Scalable Parallel Flash Firmware for Many-core Architectures

Jie Zhang[1], *Miryeong Kwon*[1], *Michael Swift*[2], *Myoungsoo Jung*[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], *University of Wisconsin at Madison*[2]
http://camelab.org

Abstract

NVMe is designed to unshackle flash from a traditional storage bus by allowing hosts to employ many threads to achieve higher bandwidth. While NVMe enables users to fully exploit all levels of parallelism offered by modern SSDs, current firmware designs are not scalable and have difficulty in handling a large number of I/O requests in parallel due to its limited computation power and many hardware contentions.

We propose `DeepFlash`, a novel manycore-based storage platform that can process more than a million I/O requests in a second (1MIOPS) while hiding long latencies imposed by its internal flash media. Inspired by a parallel data analysis system, we design the firmware based on many-to-many threading model that can be scaled horizontally. The proposed `DeepFlash` can extract the maximum performance of the underlying flash memory complex by concurrently executing multiple firmware components across many cores within the device. To show its extreme parallel scalability, we implement `DeepFlash` on a many-core prototype processor that employs dozens of lightweight cores, analyze new challenges from parallel I/O processing and address the challenges by applying concurrency-aware optimizations. Our comprehensive evaluation reveals that `DeepFlash` can serve around 4.5 GB/s, while minimizing the CPU demand on microbenchmarks and real server workloads.

1 Introduction

Solid State Disks (SSDs) are extensively used as caches, databases, and boot drives in diverse computing domains [37, 42, 47, 60, 74]. The organizations of modern SSDs and flash packages therein have undergone significant technology shifts [11, 32, 39, 56, 72]. In the meantime, new storage interfaces have been proposed to reduce overheads of the host storage stack thereby improving the storage-level bandwidth. Specifically, NVM Express (NVMe) is designed to unshackle flash from a traditional storage interface and enable users to take full advantages of all levels of SSD internal parallelism [13, 14, 54, 71]. For example, it provides streamlined commands and up to 64K deep queues, each with up to 64K entries. There is massive parallelism in the backend where requests are sent to tens or hundreds of flash packages. This enables assigning queues to different applications; multiple deep NVMe queues allow the host to employ many threads thereby maximizing the storage utilization.

An SSD should handle many concurrent requests with its massive internal parallelism [12, 31, 33, 34, 61]. However, it is difficult for a single storage device to manage the tremendous number of I/O requests arriving in parallel over many NVMe queues. Since highly parallel I/O services require simultaneously performing many SSD internal tasks, such as address translation, multi-queue processing, and flash scheduling, the SSD needs multiple cores and parallel implementation for a higher throughput. In addition, as the tasks inside the SSD increase, the SSD must address several scalability challenges brought by garbage collection, memory/storage contention and data consistency management when processing I/O requests in parallel. These new challenges can introduce high computation loads, making it hard to satisfy the performance demands of diverse data-centric systems. Thus, the high-performance SSDs require not only a powerful CPU and controller but also an efficient flash firmware.

We propose `DeepFlash`, a manycore-based NVMe SSD platform that can process more than one million I/O requests within a second (1MIOPS) while minimizing the requirements of internal resources. To this end, we design a new flash firmware model, which can extract the maximum performance of hundreds of flash packages by concurrently executing firmware components atop a manycore processor. The layered flash firmware in many SSD technologies handles the internal datapath from PCIe to physical flash interfaces as a single heavy task [66, 76]. In contrast, `DeepFlash` employs a many-to-many threading model, which multiplexes any number of threads onto any number of cores in firmware.

Specifically, we analyze key functions of the layered flash firmware and decompose them into multiple modules, each is scaled independently to run across many cores. Based on the analysis, this work classifies the modules into a queue-gather stage, a trans-apply stage, and a flash-scatter stage, inspired by a parallel data analysis system [67]. Multiple threads on the queue-gather stage handle NVMe queues, while each thread on the flash-scatter stage handles many flash devices on a channel bus. The address translation between logical block addresses and physical page numbers is simultaneously performed by many threads at the trans-apply stage. As each stage can have different numbers of threads, contention between the threads for shared hardware resources and structures, such as mapping table, metadata and memory management structures can arise. Integrating many cores in the scalable flash firmware design also introduces data consistency, coherence and hazard issues. We analyze new challenges arising from concurrency, and address them by applying concurrency-aware optimization techniques to each stage, such as parallel queue processing, cache bypassing and background work for time-consuming SSD internal tasks.

We evaluate a real system with our hardware platform that implements `DeepFlash` and internally emulates low-level flash media in a timing accurate manner. Our evaluation results show that `DeepFlash` successfully provides more than 1MIOPS with a dozen of simple low-power cores for all reads and writes with sequential and random access patterns. In addition, `DeepFlash` reaches 4.5 GB/s (above 1MIOPS), on average, under the execution of diverse real server workloads. The main contributions of this work are summarized as below:

• **Many-to-many threading firmware.** We identify scalability and parallelism opportunities for high-performance flash firmware. Our many-to-many threading model allows future manycore-based SSDs to dynamically shift their computing power based on different workload demands without any hardware modification. `DeepFlash` splits all functions from the existing layered firmware architecture into three stages, each with one or more thread groups. Different thread groups can communicate with each other over an on-chip interconnection network within the target SSD.

• **Parallel NVMe queue management.** While employing many NVMe queues allows the SSD to handle many I/O requests through PCIe communication, it is hard to coordinate simultaneous queue accesses from many cores. `DeepFlash` dynamically allocates the cores to process NVMe queues rather than statically assigning one core per queue. Thus, a single queue is serviced by multiple cores, and a single core can service multiple queues, which can deliver full bandwidth for both balanced and unbalanced NVMe I/O workloads. We show that this parallel NVMe queue processing exceeds the performance of the static core-per-queue allocation by 6x, on average, when only a few queues are in use. `DeepFlash` also balances core utilization over computing resources.

• **Efficient I/O processing.** We increase the parallel scalability of many-to-many threading model by employing non-blocking communication mechanisms. We also apply simple but effective lock and address randomization methods, which can distribute incoming I/O requests across multiple address translators and flash packages. The proposed method minimizes the number of hardware core to achieve 1MIOPS. Putting all it together, `DeepFlash` improves bandwidth by 3.4× while significantly reducing CPU requirements, compared to conventional firmware. Our `DeepFlash` requires only a dozen of lightweight in-order cores to deliver 1MIOPS.

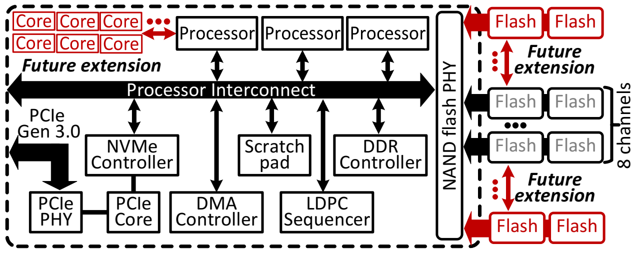


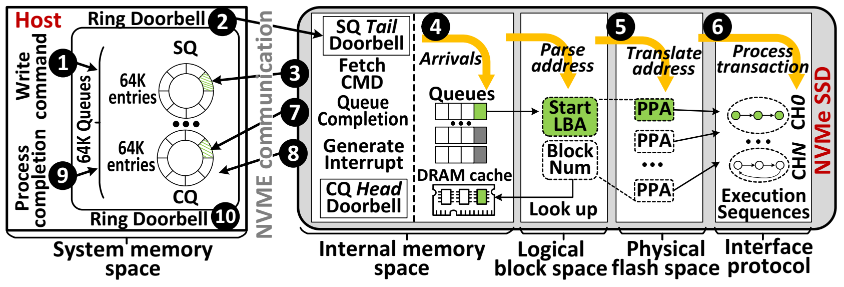
Figure 1: Overall architecture of an NVMe SSD.

