. The interrupted core executes an ISR associated with the vector's interrupt request (IRQ). Subsequently, the NVMe driver cleans up the corresponding entry of the target SQ/CQ and returns the completion results to its upper layers, such as blk-mq and filesystem.

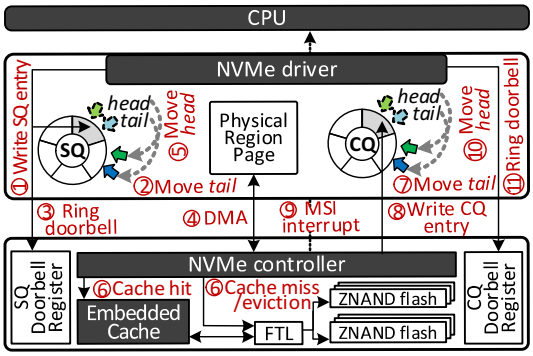


Figure 2: Overview of device firmware stack.

2.2 Device Firmware Stack

Based on the NVMe specification, the deep queues are created and initialized by the host's NVMe driver through the administrator queues, and the I/O requests in the queues are scheduled by the NVMe controller that exists on top of the NVMe SSD firmware stack [46]. Most high-performance SSDs, including all devices we tested in this study [13, 14, 36], employ a large internal DRAM (e.g., 1 GB $\sim$ 16 GB). Thus, underneath the NVMe controller, SSDs employ an embedded cache layer, which can immediately serve the I/O requests from the internal DRAM without issuing an actual storage-level operation when a cache hit occurs at the internal DRAM [42, 20]. If a cache miss or replacement is observed, the NVMe controller or cache layer generates a set of requests (associated with miss or replacement) and submits them to the underlying flash translation layer (FTL), which manages many Z-NAND chips across multiple channels [6, 18].

Figure 2 shows the components of the firmware stack and depicts how the NVMe controller pulls/pushes a request to/from the host. Specifically, when the NVMe driver receives a `request` (①), it increases the tail/head of SQ (②) and writes the doorbell register (③) that the NVMe controller manages. The NVMe controller then initiates to transfer the target data (④) associated with the tail from the host's kernel memory pointed by the corresponding Physical Region Page (PRP) (stored in `nvme_rw_command`). Once the DMA transfer is completed, the NVMe controller moves the head to the NVMe queue entry pointed by the tail (⑤), and forwards the request to either the embedded cache layer or underlying FTL (⑥). When a cache miss or replacement occurs, the FTL translates the target LBA to the corresponding physical page address of the underlying Z-NAND, and performs complex flash-related tasks (if needed), such as garbage collection and wear-leveling

Components	Spec.	Components	Spec.
CPU	i7-4790	Memory	32GB
	3.6GHz		DDR3
	8 cores	Chipset	H97

Table 1: Server configurations.

[38, 33, 5, 4, 12]. Unlike traditional NAND [11], Z-NAND completes a 4KB-sized read service within 3 μs [36] and we observed that a Z-NAND based ULL SSD can complete an I/O service within 47$\sim$52 μs, including data transfer and FTL execution latencies (cf. Figure 3a).

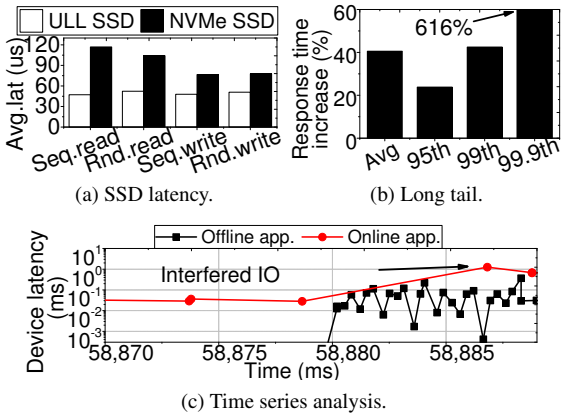
After completing the service of the I/O request, the NVMe controller increases the corresponding tail pointer of CQ (⑦). It then performs a DMA transfer to the host and changes the phase tag bit associated with the target CQ entry (⑧). The controller notifies the DMA completion by sending an MSI to the host (⑨). The host's ISR checks the phase tag by searching through the queue entries from the head to the tail. For the ones that have a valid phase tag, the ISR clears the tag bit and processes the rest of the I/O completion routines. Finally, it increases the head of CQ (⑩), removes the corresponding entry of SQ, and writes the CQ's head doorbell of the NVMe controller (⑪). While polling is not a standard method in the NVMe specification, the state-of-the-art Linux (4.9.30) can support it, since the NVMe controller directly changes the phase tags of the target entries over PCIe before sending the corresponding MSI. Thus, in the kernel storage stack, the NVMe driver checks the phase tags of CQ and simply ignores the MSI updates.

3 Cross-Layer Design

3.1 Challenges with Fast Storage

In this section, we characterize the device latency of a prototype of real 800GB Z-NAND based ULL SSD by comparing it against the latency of a high-performance NVMe SSD [14]. We then evaluate the performance of a server equipped with the ULL SSD, when *Apache* (an online latency-sensitive application) co-runs with *PageRank* (an offline throughput-oriented application). While *Apache* requires responding to the service coming from the client through TCP/IP by retrieving data on object storage, *PageRank* performs data analytics over Hadoop's MapReduce (24GB dataset). The configuration details of the server under test are listed in Table 1.

Figure 3a compares the average latency of the ULL SSD and NVMe SSD, with the number of queue entries varying from 1 to 64. The latencies of the ULL SSD for the random and sequential access patterns are 42% and 48% shorter than that of the NVMe SSD, respectively. However, as shown in Figure 3b, we observe that the turnaround response times of the online application significantly degrade when co-running it along with the offline application. Specifically, the response time of *Apache* becomes 41% longer if *PageRank* also runs on the same server. This performance deterioration is also observed in the long tail: the 95^{th} and 99^{th} response times of *Apache* under the co-running scenario increase by 24% and 43%, respectively, compared to those of an *Apache*-only execution scenario.



(a) SSD latency. (b) Long tail.

(c) Time series analysis.

Figure 3: Application co-running analysis.

The reason behind these response time increases is captured by Figure 3c. Once *PageRank* begins to perform I/O services (at 58,880 ms), the I/O services of *Apache* gets interfered by *PageRank*, and this increases the response time of *Apache* by 42$\times$ compared to the standalone execution situation (before 58,880 ms). This happens because the server storage stack has no knowledge of the ultra-low latency exposed by the underlying Z-NAND media, and also most of the components in the stack cannot differentiate *Apache*'s I/O services from *PageRank*'s I/O services (even though the two applications require different levels of the I/O responsiveness).

3.2 Responsiveness Awareness

It is very challenging for the kernel to speculate workload behaviors and predict the priority/urgency of I/O requests [24]. Since users have a better knowledge of I/O responsiveness, a more practical option is to offer a set of APIs to users. However, such APIs require significant changes to existing server application's sources. Instead, we modify the Linux process control block, called `task_struct`, to accommodate a *workload attribute* for each application. A potential issue in leveraging the attribute, stored in `task_struct`, from the software layers in the storage stack is that a reference of `task_struct` may not be valid, based on the loca-

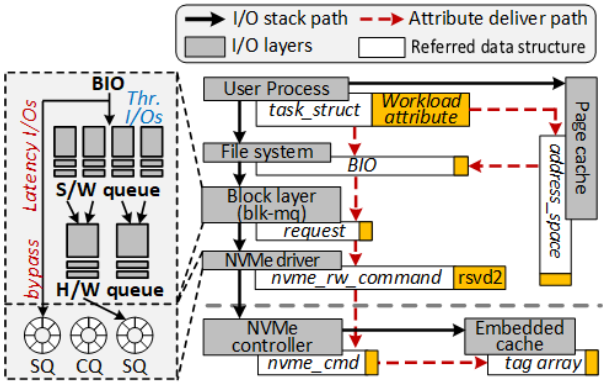


Figure 4: Overview of our server stack optimizations.

tion of the storage stack and the timing when a layer retrieves such `task_struct`. Therefore, it is necessary for blk-mq and NVMe driver to have their own copies of the workload attribute per I/O service. To this end, we further extend `address_space`, `bio`, `request`, and `nvme_rw_command` structures to punch through the storage stack and pass the workload attribute to the underlying SSD firmware.

As such, FLASHSHARE provides a utility, called chworkload_attr, which allows servers to configure and dynamically change the attribute of each application similar as the nice mechanism [9]. The chworkload_attr helps users to embed the criticality of responsiveness into each application's `task_struct`. We modify the system call table (e.g., `arch/x86/entry/syscalls/syscall_64.tbl`) and implement two system calls, to set/get the workload attribute to/from the target `task_struct`. These system invocations are registered at `/linux/syscall.h` with the `asmlinkage` tag. They change the attribute of a specific process (given by the user from a shell), which is implemented in `/sched/cores.c`. The chworkload_attr simply invokes the two system calls with an appropriate system call index, registered in the system table. Using such interfaces, the chworkload_attr can capture the attribute and fill the information in `task_struct` for each application at the kernel level. It should be noted that the chworkload_attr is designed for server-side users (e.g., datacenter operators), not for client-side users who may recklessly ask a higher priority all the time.

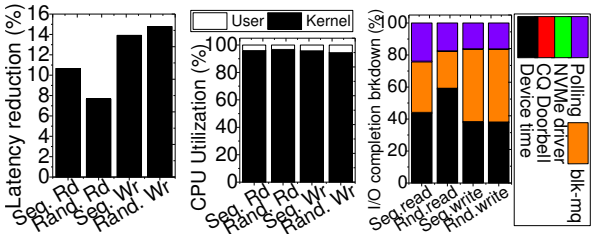
Figure 4 illustrates how the workload attribute is referred by `task_struct` in the storage stack modified by FLASHSHARE. If an incoming I/O request uses a direct I/O (`O_DIRECT`), the file system driver (EXT4 used in our implementation) retrieves the attribute and puts it into `bio`. Otherwise, the page cache copies the attribute from `task_struct` to `address_space`. Therefore, when `blk-mq` receives a `bio`, it includes the criticality of responsiveness in the attribute, and copies that information to a `request` structure. Lastly, the NVMe driver overrides the attribute to an unused field, called `rsvd2` of `nvme_rw_command`. The underlying SSD's firmware can catch the host-side attribute information per request by reading out the value in `rsvd2` and passing it to the NVMe controller and embedded cache layer, the details of which will be explained in Section 4.

3.3 Kernel Layer Enhancement

By utilizing the workload attributes, we mainly optimize the two layers underneath the file system: blk-mq and NVMe driver, as shown in Figure 4. The software and hardware queues of blk-mq hold I/O requests with the goal of merging or reordering them. Even though a deep blk-mq queue can increase chances for merging and reordering requests thereby higher bandwidth utilization, it also introduces long queue waiting delays. This can, unfortunately, hurt the responsiveness of online applications (and cannot take the advantage of ULL). To address this potential shortcoming, we enhance blk-mq to bypass all the I/O services requested from the online application to the NVMe driver (without queueing), while tracking other requests coming from the throughput applications just like normal software and hardware queues. However, this simple bypass strategy potentially raises an I/O hazard issue; a hazard could happen if an offline application has an I/O request being scheduled by blk-mq to the same LBA that a subsequent online application issued.

Because such request cannot be skipped in the queue, blk-mq retrieves it, which may have the potential hazard, from the software queue. If the operation type of the retrieved request is different from that of the incoming request that we want to bypass, blk-mq submits the retrieved request along with the incoming request in tandem. Otherwise, blk-mq merges those two requests into a single `request` structure and forwards the merged request to the underlying NVMe driver.

Under blk-mq, the NVMe driver submits the bypassed request to the corresponding SQ. One of the issues in the NVMe queue management is that the head and tail pointers for a pair of the target CQ/SQ are managed by the (kernel-side) NVMe driver and the (firmware-side) NVMe controller together in a round-robin fashion. Thus, even though our modification in blk-mq prioritizes latency-critical I/O services by expecting them to be scheduled in the SQ earlier than other requests, the NVMe controller can dispatch a service requested by an offline application prior to the latency-critical I/O



(a) Average I/O latency. (b) CPU utilization. (c) I/O completion breakdown.

Figure 5: I/O execution analysis of ULL SSD.

service. This, in turn, makes the service latency of the latter considerably longer. To address this undesirable situation, we create two SQs and one CQ per core as a pair of NVMe queue, which is different from a traditional NVMe queue management strategy. Specifically, as shown in Figure 4, an SQ between two SQs is used for the requests whose attributes come from online applications. In our implementation, the NVMe driver sends a message via the administrator queue to inform the NVMe controller of selecting a new queue arbitration method that always gives a high priority to scheduling requests in such the SQ. To avoid a starvation owing to the priority SQ, the NVMe driver drains the I/O requests originating from the offline applications if the number of such queued requests is greater than a threshold, or if they are not served within a certain amount of time. We observed that it is best to start draining the queue with a 200 μs threshold or when there are 8 pending queue entries.

4 I/O Completion and Caching

Figure 5a shows the actual latency improvement when we use the Linux 4.9.30 polling mechanism for a ULL SSD (Z-SSD prototype). In this evaluation, we set the size of all the requests to 4KB. As shown in Figure 5a, the I/O latency with the polling mechanism is 12% shorter than the one managed by MSI for all I/O request patterns. However, we also observe that the cores in the kernel mode are always busy in handling I/O completions. Specifically, Figure 5b shows the CPU utilization of the polling mechanism for both the kernel and user modes. This figure shows that the CPU utilization for polling gets significantly high (almost 97% of CPU cycles are used for only polling the I/O completion). This high CPU utilization presents two technical issues. First, as there is no core to allocate in handling the criticality of I/O responsiveness, the original polling method is not a feasible option for a server co-running multiple applications. Second, while a 12% latency improvement of the polling method is still promising, we could shorten

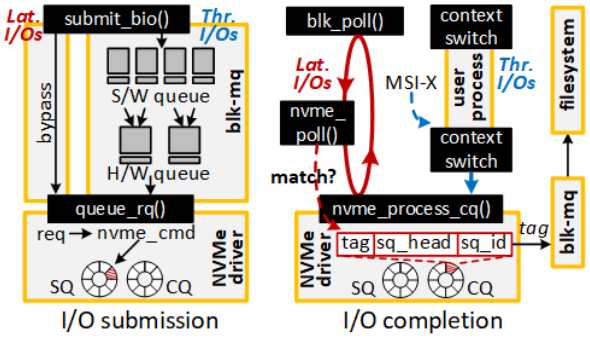


Figure 6: The process implemented by the selective interrupt service routine.

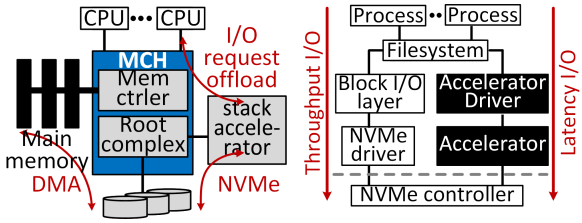
the latency even more, if we could alleviate the core-side overheads brought by polling for the I/O completion.