2 Background

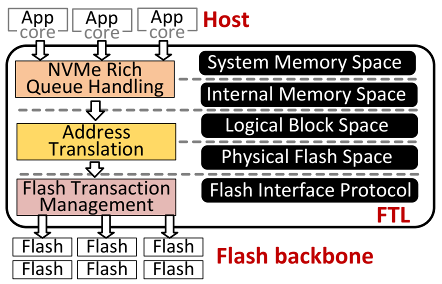
2.1 High Performance NVMe SSDs

Baseline. Figure 1 shows an overview of a high-performance SSD architecture that Marvell recently published [43]. The host connects to the underlying SSD through four Gen 3.0 PCIe lanes (4 GB/s) and a PCIe controller. The SSD architecture employs three embedded processors, each employing two cores [27], which are connected to an internal DRAM controller via a processor interconnect. The SSD employs several special-purpose processing elements, including a low-density parity-check (LDPC) sequencer, data transfer (DMA) engine, and scratch-pad memory for metadata management. All these multi-core processors, controllers, and components are connected to a flash complex that connects to eight channels, each connecting to eight packages, via flash physical layer (PHY). We select this multicore architecture description as our reference and extend it, since it is only documented NVMe storage architecture that employs multiple cores at this juncture, but other commercially available SSDs also employ a similar multi-core firmware controller [38, 50, 59].

Future architecture. The performance offered by these devices is by far below 1MIOPS. For higher bandwidth, a future device can extend storage and processor complexes with more flash packages and cores, respectively, which are highlighted by red in the figure. The bandwidth of each flash package is in practice tens of MB/s, and thus, it requires employing more flashes/channels, thereby increasing I/O parallelism. This flash-side extension raises several architectural issues. First, the firmware will make frequent SSD-internal memory accesses that stress the processor complex. Even though the PCIe core, channel and other memory control logic may be implemented, metadata information increases for the extension, and its access frequency gets higher to achieve 1MIOPS. In addition, DRAM accesses for I/O buffering can be a critical bottleneck to hide flash's long latency. Simply making cores faster may not be sufficient because the processors will suffer from frequent stalls due to less locality and contention at memory. This, in turn, makes each core bandwidth lower, which should be addressed with higher parallelism on computation parts. We will explain the current architecture and show why it is non-scalable in Section 3.



(a) NVMe SSD datapath.



(b) Flash firmware.

Figure 2: Datapath from PCIe to Flash and overview of flash firmware.

Datapath from PCIe to flash. To understand the source of scalability problems, it requires being aware of the internal datapath of NVMe SSDs and details of the datapath management. Figure 2a illustrates the internal datapath between PCIe and NV-DDR [7, 53], which is managed by NVMe [16] and ONFi [69] protocols, respectively. NVMe employs multiple device-side *doorbell* registers, which are designed to minimize handshaking overheads. Thus, to issue an I/O request, applications submit an NVMe command to a *submission queue (SQ)* (❶) and notify the SSD of the request arrival by writing to the doorbell corresponding to the queue (❷). After fetching a host request from the queue (❸), flash firmware, known as *flash translation layer* (FTL), parses the I/O operation, metadata, and data location of the target command (❹). The FTL then translates the *physical page address (PPA)* from the host's *logical block address (LBA)* (❺). In the meantime, the FTL also orchestrates data transfers. Once the address translation is completed, the FTL moves the data, based on the I/O timing constraints defined by ONFi (❻). A *completion queue (CQ)* is always paired with an SQ in the NVMe protocol, and the FTL writes a result to the CQ and updates the tail doorbell corresponding to the host request. The FTL notifies the queue completion to the host (❼) by generating a message-signaled interrupt (MSI) (❽). The host can finish the I/O process (❾) and acknowledge the MSI by writing the head doorbell associated with the original request (❿).

2.2 Software Support

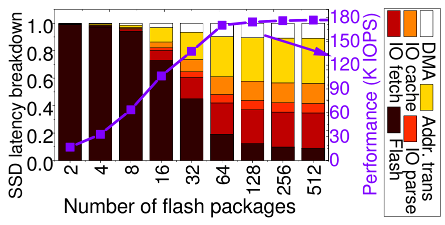
Flash firmware. Figure 2b shows the processes of the FTL, which performs the steps ❸ $\sim$ ❽. The FTL manages NVMe queues/requests and responds to the host requests by processing the corresponding doorbell. The FTL then performs address translations and manages memory transactions for the flash media. While prior studies [34, 48, 49] distinguish host command controls and flash transaction management as the host interface layer (HIL) and flash interface layer (FIL), respectively, in practice, both modules are implemented as a layered firmware calling through functions of event-based codes with a single thread [57, 65, 70]. The performance of the layered firmware is not on the critical path as flash latency is several orders of magnitude longer than one I/O command processing latency. However, SSDs require a large number of flash packages and queues to handle more than a thousand requests per *m*sec. When increasing the number of underlying flash packages, the FTL requires powerful computation not only to spread I/O requests across flash packages but also to process I/O commands in parallel. We observe that, compute latency keeps increasing due to non-scalable firmware and takes 93.6% of the total I/O processing time in worst case.

Memory spaces. While the FTL manages the logical block space and physical flash space, it also handles SSD's internal memory space and accesses to host system memory space (cf. Figure 2b). SSDs manage internal memory for caching incoming I/O requests and the corresponding data. Similarly, the FTL uses the internal memory for metadata and NVMe queue management (e.g., SQs/CQs). In addition, the FTL requires accessing the host system memory space to transfer actual data contents over PCIe. Unfortunately, a layered firmware design engages in accesses to memory without any constraint and protection mechanism, which can make the data inconsistent and incoherent in simultaneous accesses. However, computing resources with more parallelism must increase to achieve more than 1MIOPS, and many I/O requests need processing simultaneously. Thus, all shared memory spaces of a manycore SSD platform require appropriate concurrency control and resource protection, similar to virtual memory.

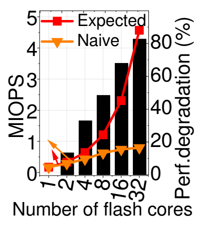
3 Challenges to Exceeding 1MIOPS

To understand the main challenges in scaling SSD firmware, we extend the baseline SSD architecture in a highly scalable environment: Intel many-integrated cores (MIC) [18]. We select this processor platform, because its architecture uses a simple in-order and low-frequency core model, but provides a high core count to study parallelism and scalability. The platform internally emulates low-level flash modules with hardware-validated software[1], so that the flash complex can be extended by adding more emulated channels and flash resources: the number of flash (quad-die package, QDP) varies from 2 to 512. Note that MIC is a prototyping platform used

[1]This emulation framework is validated by comparing with Samsung Z-SSD prototype [4], multi-stream 983 DCT (Proof of Concept), 850 Pro [15] and Intel NVMe 750 [25]. **The software will be publicly available.**



(a) Flash scaling.



(b) Core scaling.

Figure 3: Perf. with varying flash packages and cores.

for *only* exploring the limit of scalability, rather than as a suggestion for actual SSD controller.

Flash scaling. The bandwidth of a low-level flash package is several orders of magnitude lower than the PCIe bandwidth. Thus, SSD vendors integrate many flash packages over multiple channels, which can in parallel serve I/O requests managed by NVMe. Figure 3a shows the relationship of bandwidth and execution latency breakdown with various number of flash packages. In this evaluation, we emulate an SSD by creating a layered firmware instance in a single MIC core, in which two threads are initialized to process the tasks of HIL and FTL, respectively. We also assign 16 MIC cores (one core per flash channel) to manage flash interface subsystems. We evaluate the performance of the configured SSD emulation platform by testing 4KB sequential writes. For the breakdown analysis, we decompose total latency into i) NVMe management (`I/O parse` and `I/O fetch`), ii) `I/O cache`, iii) `address translation` (including flash scheduling), vi) NVMe data transfers (`DMA`) and v) flash operations (`Flash`). One can observe from the figure that the SSD performance saturates at 170K IOPS with 64 flash packages, connected over 16 channels. Specifically, the flash operations are the main contributor of the total execution time in cases where our SSD employs tens of flash packages (73% of the total latency). However, as the number of flash packages increases (more than 32), the layered firmware operations on a core become the performance bottleneck. NVMe management and address translation account for 41% and 29% of the total time, while flash only consumes 12% of the total cycles.