4.1 Selective Interrupt Service Routine

FLASHSHARE uses polling for only I/O services originating from online applications, while MSI is still used for offline applications. Figure 6 shows how this selective ISR (Select-ISR) is implemented. We change `submit_bio()` of blk-mq to insert an incoming request (i.e., `bio`), delivered by the file system or page cache, into its software queue if the attribute of `bio` indicates an offline application. This request will be reordered and served just like a normal I/O operation. In contrast, if the incoming request is associated with an online application, blk-mq directly issues it to the underlying NVMe driver by invoking `queue_rq()`. The NVMe driver then converts the I/O request into NVMe commands and enqueues it into the corresponding SQ.

With Select-ISR, the CPU core can be released from the NVMe driver through a context switch (CS), if the request came from offline applications. Otherwise, blk-mq invokes to the polling mechanism, `blk_poll()`, after recording the tag of the I/O service along with online applications. `blk_poll()` continues to invoke `nvme_poll()`, which checks whether a valid completion entry exists in the target NVMe CQ. If it is, blk-mq disables IRQ of such CQ so that MSI cannot hook the procedures of blk-mq later again. `nvme_poll()` then looks up the CQ for a new entry by checking the CQ's phase tags. Specifically, `nvme_poll()` searches an CQ entry whose request information is matched with the tag that `blk_poll()` waits for completion. Once it detects such a new entry, blk-mq exits from the infinite iteration implemented in `blk_poll()` and switches the context to its user process.

A challenge in enhancing the storage stack so that it can be aware of ULL is that, even though we propose



(a) Hardware viewpoint. (b) Software viewpoint.

Figure 7: Overview of our I/O stack accelerator.

Select-ISR, polling still wastes many CPU cycles and blk-mq consumes kernel CPU cycles to perform simple operations, such as searching the tag in SQ/CQ and merging the requests for each I/O service invocation. This is not a big issue with conventional SSDs, but with a ULL SSD, it can prevent one from enjoying the full benefits of low latency. For example, in Figure 5c, we observed that polling and storage stack modules, including ISR, context switching, and blk-mq cycles, take 58% of total I/O completion time. Thus, as a further enhancement of Select-ISR, we propose an I/O-stack accelerator. Figure 7 shows how our I/O-stack accelerator is organized from the hardware and software viewpoints. This additional enhancement migrates the management of the software and hardware queues from blk-mq to an accelerator attached to a PCIe. This allows a `bio` generated by the upper file system to be directly converted into a `nvm_rw_command`. Especially, the accelerator searches a queue entry with a specific tag index and merges `bio` requests on behalf of a CPU core. The offload of such tag search and merge operations can reduce the latency overhead incurred by the software layers in the storage stack by up to 36% of the total I/O completion time. The specifics of this accelerator are described in Section 4.3.

4.2 Firmware-Level Enhancement

In our implementation, the NVMe controller is aware of the two SQs per core, and gives a higher priority to the I/O service enqueued in the latency-critical SQ. While this I/O scheduling issue can be simply updated, a modification of the embedded cache layer to expose a shorter latency to online applications can be challenging. Specifically, the cache layer can starve latency-critical I/O services if it serves more throughput-oriented I/O services. This situation can be observed even when the cache layer understands the workload attribute brought by the NVMe driver/controller, as the internal DRAM is a shared resource in a given SSD. In addition, the I/O patterns and locality of online applications are typically different from those of offline applications. That is, a single generic

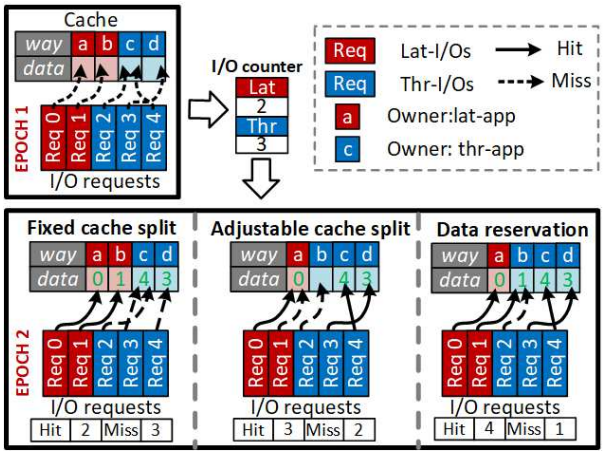


Figure 8: Example of adjustable cache partition.

cache access policy cannot efficiently manage I/O requests from both online and offline applications.

The cache layer of FLASHSHARE partitions the DRAM into two DRAM caches with the same number of sets, but different ways of associativity (i.e., cache ways) [45], and allocates each to online and offline applications, separately. If the size of partitioned caches is fixed, it can introduce cache thrashing depending on the I/O behavior of the given applications (and the corresponding workload patterns). For example, in Figure 8, if two partitioned caches (one for online applications and another for offline applications) employ two ways for each, the requests 2 $\sim$ 4 compete for the way 'c', and they experience cache misses.

To address this, in cases of high I/O demands, our cache layer collects the number of I/O accesses for the online and offline applications at each epoch. The proposed cache layer dynamically adjusts the number of cache ways allocated to two different cache partitions. As shown in Figure 8, if the cache splits can be adjusted (cf. adjustable cache split), the ways 'b'$\sim$'d' can accommodate the requests 2 $\sim$ 4. However, as the way 'b' is reallocated to the I/O requests of offline applications (e.g., throughput-oriented I/Os), the latency critical request 1 is unable to access data residing in the way 'b', introducing cache miss. To address this challenge, when adjusting the cache ways, the cache layer keeps the data for the previous owner as "read-only" until a new request is written into the corresponding way.

Most firmware in SSDs read out the data from multiple memory media to improve parallelism, and therefore, the cache can be polluted by an ill-tuned prefetching technique. As shown in Figure 9, we leverage a "ghost caching" mechanism [37, 31, 34] to help the SSD controller to evaluate the performance (i.e., cache

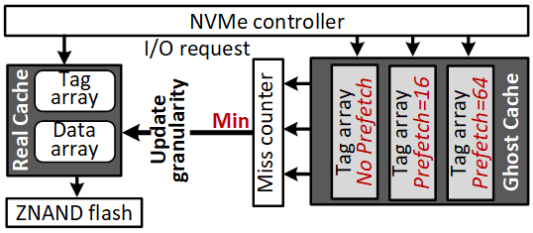


Figure 9: Adjustable cache prefetching scheme.

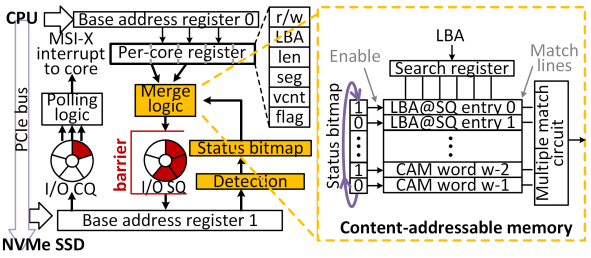


Figure 10: Design details of hardware accelerator.

miss rate) of various prefetching configurations and adjust the cache layer to the optimal prefetching configuration at runtime. Specifically, we build multiple ghost caches, each maintaining only cache tags without any actual data. The associativity and size of ghost caches are configured the same way as the configuration of the proposed cache layer, but each of these caches employs a different prefetching I/O granularity. In each epoch, FLASHSHARE identifies the ghost cache that exhibits a minimum number of cache misses, and changes the prefetch granularity of the cache layer to that of the selected ghost cache.

4.3 I/O-Stack Acceleration

We load the kernel driver of the proposed accelerator as an upper layer module of blk-mq (cf. Figure 7b). As shown in Figure 10, the accelerator driver checks if the incoming `bio` is associated with online latency-critical applications. If so, the driver extracts the operation type, LBA, I/O size, memory segment pointer (related to target data contents), and the number of memory segments from `bio`' `bio->bi_opf`, `bio->bi_iter.bi_sector`, `bio->bi_iter.bi_size`, `bio->bi_io_vec` and `bio->bi_vcnt`, respectively. The kernel driver then writes this extracted information into the corresponding registers in base address registers (BAR) of the accelerator. The accelerator then identifies an I/O submission queue entry that has an LBA, which is the same as the target LBA of incoming `bio` request. If the accelerator finds such an entry, its merge logic automatically merges the information (stored in BAR) into the target entry; otherwise, the accelerator composes a new NVMe command and appends it to the tail of the target SQ. Then, the accelerator rings (writes) the doorbell register of the underlying ULL SSD. However, as the merge logic and ULL SSD can simultaneously access the I/O SQ entries, an I/O hazard may occur. To prevent such situations, we propose to add a *barrier logic*, which is a simple MUX and works as a hardware arbitrator. It allows either the merge logic or ULL SSD (via BAR1 register) to access the target NVMe SQ at one time. Once the I/O request is inserted into the SQ, the polling logic of our accelerator starts to poll the corresponding CQ. When the I/O service is completed, the accelerator raises an interrupt signal for the accelerator driver. The driver then takes the control of I/O completion.

Note that, since searching through all entries can introduce a long latency, the accelerator employs content-addressable memory (CAM), which keeps the LBA of received `nvme_cmd` instances (for I/O submissions). Using the content-addressable memory, our accelerator in parallel compares the incoming LBA with all recorded LBAs, thereby significantly reducing the search latency. For the simultaneous comparison, the number of CAM entries is set to be same as the number of SQ entries. In addition, if there is no issued `nvme_cmd` or `nvme_cmd` instance(s) is fetched by the underlying ULL SSD, the corresponding SQ entries should be invalid. Thus, we introduce a *status bitmap* to filter the entries, which do not contain valid `nvme_cmd` instances. Specifically, our status bitmap is set to 1, if merge logic inserts a new NVMe command; the status bitmap is reset to 0, if the ULL SSD is detected to pull NVMe commands from I/O SQ. If the status bitmap indicates that the request entries in CAM (associated with the target SQ) are invalid, CAM will skip searching those entries.

5 Evaluation

5.1 Experimental Setup

Implementation environment. We simulate our holistic optimization approaches on an event-driven computer-system architecture simulator, gem5 [2]. Specifically, we configure it to a full system mode which runs on 64-bit ARM ISA. We use Linux 4.9.30 as the default kernel in this simulation, and set up 8 ARM cores with 2GHz, which have private 64KB L1 data and 64KB L1 instruction caches. We also configure a 2GB DRAM-based main memory, which is shared by all 8 cores. Note that, as we employ a detailed architectural simulator (simulation is up to 1000x slower than native-execution), we scale the simulation memory size to reduce the warmup

gem5		SimpleSSD	
parameters	**value**	**parameters**	**values**
core	64-bit ARM, 8, 2GHz	**read/write/erase**	3us/100us/1ms
L1D$/L1I$	64KB, 64KB	**channel/package**	16/1
mem ctrler	1	**die/plane**	8/8
memory	DDR3, 2GB	**page size**	2KB
Kernel	4.9.30	**DMA/PCIe**	800MHz,3.0, x4
Image	Ubuntu 14.04	**DRAM cache**	1.5GB

Table 2: The configurations of gem5 and SimpleSSD.

App	Read Ratio	I/O size	I/Os per MegaInst.	Seq Ratio
bfs	0.997	238KB	0.025	0.70
gpgnu	1.000	134KB	1.985	0.78
gp	0.995	23KB	0.417	0.81
gzip	0.989	150KB	0.105	0.66
index	0.998	28KB	0.205	0.79
kmn	1.000	122KB	0.037	0.82
PR	0.995	30KB	0.367	0.71
ungzip	0.096	580KB	0.076	0.79
wcgnu	1.000	268KB	0.170	0.75
wc	0.999	25KB	0.548	0.89
ap	0.999	24KB	0.666	0.11
au	0.476	27KB	1.205	0.13
is	0.990	12KB	5.761	0.86

Table 3: The characteristics of workloads.

time of the CPU caches and DRAM. This is a common practice in architectural studies [1]. Our simulation environment integrates an accurate SSD simulator, SimpleSSD [21], which is attached to the computer system as a PCIe device. When booting, Linux running on gem5 recognizes SimpleSSD as a storage by creating a pair of NVMe SQ and NVMe CQ for each ARM core via the NVMe protocol [10]. Our SimpleSSD simulator is highly flexible and can configure various SSD prototypes. In this experiment, we configure SimpleSSD simulator as an ultra low latency SSD, similar to 800GB ZSSD [36]. The important characteristics of our gem5 simulation and SimpleSSD simulation setups are shown in Table 2.

Configurations. We implement four different computer systems by adding the optimization techniques proposed in FLASHSHARE, and compare them against Vanilla.

1. **Vanilla**: a vanilla Linux-based computer system running on ZSSD.
2. **CacheOpt**: compared to `Vanilla`, we optimize the cache layer of the SSD firmware by being aware of responsiveness.
3. **KernelOpt**: compared to `CacheOpt`, we further optimize the block I/O layer to enable latency-critical I/Os to bypass the software and hardware queues. In addition, this version also supports reordering between the NVMe driver and the NVMe controller.
4. **SelectISR**: compared to `KernelOpt`, we add the optimization of selective ISR (cf. Section 4).
5. **XLER**: based on `SelectISR`, we improve the I/O stack latency by employing our hardware accelerator.

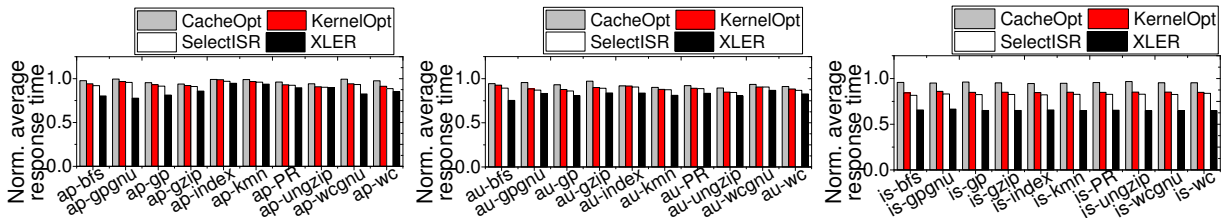
Workloads. We evaluate three representative online interactive workloads (latapp): *Apache (ap)*, *Apache-update (au)*, and *ImageServer (is)*. All these workloads create a web service scenario, in which a client thread is created to send client requests periodically and a server thread is created to receive the client requests. Once requests arrive in the server side, the server thread creates multiple worker threads to process these requests and respond to the client after the completion. For `Apache` and `Apache-update`, the requests are "SELECT" and "UPDATE" commands targeting a database, while the requests of `ImageServer` are image access requests. Note that the response time we measure in our experiments is the time between when the client thread issues a request and when the response of the request is received by the client. To satisfy the SLA requirements of the online applications, we select the request issue rate (as 400) right at the knee of the latency-load curve, which is also suggested by [28]. We also collect ten different offline workloads (thrapp) from BigDataBench (a Big Data benchmark Suite) [43] and GNU applications. The salient characteristics of our online and offline applications are listed in Table 3. In our evaluations, we co-run online interactive workloads and offline workloads together to simulate a real-world server environment.