There are two reasons that flash firmware turns into the performance bottleneck with many underlying flash devices. First, NVMe queues can supply many I/O requests to take advantages of the SSD internal parallelism, but a single-core SSD controller is insufficient to fetch all the requests. Second, it is faster to parallelize I/O accesses across many flash chips than performing address translation only on one core. These new challenges make it difficult to fully leverage the internal parallelism with the conventional layered firmware model.

Core scaling. To take flash firmware off the critical path in scalable I/O processing, one can increase computing power with the execution of many firmware instances. This approach can allocate a core per NVMe SQ/CQ and initiate one layered firmware instance in each core. However, we observe that this naive approach cannot successfully address the burdens brought by flash firmware. To be precise, we evaluate IOPS with varying number of cores, ranging from 1 to 32. Figure 3b compares the performance of aforementioned naive many-core approach (e.g., `Naive`) with the system that expects perfect parallel scalability (e.g., `Expected`). `Expected`'s performance is calculated by multiplying the number of cores with IOPS of `Naive` built on a single core SSD. One can observe from this figure that `Naive` can only achieve 813K IOPS even with 32 cores, which exhibits 82.6% lower performance, compared to `Expected`. This is because contention and consistency management for the memory spaces of internal DRAM (cf. Section 5.3) introduces significant synchronization overheads. In addition, the FTL must serialize the I/O requests to avoid hazards while processing many queues in parallel. Since all these issues are not considered by the layered firmware model, it should be re-designed by considering core scaling.

The goal of our new firmware is to fully parallelize multiple NVMe processing datapaths in a highly scalable manner while minimizing the usage of SSD internal resources. `DeepFlash` requires only 12 in-order cores to achieve 1M or more IOPS.

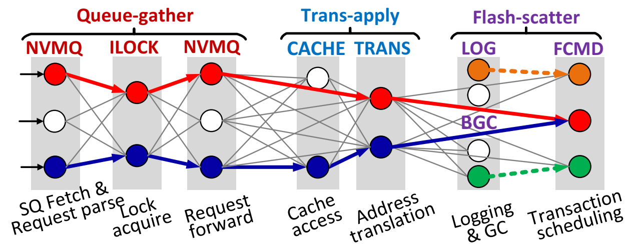


Figure 4: Many-to-many threading firmware model.

4 Many-to-Many Threading Firmware

Conventional FTL designs are unable to fully convert the computing power brought by a manycore processor to storage performance, as they put all FTL tasks into a single large block of the software stack. In this section, we analyze the functions of the traditional FTLs and decompose them into seven different function groups: 1) *NVMe queue handling (NVMQ)*, 2) *data cache (CACHE)*, 3) *address translation (TRANS)*, 4) *index lock (ILOCK)*, 5) *logging utility (LOG)*, 6) *background garbage collection utility (BGC)*, and 7) *flash command and transaction scheduling (FCMD)*. We then reconstruct the key function groups from the ground up, keeping in mind concurrency, and deploy our reworked firmware modules across multiple cores in a scalable manner.

4.1 Overview

Figure 4 shows our `DeepFlash`'s many-to-many threading firmware model. The firmware is a set of modules (i.e., *threads*) in a request-processing network that is mapped to a set of processors. Each thread can have a firmware operation, and the task can be scaled by instantiating into multiple parallel threads, referred to as *stages*. Based on different data

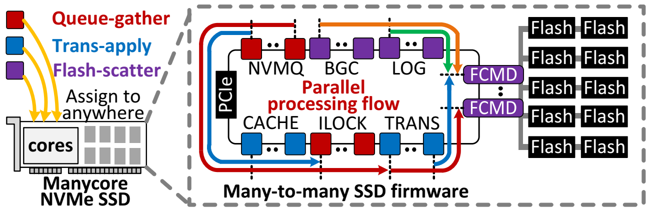
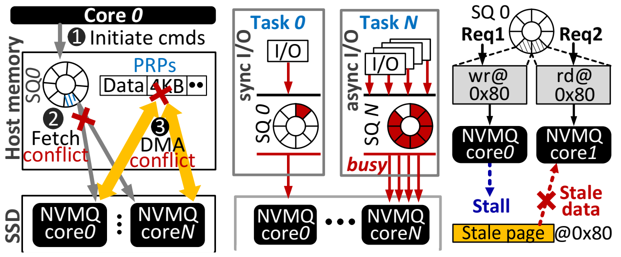


Figure 5: Firmware architecture.



(a) Data contention (1:N). (b) Unbalanced task. (c) I/O hazard.

Figure 6: Challenges of NVMQ allocation (SQ:NVMQ).

processing flows and tasks, we group the stages into *queue-gather*, *trans-apply*, and *flash-scatter* modules. The queue-gather stage mainly parses NVMe requests and collects them to the SSD-internal DRAM, whereas the trans-apply stage mainly buffers the data and translates addresses. The flash-scatter stage spreads the requests across many underlying flash packages and manages background SSD-internal tasks in parallel. This new firmware enables scalable and flexible computing, and highly parallel I/O executions.

All threads are maximally independent, and I/O requests are always processed from left to right in the thread network, which reduces the hardware contentions and consistency problems, imposed by managing various memory spaces. For example, two independent I/O requests are processed by two different network paths (which are highlighted in Figure 4 by red and blue lines, respectively). Consequently, it can simultaneously service incoming I/O requests as many network paths on as DeepFlash can create. In contrast to the other threads, background threads are asynchronous with the incoming I/O requests or host-side services. Therefore, they create their own network paths (dashed lines), which perform SSD internal tasks at background. Since each stage can process a different part of an I/O request, DeepFlash can process multiple requests in a pipeline manner. Our firmware model also can be simply extended by adding more threads based on performance demands of the target system.

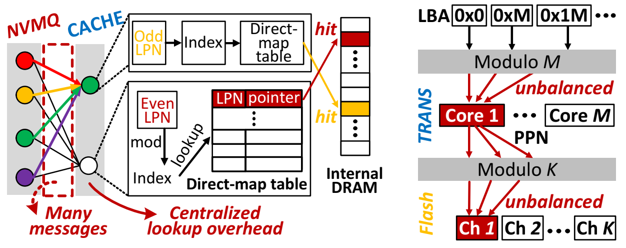
Figure 5 illustrates how our many-to-many threading model can be applied to and operate in the many-core based SSD architecture of DeepFlash. While the procedure of I/O services is managed by many threads in the different data processing paths, the threads can be allocated in any core in the network, in a parallel and scalable manner.

4.2 Queue-gather Stage

NVMe queue management. For high performance, NVMe supports up to 64K queues, each up to 64K entries. As shown in Figure 6a, once a host initiates an NVMe command to an SQ and writes the corresponding doorbell, the firmware fetches the command from the SQ and decodes a non-contiguous set of host physical memory pages by referring a kernel list structure [2], called a *physical region page* (PRP) [23]. Since the length of the data in a request can vary, its data can be delivered by multiple data frames, each of which is usually 4KB. While all command information can be retrieved by the device-level registers and SQ, the contents of such data frames exist across non-contiguous host-side DRAM (for a single I/O request). The firmware parses the PRP and begins DMA for multiple data frames per request. Once all the I/O services associated with those data frames complete, firmware notifies the host of completion through the target CQ. We refer to all the tasks, related to this NVMe command and queue management, as *NVMQ*.