In our evaluations, the response time means "end-to-end latency", collected from interacting workloads between client and server, which is different with other storage performance metrics that we used (i.e., I/O latency). Specifically, while the storage performance metrics only consider the characteristics of storage subsystems, the response time in our evaluations includes the request generate/send/receive latencies in a client, network latency, request receive/process/response latencies in a server, and storage-system latency.

5.2 Performance Analysis

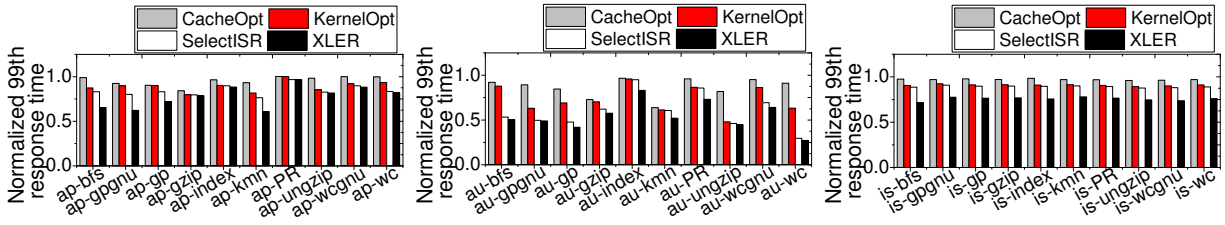
Figures 11 and 12 plot the average response time and the 99th response time, respectively, with the five different system configurations, normalized to those of `Vanilla`. Overall, `CacheOpt`, `KernelOpt`, `SelectISR` and `XLER` reduce the average response time by 5%, 11%, 12% and 22%, respectively, compared to `Vanilla`, while achieving 7%, 16%, 22% and 31% shorter 99th response times than `Vanilla` in that order.

`Vanilla` has the longest response time across all system configurations tested, because, it is not aware of the different workload attributes, and in turn loses the opportunity to optimize the kernel stack and flash firmware for latency-critical I/Os. In contrast, `CacheOpt` catches the



(a) `Apache (ap).` (b) `Apache-update (au).` (c) `ImageServer (is).`

Figure 11: Average response times of our online interactive applications normalized to `Vanilla`.



(a) `Apache (ap).` (b) `Apache-update (au).` (c) `ImageServer (is).`

Figure 12: 99th response times of our online interactive applications normalized to `Vanilla`.

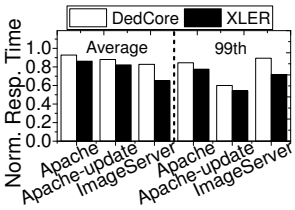


Figure 13: Response time analysis of using different polling techniques (normalized to `Vanilla`).

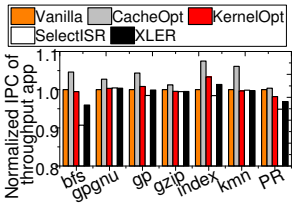
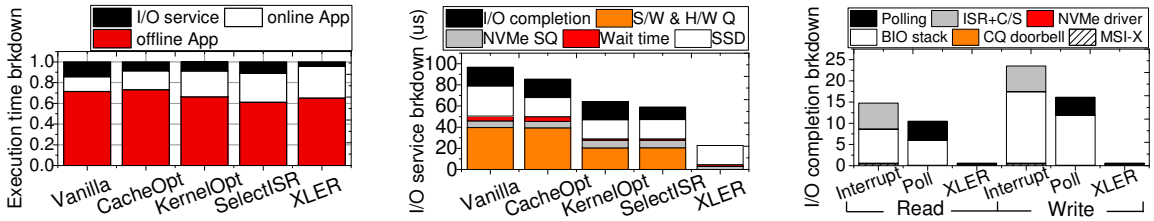


Figure 14: Performance analysis of offline applications (normalized to `Vanilla`).

workload attributes from the user processes and passes them to the underlying flash firmware. It further optimizes the SSD embedded cache by isolating latency-critical I/Os from the interference coming from throughput I/Os and customizing the cache access policy for latency-critical I/Os. As a result, it can accommodate more latency-critical I/Os in the SSD-embedded cache. As shown in Figures 11b and 12b, `CacheOpt` can reduce the average response time and 99th response time by up to 11% and 27%, respectively, if there are intensive write I/Os. Nonetheless, as flash firmware sits at the bottom of overall I/O stack, `CacheOpt` cannot effectively prevent throughput I/Os from impeding latency-critical I/Os from upper Linux kernel layers. Compared to `CacheOpt`, `KernelOpt` detects latency-critical I/O when it is inserted into the block I/O layer and creates a short path to send latency-critical I/O directly to the ULL SSD. Specifically, it enables latency-critical I/Os to directly bypass the software and hardware queues in the block I/O layer. It also collaborates with NVMe driver and NVMe controller to allocate an NVMe submission queue dedicated to latency-critical I/Os and fetch the latency-critical I/Os with a higher priority. Note that, `KernelOpt` can significantly reduce the 99th response time when offline applications generate intensive I/O requests (e.g., *ungzip*). The optimizations mentioned above can further reduce the average response time and the 99th response time by 6% and 8%, respectively, compared to `CacheOpt`. While `KernelOpt` works well for I/O submission optimization, it fails to handle the software overhead introduced by the interrupt-based I/O completion approach. For example, while an SSD read is as short as 3us, the ISR of the MSI-X interrupt and context switch collectively consume more than 6us. `SelectISR` selectively polls the latency-critical I/Os to avoid the use of ISR and context switch. Compared to the relatively long response time (i.e., more than 3 ms), the time saved from the ISR and context switch does not significantly reduce the average response time. However, in Figure 12b, we can observe that `SelectISR` can reduce the 99th request time by up to 34% in workload *bfs*. This is because, this compute-intensive workload creates multiple working threads that use up CPU resources and postpone the scheduling of the latency-critical I/Os. `SelectISR` secures CPU resources for the online interactive workloads as the CPU resources are not yielded to other user processes during polling. `XLER` can further reduce the average response time and 99th response time by 8% and 10%, respectively, compared to `SelectISR`. This is because, `XLER` simplifies the multiple queue management of the block I/O layer and NVMe driver, and accelerates the execution by employing customized circuits.

Since it would be possible to shorten the latency of storage stack by allocating a dedicated CPU core, we



(a) The execution time breakdown. (b) I/O stack breakdown. (c) I/O completion breakdown.

Figure 15: Execution time analysis of co-running `Apache-Index`.

also compare this alternative option with our hardware-assisted approach. Figure 13 shows the average response time and the 99th response time of online applications with a dedicated CPU core (`DedCore`) and a hardware accelerator (`XLER`). `XLER` reduces the average response time and the 99th response time by 12% and 15%, compared to `DedCore`, respectively. This is because, to leverage a dedicated core to poll NVMe CQs, `DedCore` requires intensive communications with the general cores, which are in process of the actual online applications. Unfortunately, such communications introduce different types of overheads associated with CPU cache flushes and spin-lock management.