A challenge to employ many cores for parallel queue processing is that, multiple NVMQ cores may simultaneously fetch a same set of NVMe commands from a single queue. This in turn accesses the host memory by referring a same set of PRPs, which makes the behaviors of parallel queue accesses undefined and non-deterministic (Figure 6a). To address this challenge, one can make each core handle only a set of SQ/CQ, and therefore, there is no contention, caused by simultaneous queue processing or PRP accesses (Figure 6b). In this "static" queue allocation, each NVMQ core fetches a request from a different queue, based on the doorbell's queue index and brings the corresponding data from the host system memory to SSD internal memory. However, this static approach requires that the host balance requests across queues to maximize the resource utilization of NVMQ threads. In addition, it is difficult to scale to a large number of queues. DeepFlash addresses these challenges by introducing a dynamic I/O serialization, which allows multiple NVMQ threads to access each SQ/CQ in parallel while avoiding a consistency violation. Details of NVMQ will be explained in Section 5.1.

I/O mutual exclusion. Even though the NVMe specification does not regulate the processing ordering of NVMe commands in a range from where the head pointer indicates to the entry that the tail pointer refers to [3], users may expect that the SSD processes the requests in the order that users submitted. However, in our DeepFlash, many threads can simultaneously process I/O requests in any order of accesses. It can make the order of I/O processing different with the order NVMe queues (and users) expected, which may in turn introduce an I/O hazard or a consistency issue. For example, Figure 6c shows a potential problem brought by parallel I/O processing. In this figure, there are two different I/O requests from the same NVMe SQ, request-1 (a write) and request-2 (a read), which create two different paths, but target to the same PPA. Since these two requests are processed by different NVMQ threads, the request-2 can be served from the target slightly earlier than the request-1. The request-1 then will be



(a) Main procedure of CACHE. (b) Shards (TRANS).

Figure 7: Challenge analysis in CACHE and TRANS.

stalled, and the request-2 will be served with stale data. During this phase, it is also possible that any thread can invalidate the data while transferring or buffering them out of order.

While serializing the I/O request processing with a strong ordering can guarantee data consistency, it significantly hurts SSD performance. One potential solution is introducing a locking system, which provides a lock per page. However, per-page lock operations within an SSD can be one of the most expensive mechanisms due to various I/O lengths and a large storage capacity of the SSD. Instead, we partition physical flash address space into many *shards*, whose access granularity is greater than a page, and assign an index-based lock to each shard. We implement the index lock as a red-black tree and make this locking system as a dedicated thread (ILOCK). This tree helps ILOCK quickly identify which lock to use, and reduces the overheads of lock acquisition and release. Nevertheless, since NVMQ threads may access a few ILOCK threads, it also can be resource contention. `DeepFlash` optimizes ILOCK by redistributing the requests based on lock ownership (cf., Section 5.2). Note that there is no consistency issue if the I/O requests target different LBAs. In addition, as most OSes manage the access control to prevent different cores from accessing the same files [19, 41, 52], I/O requests from different NVMe queues (mapping to different cores) access different LBAs, which also does not introduce the consistency issue. Therefore, `DeepFlash` can solve the I/O hazard by guaranteeing the ordering of I/O requests, which are issued to the same queue and access the same LBAs, while `DeepFlash` can process other I/O requests out of order.

4.3 Trans-apply Stage

Data caching and buffering. To appropriately handle NVMe's parallel queues and achieve more than 1MIOPS, it is important to utilize the internal DRAM buffer efficiently. Specifically, even though modern SSDs enjoy the massive internal parallelism stemming from tens or hundreds of flash packages, the latency for each chip is orders of magnitude longer than DRAM [22, 45, 46], which can stall NVMQ's I/O processing. `DeepFlash`, therefore, incorporates CACHE threads that incarnate SSD internal memory as a burst buffer by mapping LBAs to DRAM addresses rather than flash ones. The data buffered by CACHE can be drained by striping requests across many flash packages with high parallelism.

As shown in Figure 7a, each CACHE thread has its own mapping table to record the memory locations of the buffered requests. CACHE threads are configured with a traditional direct-map cache to reduce the burden of table lookup or cache replacement. In this design, as each CACHE thread has a different memory region to manage, NVMQ simply calculates the index of the target memory region by modulating the request's LBA, and forwards the incoming requests to the target CACHE. However, since all NVMQ threads possibly communicate with a CACHE thread for every I/O request, it can introduce extra latency imposed by passing messages among threads. In addition, to minimize the number of cores that `DeepFlash` uses, we need to fully utilize the allocated cores and dedicate them to each firmware operation while minimizing the communication overhead. To this end, we put a cache tag inquiry method in NVMQ and make CACHE threads fully handle cache hits and evictions. With the tag inquiry method, NVMQ can create a bypass path, which can remove the communication overheads (cf. Section 5.3).

Parallel address translation. The FTL manages physical blocks and is aware of flash-specific behavior such as erase-before-write and asymmetric erase and read/write operation unit (block vs. page). We decouple FTL address translation from system management activities such as garbage collection or logging (e.g., journaling) and allocate the management to multiple threads. The threads that perform this simplified address translation are referred to as *TRANS*. To translate addresses in parallel, it needs to partition both LBA space and PPA space and allocate them to each TRANS thread.

As shown in Figure 7b, a simple solution is to split a single LBA space into m numbers of address chunks, where m is the number of TRANS threads, and map the addresses by wrapping around upon reaching m. To take advantage of channel-level parallelism, it can also separate a single PPA space into k shards, where k is the number of underlying channels, and map the shards to each TRANS with arithmetic modulo k. While this address partitioning can make all TRANS threads operate in parallel without interference, unbalanced I/O accesses can activate a few TRANS threads or channels. This can introduce a poor resource utilization and many resource conflicts and stall a request service on the fly. Thus, we randomize the addresses when partitioning the LBA space with simple XOR operators. This can scramble LBA and statically assign all incoming I/O requests across different TRANS threads in an evenly distributed manner. We also allocate all the physical blocks of the PPA space to each TRANS in a round-robin fashion. This block-interleaved virtualization allows us to split the PPA space with finer granularity.

4.4 Flash-scatter Stage

Background task scheduling. The datapath for garbage collection (GCs) can be another critical path to achieve high bandwidth as it stalls many I/O services while reclaiming

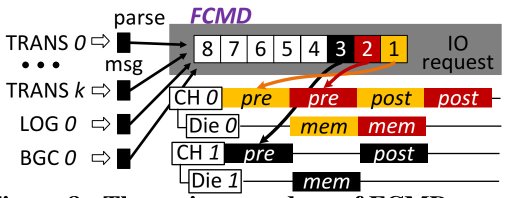


Figure 8: The main procedure of FCMD cores.

flash block(s). In this work, GCs can be performed in parallel by allocating separate core(s), referred to as *BGC*. BGC(s) records the block numbers that have no more entries to write when TRANS threads process incoming I/O requests. BGC then merges the blocks and update the mapping table of corresponding TRANS in behind I/O processing. Since a thread in TRANS can process address translations during BGC's block reclaims, it would introduce a consistency issue on mapping table updates. To avoid conflicts with TRANS threads, BGC reclaims blocks and updates the mapping table at background when there is no activity in NVMQ and the TRANS threads complete translation tasks. If the system experiences a heavy load and clean blocks are running out, our approach performs on-demand GC. To avoid data consistency issue, we only block the execution of the TRANS thread, which is responsible for the address translation of the reclaiming flash block.

Journalling. SSD firmware requires journalling by periodically dumping the local metadata of TRANS threads (e.g., mapping table) from DRAM to a designated flash. In addition, it needs to keep track of the changes, which are not dumped yet. However, managing consistency and coherency for persistent data can introduce a burden to TRANS. Our `DeepFlash` separates the journalling from TRANS and assigns it to a LOG thread. Specifically, TRANS writes the LPN-to-PPN mapping information of a FTL page table entry (PTE) to out-of-band (OoB) of the target flash page [64] in each flash program operation (along with the per-page data). In the meantime, LOG periodically reads all metadata in DRAM, stores them to flash, and builds a checkpoint in the background. For each checkpoint, LOG records a version, a commit and a page pointer indicating the physical location of the flash page where TRANS starts writing to. At a boot time, LOG checks sanity by examining the commit. If the latest version is staled, LOG loads a previous version and reconstructs mapping information by combining the checkpointed table and PTEs that TRANS wrote since the previous checkpoint.

Parallel flash accesses. At the end of the `DeepFlash` network, the firmware threads need to i) compose flash transactions respecting flash interface timing and ii) schedule them across different flash resources over the flash physical layer (PHY). These activities are managed by separate cores, referred to as *FCMD*. As shown in Figure 8, each thread in FCMD parses the PPA translated by TRANS (or generated by BGC/LOG) into the target channel, package, chip and plane numbers. The threads then check the target resources' availability and compose flash transactions by following the underlying flash interface protocol. Typically, memory tim-

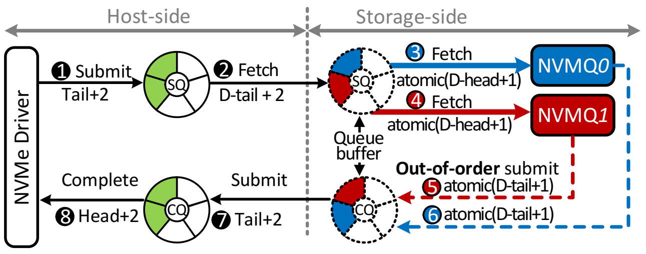


Figure 9: Dynamic I/O serialization (DIOS).

ings within a flash transaction can be classified by *pre-dma*, *mem-op* and *post-dma*. While pre-dma includes operation command, address, and data transfer (for writes), post-dma is composed by completion command and another data transfer (for reads). Memory operations of the underlying flash are called mem-op in this example. FCMD(s) then scatters the composed transactions over multiple flash resources. During this time, all transaction activities are scheduled in an interleaved way, so that it can maximize the utilization of channel and flash resources. The completion order of multiple I/O requests processed by this transaction scheduling can be spontaneously an out-of-order.

In our design, each FCMD thread is statically mapped to one or more channels, and the number of channels that will be assigned to the FCMD thread is determined based on the SSD vendor demands (and/or computing power).

5 Optimizing DeepFlash

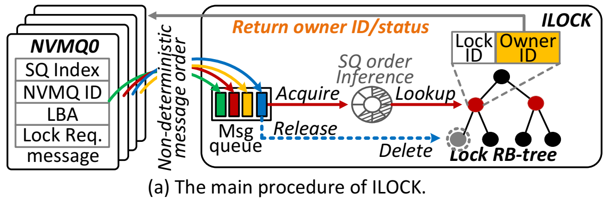
While the baseline `DeepFlash` architecture distributes functionality with many-to-many threading, there are scalability issues. In this section, we will explain the details of thread optimizations to increase parallel scalability that allows faster, more parallel implementations.

5.1 Parallel Processing for NVMe Queue

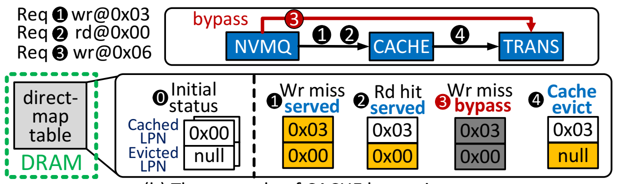
To address the challenges of the static queue allocation approach, we introduce the *dynamic I/O serialization (DIOS)*, which allows a variable ratio of queues to cores. DIOS decouples the fetching and parsing processes of NVMe queue entries. As shown in Figure 9, once a NVMQ thread fetches a batch of NVMe commands from a NVMe queue, other NVMQ threads can simultaneously parse the fetched NVMe queue entries. This allows all NVMQ threads to participate in processing the NVMe queue entries from the same queue or multiple queues. Specifically, DIOS allocates a storage-side SQ buffer (per SQ) in a shared memory space (visible to all NVMQ threads) when the host initializes NVMe SSD. If the host writes the tail index to the doorbell, a NVMQ thread fetches multiple NVMe queue entries and copies them (not actual data) to the SQ buffer. All NVMQ threads then process the NVMe commands existing in the SQ buffer in parallel. The batch copy is performed per 64 entries or till the tail for SQ and CQ points a same position. Similarly, DIOS creates a CQ buffer (per CQ) in the shared memory. NVMQ threads update the CQ buffer instead of the actual CQ as an out of order, and flush the NVMe completion messages from the CQ buffer to the CQ in batch. This allows multiple threads update an NVMe queue in parallel without a modification of the NVMe protocol and host side storage stack. Another technical challenge for processing a queue in parallel is that the head and tail pointers of SQ and CQ buffers are also shared resources, which requires a protection for simultaneous access. `DeepFlash` offers DIOS's head (*D-head*) and tail (*D-tail*) pointers, and allows NVMQ threads to access SQ and CQ through those pointers, respectively. Since D-head and D-tail pointers are managed by gcc atomic built-in function, `__sync_fetch_and_add` [21], and the core allocation is performed by all NVMQ threads, in parallel, the host memory can be simultaneously accessed but at different locations.

5.2 Index Lock Optimization

When multiple NVMQ threads contend to acquire or release the same lock due to their same target address range, it can raise two technical issues: i) lock contention and ii) low resource utilization of NVMQ. As shown in Figure 10a, an ILOCK thread sees all incoming lock requests (per page by LBA) through its message queue. This queue sorts the messages based on SQ indices, and each message maintains thread request structure that includes an SQ index, NVMQ ID, LBA, and lock request information (e.g., acquire and release). Since the order of queue's lock requests is non-deterministic, in a case of contention on acquisition, it must perform I/O services by respecting the order of requests in the corresponding SQ. Thus, the ILOCK thread infers the SQ order by referring to the SQ index in the message queue if the target LBA with the lock request has a conflict. It then checks the red-black (RB) tree whose LBA-indexed node contains the lock number and owner ID that already acquired the corresponding address. If there is no node in the lock RB tree, the ILOCK thread allocates a node with the request's NVMQ ID. When ILOCK receives a release request, it directly removes the target node without an SQ inference process. If the target address is already held by another NVMQ thread, the lock requester can be stalled until the corresponding I/O service is completed. Since low-level flash latency takes hundreds of microseconds to a few milliseconds, the stalled NVMQ can hurt overall performance. In our design, ILOCK returns the owner ID for all lock acquisition requests rather than returning simply acquisition result (e.g., false or fail). The NVMQ thread receives the ID of the owning NVMQ thread, and can forward the request there to be processed rather than communicating with ILOCK again. Alternatively, the NVMQ thread can perform other tasks, such as issuing the I/O service to TRANS or CACHE. The insight behind this *forwarding* is that if another NVMQ owns the corresponding lock of request, then forwards the request to owner and stop further communication with ILOCK.



(a) The main procedure of ILOCK.



(b) The example of CACHE bypassing.

Figure 10: Optimization details.

This, in turn, can free the NVMQ thread from waiting for the lock acquisition, which increases the parallelism of DIOS.

5.3 Non-blocking Cache

To get CACHE off the critical path, we add a direct path between NVMQ and TRANS threads and make NVMQ threads access CACHE threads "only if" there is data in CACHE. We allocate direct-map table(s) in a shared memory space to accommodate the cache metadata so that NVMQ threads can lookup the cache metadata on their own and send I/O requests only if there is a hit. However, this simple approach may introduce inconsistency between the cache metadata of the direct-map table and target data of the CACHE. When a write evicts a dirty page from the burst buffer, the metadata of such evicted page is removed from the direct-map table immediately. However, the target data of the evicted page may still stay in the burst buffer, due to the long latency of a flash write. Therefore, when a dirty page is in progress of eviction, read requests, which target for the same page, may access stale data from the flash. To coordinate the direct-map table and CACHE correctly, we add "evicted LPN" field in each map table entry that presents the page number, being in eviction (cf. Figure 10b). In this example of the figure, we assume the burst buffer is a direct mapped cache with 3 entries. The request (Req ❶) evicts the dirty page at LPN 0x00. Thus, NVMQ records the LPN of Req ❶ in the cached LPN field of the direct-map table and moves the address of the evicted page to its evicted LPN field. Later, as the LPN of Req ❷ (the read at 0x00) matches with the evicted LPN field, Req ❷ is served by CACHE instead of accessing the flash. If CACHE is busy in evicting the dirty page at LPN 0x00, Req ❸ (the write at 0x06) has to be stalled. To address this, we make Req ❸ directly bypass CACHE. Once the eviction successfully completes, CACHE clears the evicted LPN field (❹).