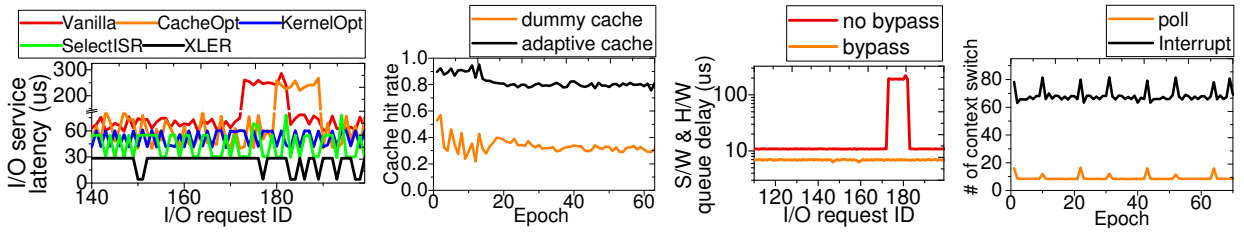
Although all our proposed techniques are oriented towards reducing the latency of the online interactive applications, offline workloads actually do not suffer from severe performance degradation. Figure 14 shows the performance of all evaluated offline workloads. Overall, FLASHSHARE does not degrade the performance of the offline applications, compared to `Vanilla` (the worst degradation observed is around 4%). This is because, the offline applications are not sensitive to the latency of each individual I/O, but instead rely on the storage bandwidth. Specifically, `CacheOpt` improves the performance of the offline applications by 3.6%, compared to `Vanilla`. This benefit comes mainly from the effective cache design and management. As `CacheOpt` separates the cache spaces for latency-critical I/O and throughput I/O, the throughput I/Os can better enjoy the fruits of short cache access latency without any competition originating from the latency-critical I/Os. On the other hand, we tune the delay time threshold and maximal number of throughput I/Os in the NVMe submission queue to make sure that all the delayed throughput I/Os are flushed by the NVMe controller before they start to introduce severe performance degradation. `SelectISR` degrades the performance of offline workloads by 2%, compared to `KernelOpt`. This is because, `SelectISR` uses up CPU resources for polling the latency-critical I/Os rather than executing the offline workloads. `XLER` achieves 1.2% higher performance than `SelectISR`, as it can effectively reduce the time spent for polling.

5.3 Effectiveness of Holistic Optimization

Figure 15a shows the execution time breakdown of co-running workloads *Apache* and *Index*. As shown in the figure, `CacheOpt` reduces the time needed to serve I/Os by 6%, compared to `Vanilla`, which in turn allows CPU to allocate more time for the offline application. On the other hand, `KernelOpt` postpones throughput I/Os, which blocks CPU from executing the offline application. For `SelectISR`, as CPU is used up for polling, less CPU time is allocated to the offline application. Finally, as `XLER` offloads the polling function to our hardware accelerator (cf. Section 4.3) and also reduces the time of I/O stack, both the online applications and offline applications can benefit from the reduced I/O service time.

Figure 15b plots the latency-critical I/O service breakdown between the co-running workloads, *Apache* and *Index*. `CacheOpt` reduces the average SSD access latency from 29 us to 18 us, compared to `Vanilla`, thanks to the short cache access latency. As the latency-critical I/Os are not queued in the software and hardware queues, the time for latency-critical I/O to pass through the block I/O layer is reduced from 39 us to 21 us when employing `KernelOpt`. Since the block I/O layer still needs to merge the latency-critical I/Os with the I/Os queued in software queues, the delay of the software and hardware queues cannot be fully eliminated. Compared to `KernelOpt`, `SelectISR` reduces the total I/O completion time by 5 us. We will present a deeper analysis of the I/O completion procedure shortly. As `XLER` removes the software and hardware queues in its implementation, it fully removes overheads of the block I/O layer.

Figure 15c plots the read and write I/O completion time breakdown. As shown in the figure, the polling function, interrupt service routine (ISR), context switch (CS) and block I/O layer collectively consume 96% of the I/O completion time, while the NVMe driver, sending MSI-X interrupt, and ringing CQ doorbell register together cost less than 0.5 us. Interestingly, although polling can remove the overhead caused by ISR and context switch, the polling function itself also introduces a 6 us delay. This delay is mainly caused by inquiring the



(a) Online I/O service. (b) Cache hit rates. (c) Blk-mq queue delay. (d) Context switch analysis.

Figure 16: Online interactive I/O execution time analysis.

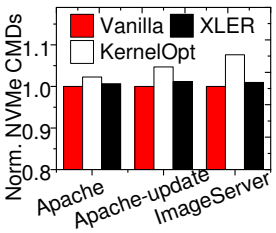


Figure 17: Analysis of # of NVMe commands.

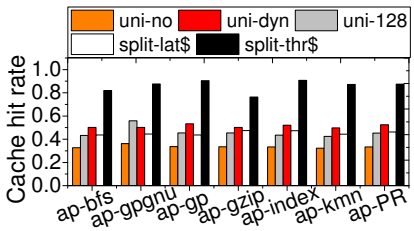


Figure 18: Various cache performance by co-running *Apache (ap)*.

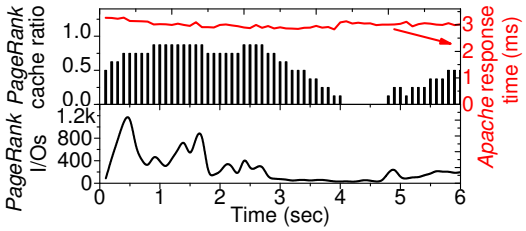


Figure 19: Analysis of performance dynamics when co-running *Apache* and *PageRank*.

IRQ locks of both BIO and NVMe CQ, setting current task status, and checking if there are any valid completion queue entries. In addition, although both the interrupt based approach and polling based approach execute the same block I/O completion function, polling reduces the average completion latency of block I/O by 4 us in both read I/Os and write I/Os. This is because, the interrupt-based approach context-switches CPU to other user processes which can pollute CPU caches, while the polling-based approach buffers the data used for I/O completion in the CPU cache.

5.4 I/O Stack Delay Analysis

Figure 16a shows the I/O service delay of the five different system configurations tested, for *Apache-Index* over time. In addition, Figure 16b plots the cache miss rates of a dummy cache (i.e., traditional SSD internal cache that has no knowledge of the host-side information) and our adaptive cache, while Figures 16c and 16d plot, respectively, the software and hardware queue latencies if we bypass or do not bypass latency-critical I/Os, and the number of context switches over time if we use the poll-based approach and the interrupt-based approach. `CacheOpt` exhibits a shorter I/O service delay than `Vanilla`, because it adjusts the cache space and prefetch granularity for the latency-critical I/Os and throughput I/Os, separately, resulting in fewer cache misses than the dummy cache in `Vanilla` (cf. Figure 16b). However, as shown in Figure 16a, `CacheOpt` cannot mitigate the long tail latency which is also observed in `Vanilla`, while `KernelOpt` successfully removes such tail latency. As shown in Figure 16c, the long tail latency comes from the software queue and hardware queue, if we buffer I/O requests in the software queue and hardware queue for merging and reordering. As `KernelOpt` enables the latency-critical I/Os to bypass the queues, it successfully eliminates the latency impact from the queues. `SelectISR` reduces the I/O service latency further, compared to `KernelOpt`. This is because, the polling-based approach can effectively reduce the number of interrupt service routine invocations and the number of context switches compared to the interrupt-based approach (cf. Figure 16d).

Since the I/O requests of online applications directly bypass the blk-mq layer under `KernelOpt`, the incoming I/O requests can lose their chances for merging, which can in turn increase the number of NVMe commands. Figure 17 shows the total number of NVMe commands, generated under different system configurations, namely `Vanilla`, `KernelOpt` and `XLER`. Compared to `Vanilla` that disables the bypassing scheme, `KernelOpt` increases the number of NVMe commands by only 2%. This is because the latency-critical I/O requests (coming from the online applications) exhibit low locality, and their arrival times are sporadic. Thanks to its merge logic, `XLER` further reduces the number of NVMe commands by 0.4%, on average, compared to `Vanilla`.