To make this *non-blocking cache* more efficient, we add a simple randomizing function to retrieve the target TRANS index for NVMQ and CACHE threads, which can evenly distribute their requests in a static manner. This function performs an XOR operation per bit for all the bit groups and generates the target TRANS index, which takes less than 20 ns. The randomization allows queue-gather stages to issue requests to TRANS by addressing load imbalance.

6 Evaluation

Implementation platform. We set up an accurate SSD emulation platform by respecting the real NVMe protocol, the timing constraints for flash backbone and the functionality of a flexible firmware. Specifically, we emulate a manycore-based SSD firmware by using a MIC 5120D accelerator that employs 60 lightweight in-order cores (4 hardware threads per core) [28]. The MIC cores operate at 1GHz and are implemented by applying low power techniques such as short in-order pipeline. We emulate the flash backbone by modelling various flash latencies, different levels of parallelism (i.e., channel/way/flash) and the request conflicts for flash resources. Our flash backbone consists of 16 channels, each connecting 16 QDP flash packages [69]; we observed that the performance of both read and write operations on the backbone itself is not the bottleneck to achieve more than 1 MIOPS. The NVMe interface on the accelerator is also fully emulated by wrapping Intel's symmetric communications interface (SCIF) with an NVMe emulation driver and controller that we implemented. The host employs a Xeon 16-core processor and 256 GB DRAM, running Linux kernel 2.6.32 [62]. It should be noted that this work uses MIC to explore the scalability limits of the design; the resulting software can run with fewer cores if they are more powerful, but the design can now be about what is most economic and power efficient, rather than whether the firmware can be scalable.

Configurations. `DeepFlash` is the emulated SSD platform including all the proposed designs of this paper. Compared to `DeepFlash`, `BaseDeepFlash` does not apply the optimization techniques (described in Section 5). We evaluate the performance of a real Intel customer-grade SSD (`750SSD`) [25] and high-performance NVMe SSD (`4600SSD`) [26] for a better comparison. We also emulate another SSD platform (`ManyLayered`), which is an approach to scale up the layered firmware on many cores. Specifically, `ManyLayered` statically splits the SSD hardware resources into multiple subsets, each containing the resources of one flash channel and running a layered firmware independently. For each layered firmware instance, `ManyLayered` assigns a pair of threads: one is used for managing flash transaction, and another is assigned to run HIL and FTL. All these emulation platforms use "12 cores" by default. Lastly, we also test different flash technologies such as SLC, MLC, TLC, each of which latency characteristics are extracted from [44], [45] and [46], respectively. By default, the MLC flash array in pristine state is used for our evaluations. The details of SSD platform are in Table 1.

Workloads. In addition to microbenchmarks (reads and writes with sequential and random patterns), we test diverse server workloads, collected from Microsoft Production Server (MPS) [35], FIU SRCMap [63], Enterprise, and FIU IODedup [40]. Each workload exhibits various request sizes, ranging from 4KB to tens of KB, which are listed in Table 1. Since all the workload traces are collected from the narrow-queue SATA hard disks, replaying the traces with the original timestamps cannot fully utilize the deep NVMe queues, which in turn conceals the real performance of SSD [29]. To this end, our trace replaying approach allocates 16 worker threads in the host to keep issuing I/O requests, so that the NVMe queues are not depleted by the SSD platforms.

6.1 Performance Analysis

Microbenchmarks. Figure 11 compares the throughput of the five SSD platforms with I/O sizes varying from 4KB to 32KB. Overall, `ManyLayered` outperforms `750SSD` and `4600SSD` by 1.5× and 45%, on average, respectively. This is because `ManyLayered` can partially take the benefits of manycore computing and parallelize I/O processing across multiple queues and channels over the static resource partitioning. `BaseDeepFlash` exhibits poor performance in cases that the request size is smaller than 24KB with random patterns. This is because threads in NVMQ/ILOCK keep tight inter-thread communications to appropriately control the consistency over locks. However, for large requests (32KB), `BaseDeepFlash` exhibits good performance close to `ManyLayered`, as multiple pages in large requests can be merged to acquire one range lock, which reduces the communication (compared to smaller request sizes), and thus, it achieves higher bandwidth.

We observe that `ManyLayered` and `BaseDeepFlash` have a significant performance degradation in random reads and random writes (cf. Figures 11b and 11d). `DeepFlash`, in contrast, provides more than 1MIOPS in all types of I/O requests; 4.8 GB/s and 4.5 GB/s bandwidth for reads and writes, respectively. While those many-core approaches suffer from many core/flash-level conflicts (`ManyLayered`) and lock/sync issues (`BaseDeepFlash`) on the imbalanced random workloads, `DeepFlash` scrambles the LBA space and evenly distributes all the random I/O requests to different TRANS threads with a low overhead. In addition, it applies cache bypass and lock forwarding techniques to mitigate the long stalls, imposed by lock inquiry and inter-thread communication. This can enable more threads to serve I/O requests in parallel.

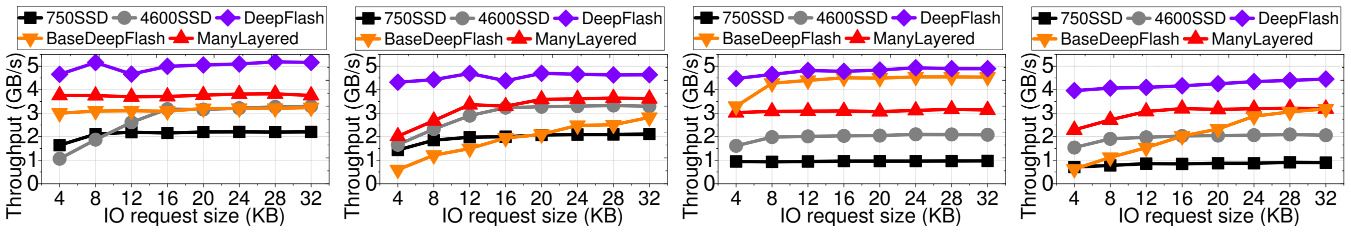
As shown in Figure 12, `DeepFlash` can mostly activate 6.3 cores that run 25 threads to process I/O services in parallel, which is better than `BaseDeepFlash` by 127% and 63% for reads and writes, respectively. Note that, for the random writes, the bandwidth of `DeepFlash` is sustainable (4.2 GB/s) by activating only 4.5 cores (18 threads). This is because although many cores contend to acquire ILOCK which makes more cores stay in idle, the burst buffer successfully overcomes the long write latency of the flash.

Figure 12e shows the active core decomposition of `DeepFlash`. As shown in the figure, reads require 23% more

Host	
CPU/mem	Xeon 16-core processor/256GB, DDR4
Storage platform/firmware	
Controller	Xeon-phi, 12 cores by default
FTL/buffer	hybrid, n:m=1:8, 1 GB/512 MB
Flash array	16 channels/16 pkgs per channel/1k blocks per die 512GB(SLC),1TB(MLC),1.5TB(TLC)
SLC	R: 25us, W: 300us, E: 2ms, Max: 1.4 MIOPS
MLC	R: 53us, W: 0.9ms, E: 2.3ms, Max: 1.3 MIOPS
TLC	R: 78us, W: 2.6ms, E: 2.3ms, Max: 1.1 MIOPS

Workloadsets	Microsoft,Production Server							FIU IODedup		
Workloads	24HR	24HRS	BS	CFS	DADS	DAP	DDR	cheetah	homes	webonline
Read Ratio	0.06	0.13	0.11	0.82	0.87	0.57	0.9	0.99	0	0
Avg length (KB)	7.5	12.1	26.3	8.6	27.6	63.4	12.2	4	4	4
Randomness	0.3	0.4937	0.87	0.94	0.99	0.38	0.313	0.12	0.14	0.14
Workloadsets	FIU SRCMap								Enterprise	
Workloads	ikki	online	topgun	webmail	casa	webresearch	webusers	madmax	Exchange	
Read Ratio	0	0	0	0	0	0	0	0.002	0.24	
Avg length (KB)	4	4	4	4	4	4	4	4.005	9.2	
Randomness	0.39	0.17	0.14	0.21	0.65	0.11	0.14	0.08	0.84	

Table 1: H/W configurations and Important workload characteristics of the workloads that we tested.



(a) Sequential reads. (b) Random reads. (c) Sequential writes. (d) Random writes.

Figure 11: Performance comparison.

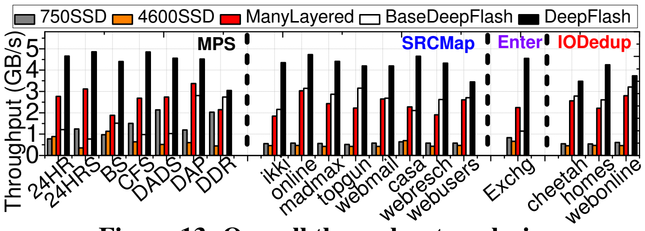


Figure 13: Overall throughput analysis.

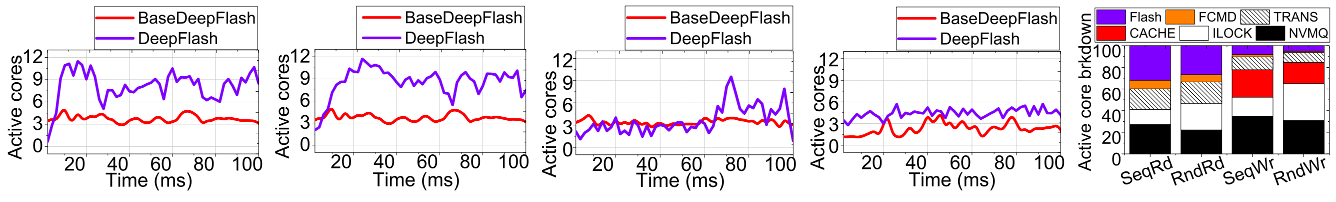
Flash activities, compared to writes, as they are accommodated in internal DRAM. In addition, NVMQ requires 1.5% more compute resources to process write requests than processing read requests, which can offer slightly worse bandwidth on writes, compared to that of reads. Note that background activities such as garbage collection and logging are not invoked during this evaluation as we configured the emulation platform as a pristine SSD.

Server workload traces. Figure 13 illustrates the throughput of server workloads. As shown in the figure, `BaseDeepFlash` exhibits 1.6, 2.7, 1.1, and 2.9 GB/s, on average, for MPS, SRCMap, Enterprise and IODedup workload sets, respectively, and `DeepFlash` improves those of `BaseDeepFlash`, by 260%, 64%, 299% and 35%, respectively. `BaseDeepFlash` exhibits a performance degradation, compared to `ManyLayered` with MPS. This is because MPS generates multiple lock contentions, due to more small-size random accesses than other workloads (c.f. Table 1). Interestingly, while `DeepFlash` outperforms other SSD platforms in most workloads, its performance is not as good under *DDR* workloads (slightly better than `BaseDeepFlash`). This is because FCMD utilization is lower than 56% due to the address patterns of *DDR*. However, since all NVMQ threads parse and fetch incoming requests in parallel, even for such workloads, `DeepFlash` provides 3 GB/s, which is 42% better than `ManyLayered`.

6.2 CPU, Energy and Power Analyses

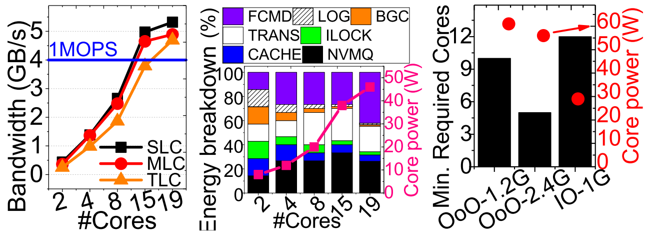
CPU usage and different flashes. Figures 14a and 14b show sensitivity analysis for bandwidth and power/energy, respectively. In this evaluation, we collect and present the performance results of all four microbenchmarks by employing a varying number of cores (2∼19) and different flash technologies (SLC/MLC/TLC). The overall SSD bandwidth starts to saturate from 12 cores (48 hardware threads) for the most case. Since TLC flash exhibits longer latency than SLC/MLC flash, TLC-based SSD requires more cores to reduce the firmware latency such that it can reach 1MIOPS. When we increase the number of threads more, the performance gains start to diminish due to the overhead of exchanging many messages among thread groups. Finally, when 19 cores are employed, SLC, MLC, and TLC achieve the maximum bandwidths that all the underlying flashes aggregately expose, which are 5.3, 4.8, and 4.8 GB/s, respectively.

Power/Energy. Figure 14b shows the energy breakdown of each SSD stage and the total core power. The power and energy are estimated based on an instruction-level energy/power model of Xeon Phi [55]. As shown in Figure 14b, `DeepFlash` with 12 cores consumes 29 W, which can satisfy the power delivery capability of PCIe [51]. Note that while this power con-



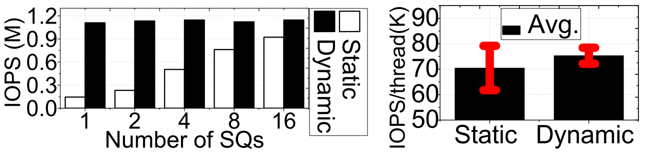
(a) Sequential reads. (b) Random reads. (c) Sequential writes. (d) Random writes. (e) Core decomposition.

Figure 12: Dynamics of active cores for parallel I/O processing.



(a) Bandwidth. (b) Breakdown. (c) Cores.

Figure 14: Resource requirement analysis.



(a) NVMQ performance. (b) IOPS per NVMQ thread.

Figure 15: Performance on different queue allocations.

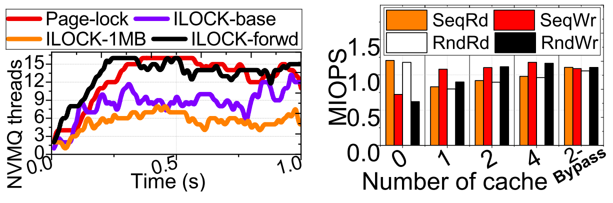
sumption is higher than existing SSDs (20 ~ 30W [30,58,73]), power-efficient manycores [68] can be used to reduce the power of our prototype. When we break down energy consumed by each stage, FCMD, TRANS and NVMQ consume 42%, 21%, and 26% of total energy, respectively, as the number of threads increases. This is because while CACHE, LOG, ILOCK, and BGC require more computing power, most cores should be assigned to handle a large flash complex, many queues and frequent address translation for better scalability.

Different CPUs. Figure 14c compares the minimum number of cores that DeepFlash requires to achieve 1MIOPS for both reads and writes. We evaluate different CPU technologies: i) OoO-1.2G, ii) OoO-2.4G and iii) IO-1G. While IO-1G uses the default in-order pipeline 1GHz core that our emulation platform employs, OoO-1.2G and OoO-2.4G employ Intel Xeon CPU, an out-of-order execution processor [24] with 1.2 and 2.4GHz CPU frequency, respectively. One can observe from the figure that a dozen of cores that DeepFlash uses can be reduced to five high-frequency cores (cf. OoO-2.4G). However, due to the complicated core logic (e.g., reorder buffer), OoO-1.2G and OoO-2.4G consume 93% and 110% more power than IO-1G to achieve the same level of IOPS.