5.5 Sensitivity to Embedded Cache Design

Figure 18 gives the cache hit rate of various cache configurations when co-running *Apache* with offline workloads. Specifically, *uni-no*, *uni-128* and *uni-dyn* con-

figure a uniform cache for both the latency-critical and throughput I/Os. However, *uni-no* disables page prefetching and *uni-128* always prefetches 128 pages when cache misses, while *uni-dynfetch* employs the adaptive prefetching scheme we proposed (cf. Section 4.2). On the other hand, *split-lat$* and *split-thr$* represent the separate caches owned by the latency-critical I/Os and throughput I/Os, respectively. The separate cache employs adaptive prefetch scheme and adjusts cache space at runtime. *Apache* is a representative workload which randomly accesses the storage (cf. Table 3). As shown in the figure, although offline applications access the storage in a sequential manner, *uni-128* achieves only a 12% higher cache hit rate than *uni-no*. This is because, the random access pattern exhibited by *Apache* can pollute the cache space and make prefetching less effective. On the other hand, *uni-dyn* adjusts the prefetch granularity to small number of pages at runtime so that prefetching pages for latency-critical I/Os will not pollute all the caches. *split-lat$* does not achieve a high cache hit rate, due to the random access pattern of *Apache*. However, as we split the cache and isolate the interference from online applications, *split-thr$* achieves a great improvement in terms of hit rates.

Figure 19 further demonstrates the effectiveness of our dynamic cache partitioning scheme. Specifically, the lower half of Figure 19 shows the number of I/O requests, generated by offline applications during the execution, while the upper half shows the dynamics of the cache spaces, allocated to the offline application (in terms of ratio), and in parallel, demonstrates the response time of the online application. When there is an I/O burst imposed by *PageRank* (0∼2.8 seconds), our SSD controller isolates the negative impact of this I/O burst by preserving the cache spaces for *Apache* as 23% of the total cache space, on average. By partitioning the cache space being aware of the responsiveness for different applications, our cache partitioning secures just enough cache spaces for both *PageRank* and *Apache* such that the response time of *Apache* can be sustained at 3.1 ms while *PageRank* improves the cache hit rates by approximately 36% compared to a dummy cache. In cases where the offline application requests I/O services less than before (3∼6 seconds), our dynamic cache partitioning method increases the faction of internal cache spaces for the online application, which can be used for the application to perform pre-fetch or read-ahead in helping with an immediate response from the internal cache.

6 Related Work

In various computing domains, there are multiple studies to vertically-optimize storage stack [16, 24, 47, 50, 19, 22]. For example, [19] and [22] take flash firmware out of an SSD and locate it to the host, in order to remove the redundant address translation between a file system and FTL. In comparison, [47] proposes multiple partitioned caches on the host side. These caches understand multiple client characteristics by profiling the hints passed from one or more clients through out-of-bound protocol. However, the application hints are used only for cache management; such hints/approaches have no knowledge of the underlying storage stack and they do not consider the potential benefits of ULL. [16] optimizes mobile systems from the viewpoint of a file system and a block I/O device to improve the performance of databases such as SQLite. However, this optimization is applied only for the logging performance of mobile databases; it cannot be applied to other applications and cannot expose ULL to them. [50] schedules write requests by considering multiple layers on the kernel-side. While this can improve the write performance, such writes can block reads or ULL operations at the device level, as the ULL SSD also includes embedded DRAM caches and schedulers.

[24] observes that there exists an I/O dependency between background and foreground tasks. This dependency degrades overall system performance with a conventional storage stack since kernel always assigns a high priority to I/O services generated from the foreground tasks and postpones the background I/O requests. This I/O stack optimization allows the foreground tasks to donate their I/O priority to the background I/O services, when an I/O dependency is detected. However, this approach does not well fit with I/O workloads that often exhibit no I/O dependency. In particular, multiple applications executed on a same server (for a high resource utilization and energy efficiency) are already independent (as they operate in a different domain).

[32] and [23] propose sharing a hint with the underlying components to have a better data allocation in disk array or virtual machine domains. Similarly, [51] modifies a disk scheduler to priotize I/O requests, which are tagged by interactive applications. While most prior approaches leverage the hints from users/applications to improve the design of specific software layers, they do not consider the impact from the other parts of the storage stack.

[41] simplifies the handshaking processes of the NVMe protocol by removing doorbell registers and completion signals. Instead of using MSI, it employs a polling-like scheme for the target storage system. More

recently, [16, 7, 40, 3, 48] also observed that polling can consume significant CPU cycles to perform I/O completion. [7, 44] applies a hybrid polling method, which puts the core into sleep for a certain period of time, and just performs poll the request only when the sleep time has passed. While this strategy can reduce the CPU overheads to some extent, it is not trivial to determine the optimal time-out period for sleeps. Even in cases where system architects can decide a suitable time-out period, I/O request patterns can dynamically change and the determined time-out period may not be able to satisfy all user demands. Further, this can waste CPU cycles (if the time-out is short) or make the latency of I/O request longer (if for example the time-out is longer than the actual completion time). In addition, unlike our FLASHSHARE, the hybrid scheme cannot reduce the latency burden imposed by the software modules in the storage stack.

7 Discussion and Future Work

While the hardware accelerator of FLASHSHARE can perform a series of time-consuming tasks such as NVMe queue/entry handling and I/O merging operations on be half of CPUs, the accelerator employed in the target system is optional; we can drop the accelerator in favor of a software-only approach. This software-only FLASHSHARE (as a less-invasive option) makes performance of the server-side storage system approximately 10% worse and consumes 57% more storage-stack side CPU-cycles than hardware-assisted FLASHSHARE. Note that the hardware accelerator does not require high-performance embedded-cores and needs no high-capacity memory either, since it only deals with NVMe-commands and reuses the system-memory for data-management (via PRPs).

Bypassing a request is not a new idea, but it requires the proposed optimization of FLASHSHARE to apply such bypassing concept from the kernel to the firwmare. For example, bypassing scheme has been well investigated to improve the throughput of network [27]. While network kernel bypassing transfers data by directly mapping user memory to physical memory, the storage stack cannot simply adopt the same idea, due to ACID capability supports and block interface requirements. In addition, bypassing in block interface devices should still go through filesystem, page-cache, scheduler and interface-driver for user-level services. This introduces higher complexity and multiple interface boundaries than network, and also renders the direct mapping between user memory and phyiscal memory not a viable option. On the other hand, SPDK [39] is designed for a specific-purpose, namely, NVMe-over-Fabric that requires client-side file-systems or RocksDB-based applications, which is different from the datacenter's co-located workload scenario that FLASHSHARE works on.

Even though FLASHSHARE can remove a significant chunk of CPU-side overheads (around 79%, compared to naive-polling) with 20%$\sim$31% better user experience from the client-side, it also has a limit; FLASHSHARE is mainly designed towards accelerating the services in the cluster servers, but it unfortunately does not fit for the workload scenarios that rent computing-capability to multiple tenants, such as Infrastructure as a Service (IaaS). In our on-going work, we are extending FLASHSHARE with a different type of storage, such as multi-streamed (or ported) SSDs [52, 25, 49, 35] over diverse storage I/O virtualization techniques.

8 Acknowledgement

The authors thank Steven Swanson of UCSD for shepherding their paper. This research is mainly supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, Yonsei Future Research Grant (2017-22-0105) and MemRay grant (2015-11-1731). The authors thank Samsungs Jaeheon Jeong, Jongyoul Lee, Se-Jeong Jang and JooYoung Hwang for their SSD sample donations. N.S. Kim is supported in part by grants from NSF CNS-1557244 and CNS-1705047. M. Kandemir is supported in part by grants by NSF grants 1822923, 1439021, 1629915, 1626251, 1629129, 1763681, 1526750 and 1439057. Myoungsoo Jung is the corresponding author.

9 Conclusion

We propose FLASHSHARE, which punches through the storage stack from kernel to firmware, helping ULL SSDs satisfy different levels of user requirements. At the kernel level, we extend the data structures of the storage stack to pass attributes of (co-running) applications through all software modules of the underlying OS and device. Given such attributes, the block layer and NVMe driver of FLASHSHARE custom-manage the I/O scheduler and interrupt handler of NVMe. The target ULL SSD dynamically partitions the internal DRAM and adjust its caching strategies to meet diverse user demands. By taking full advantage of the ULL services, this holistic approach significantly reduces the inter-application I/O interferences in servers co-running multiple applications, without modifying any of the applications.

References

[1] ALIAN, M., ABULILA, A. H., JINDAL, L., KIM, D., AND KIM, N. S. Ncap: Network-driven, packet context-aware power management for client-server architecture. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 25–36.

[2] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[3] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), IEEE Computer Society, pp. 385–395.

[4] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing* (2007), ACM, pp. 1126–1130.

[5] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 3*, 4 (2004), 837–863.

[6] CHEONG, W., YOON, C., WOO, S., HAN, K., KIM, D., LEE, C., CHOI, Y., KIM, S., KANG, D., YU, G., ET AL. A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time. In *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International* (2018), IEEE, pp. 338–340.

[7] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, p. 42.

[8] FARRINGTON, N., AND ANDREYEV, A. Facebook's data center network architecture. In *Optical Interconnects Conference, 2013 IEEE* (2013), Citeseer, pp. 49–50.

[9] FREE SOFTWARE FOUNDATION. nice. `http://www.gnu.org/software/coreutils/manual/coreutils.html#nice-invocation`.

[10] GOUK, D., KWON, M., ZHANG, J., KOH, S., CHOI, W., KIM, N. S., KANDEMIR, M., AND JUNG, M. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *Proceedings of the 2018 51rd Annual IEEE/ACM International Symposium on Microarchitecture* (2018), IEEE Computer Society.

[11] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), IEEE, pp. 24–33.

[12] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The harey tortoise: Managing heterogeneous write performance in ssds. In *USENIX Annual Technical Conference* (2013), pp. 79–90.

[13] INTEL CORPORATION. Intel 535. `https://ark.intel.com/products/86734/Intel-SSD-535-Series-240GB-2_5in-SATA-6Gbs-16nm-MLC`, 2015.

[14] INTEL CORPORATION. Intel 750. `https://ark.intel.com/products/86740/Intel-SSD-750-Series-400GB-12-Height-PCIe-3_0-20nm-MLC`, 2015.

[15] INTEL CORPORATION. Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2016.

[16] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/o stack optimization for smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 309–320.

[17] JIN, Y., WEN, Y., AND CHEN, Q. Energy efficiency and server virtualization in data centers: An empirical investigation. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on* (2012), IEEE, pp. 133–138.

[18] JUNG, M. Exploring parallel data access methods in emerging non-volatile memory systems. *IEEE Transactions on Parallel & Distributed Systems*, 3 (2017), 746–759.

[19] JUNG, M., AND KANDEMIR, M. Middleware - firmware cooperation for high-speed solid state drives. In *Middleware '12* (2012).

[20] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review* (2013), vol. 41, ACM, pp. 203–216.

[21] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. Simplessd: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters PP*, 99 (2017), 1–1.

[22] JUNG, M.-S., IK PARK, C., AND OH, S.-J. Cooperative memory management. In *US 8745309* (2009).

[23] KIM, J., LEE, D., AND NOH, S. H. Towards slo complying ssds through ops isolation. In *FAST* (2015), pp. 183–189.

[24] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the i/o path: A holistic approach for application performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2017), FAST'17, USENIX Association, pp. 345–358.

[25] KIM, T., HAHN, S. S., LEE, S., HWANG, J., LEE, J., AND KIM, J. Pcstream: automatic stream allocation using program contexts. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018), USENIX Association.

[26] LEVERICH, J., AND KOZYRAKIS, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 4:1–4:14.

[27] LIU, J., WU, J., AND PANDA, D. K. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming 32*, 3 (2004), 167–198.

[28] LO, D., CHENG, L., GOVINDARAJU, R., BARROSO, L. A., AND KOZYRAKIS, C. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 301–312.

[29] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 450–462.

[30] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 248–259.

[31] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *FAST* (2003), vol. 3, pp. 115–130.

[32] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 57–70.

[33] MURUGAN, M., AND DU, D. H. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on* (2011), IEEE, pp. 1–12.

[34] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. *Informed prefetching and caching*, vol. 29. ACM, 1995.

[35] RHO, E., JOSHI, K., SHIN, S.-U., SHETTY, N. J., HWANG, J.-Y., CHO, S., LEE, D. D., AND JEONG, J. Fstream: managing flash streams in the file system. In *16th USENIX Conference on File and Storage Technologies* (2018), p. 257.

[36] SAMSUNG MEMORY SOLUTIONS LAB. Ultra-low latency with samsung z-nand ssd. `https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf`, 2017.

[37] SHIM, H., SEO, B.-K., KIM, J.-S., AND MAENG, S. An adaptive partitioning scheme for dram-based cache in solid state drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–12.

[38] SMITH, K. Garbage collection. *SandForce, Flash Memory Summit, Santa Clara, CA* (2011), 1–9.

[39] SOURCE, I. O. org. storage performance development kit (spdk), 2016.

[40] SUNGJOON KOH, CHANGRIM LEE, M. K. M. J. Exploring system challenges of ultra-low latency solid state drives. In *The 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2018), IEEE.

[41] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., LE MOAL, D., BUNKER, T., XU, J., SWANSON, S., ET AL. Dc express: shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX conference on File and Storage Technologies* (2014), USENIX Association, pp. 309–315.

[42] WANG, J., PARK, D., PAPAKONSTANTINOU, Y., AND SWANSON, S. Ssd in-storage computing for search engines. *IEEE Transactions on Computers* (2016).

[43] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ZHENG, C., LU, G., ZHAN, K., LI, X., AND QIU, B. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2014), pp. 488–499.

[44] WEST DIGIT. I/o latency optimization with polling. `https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final_0.pdf`.

[45] WIKI. Cpu cache. `https://en.wikipedia.org/wiki/CPU_cache`.

[46] WORKGROUP, N. E. Nvm express revision 1.3. `https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf`.

[47] YADGAR, G., FACTOR, M., LI, K., AND SCHUSTER, A. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems (TOCS) 29*, 2 (2011), 5.

[48] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *FAST* (2012), vol. 12, pp. 3–3.

[49] YANG, J., PANDURANGAN, R., CHOI, C., AND BALAKRISHNAN, V. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), ACM, p. 3.

[50] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 474–489.

[51] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First class support for interactivity in commodity operating systems. In *OSDI* (2008), vol. 8, pp. 73–86.

[52] YONG, H., JEONG, K., LEE, J., AND KIM, J.-S. vstream: virtual stream management for multi-streamed ssds. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018), USENIX Association.