6.3 Performance Analysis of Optimization

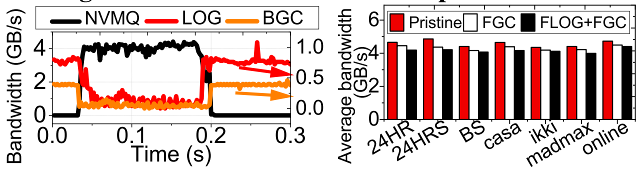
In this analysis, we examine different design choices of the components in `DeepFlash` and evaluate their performance impact on our proposed SSD platform. The following experiments use the configuration of `DeepFlash` by default.

NVMQ. Figures 15a and 15b compare NVMQ's IOPS and per-thread IOPS, delivered by a non-optimized queue allocation (i.e., `Static`) and our DIOS (i.e., `Dynamic`), respectively. `Dynamic` achieves the bandwidth goal, irrespective of the number of NVMe queues that the host manages, whereas `Static` requires more than 16 NVMe queues to achieve 1MIOPS (cf. Figure 15a). This implies that the host also requires more cores since the NVMe allocates a queue per host



(a) ILOCK impact. (b) CACHE IOPS.

Figure 16: ILOCK and CACHE optimizations.



(a) LOG/BGC. (b) BGC overhead.

Figure 17: Background task optimizations.

CPU core [10]. Furthermore, the per-thread IOPS of `Dynamic` (with 16 queues) is better than `Static` by 6.9% (cf. Figure 15b). This is because `Dynamic` can fully utilize all NVMQ threads when the loads of different queues are unbalanced; the NVMQ performance variation of `Dynamic` (between min and max) is only 12%, whereas that of `Static` is 48%.

ILOCK. Figure 16a compares the different locking systems. `Page-lock` is a page-granular lock, while `ILOCK-base` is ILOCK that has no ownership forwarding. `ILOCK-forwd` is the one that `DeepFlash` employs. While `ILOCK-base` and `ILOCK-forwd` use a same granular locking (256KB), `ILOCK-1MB` employs 1MB for its lock range but has no forwarding. `Page-lock` can activate NVMQ threads more than `ILOCK-1MB` by 82% (Figure 16a). However, due to the overheads imposed by frequent lock node operations and RB tree management, the average lock inquiry latency of `Page-lock` is as high as 10 us, which is 11× longer than that of `ILOCK-forwd`. In contrast, `ILOCK-forwd` can activate the similar number of NVMQ threads as `Page-lock`, and exhibits 0.93 us average lock inquiry latency.

CACHE. Figure 16b illustrates CACHE performance with multiple threads varying from 0 to 4. "2-Bypass" employs the bypass technique (with only 2 threads). Overall, the read performance (even with no-cache) is close to 1MIOPS, thanks to massive parallelism in back-end stages. However, write performance with no-cache is only around 0.65 MIOPS, on average. By enabling a single CACHE thread to buffer data in SSD internal DRAM rather than underlying flash media, write bandwidth increases by 62%, compared to the system of no-cache. But single CACHE thread reduces read bandwidth by 25%, on average, due to communication overheads (between CACHE and NVMQ) for each I/O service. Even with more CACHE threads, performance gains diminish due to communication overhead. In contrast, `DeepFlash`'s 2-Bypass can be ideal as it requires fewer threads to achieve 1MIOPS.

Background activities. Figure 17a shows how `DeepFlash` coordinates NVMQ, LOG and BGC threads to avoid contentions on flash resources and maximize SSD performance.

As shown in the figure, when NVMQ actively parses and fetches data (between 0.04 and 0.2 s), LOG stops draining the data from internal DRAM to flash, since TRANS needs to access their meta information as a response of NVMQ's queue processing. Similarly, BGC also suspends the block reclaiming since data migration (associated to the reclaim) may cause flash-level contentions, thereby interfering NVMQ's activities. As `DeepFlash` can minimize the impact from LOG and BGC, the I/O access bandwidth stays above 4 GB/s. Once NVMQ is in idle, LOG and BGC reactivate their work.

STEADY-STATE performance. Figure 17b shows the impact of on-demand garbage collection (`FGC`) and journalling (`FLOG`) on the performance of `DeepFlash`. The results are compared to the ideal performance of `DeepFlash` (`Pristine`), which has no GC and LOG activities. Compared to `Pristine`, the performance of `FGC` degrades by 5.4%, while `FLOG+FGC` decreases the throughput by 8.8%, on average. The reason why there is negligible performance loss is that on-demand GC only blocks single TRANS thread that manages the reclaimed flash block, while the remaining TRANS threads keep serving the I/O requests. In the meantime, LOG works in parallel with TRANS, but consumes the usage of FCMD to dump data.

7 Related Work and Discussion

OS optimizations. To achieve higher IOPS, host-level optimization on multicore systems [8, 36, 75] have been studied. Bjorling *et al.* changes Linux block layer in OS and achieves 1MIOPS on the high NUMA-factor processor systems [8]. Zheng *et al.* redesigns buffer cache on file systems and reinvent overhead and lock-contention in a 32-core NUMA machine to achieve 1MIOPS [75]. All these systems exploit heavy manycore processors on the host and buffer data atop SSDs to achieve higher bandwidth.

Industry trend. To the best of our knowledge, while there are no manycore SSD studies in literature, industry already begun to explore manycore based SSDs. Even though they do not publish the actual device in publicly available market, there are several devices that partially target to 1MIOPS. For example, FADU is reported to offer around 1MIOPS (only for sequential reads with prefetching) and 539K IOPS (for writes) [20]; Samsung PM1725 offers 1MIOPS (for reads) and 120K IOPS (for writes). Unfortunately, there are no information regarding all industry SSD prototypes and devices in terms of hardware and software architectures. We believe that future architecture requires brand-new flash firmware for scalable I/O processing to reach 1MIOPS.

Host-side FTL. LightNVM [9], including CNEX solution [1], aims to achieve high performance ($\sim$1MIOPS) by moving FTL to the host and optimizing user-level and host-side software stack. But their performance are achieved by evaluating only specific operations (like reads or sequential accesses). In contrast, `DeepFlash` reconstructs device-level software/hardware with an in-depth analysis and offers 1MIOPS for all microbenchmarks (read, write, sequential and random) with varying I/O sizes. In addition, our solution is orthogonal to (and still necessary for) host-side optimizations.

Emulation. There is unfortunately no open hardware platform, employing multiple cores and flash packages. For example, OpenSSD has two cores [59], and Dell/EMC's Open-channel SSD (only opens to a small and verified community) also employs 4$\sim$8 NXP cores on a few flash [17]. Although this is an emulation study, we respected all real NVMe/ONFi protocols and timing constraints for SSD and flash, and the functionality and performance of flexible firmware are demonstrated by a real lightweight many-core system.

Scale-out vs. scale-up options. A set of prior work proposes to architect the SSD as the RAID0-like scale-out option. For example, Amfeltec introduces an M.2-to-PCIe carrier card, which can include four M.2 NVMe SSDs as the RAID0-like scale-up solution [5]. However, this solution only offers 340K IOPS due to the limited computing power. Recently, CircuitBlvd overcomes such limitation by putting eight carrier cards into a storage box [6]. Unfortunately, this scale-out option also requires two extra E5-2690v2 CPUs (3.6GHz 20 cores) with seven PCIe switches, which consumes more than 450W. In addition, these scale-out solutions suffer from serving small-sized requests with a random access-pattern (less than 2GB/sec) owing to frequent interrupt handling and I/O request coordination mechanisms. In contrast, `DeepFlash`, as an SSD scale-up solution, can achieve promising performance of random accesses by eliminating the overhead imposed by such RAID0 design. In addition, compared to the scale-out options, `DeepFlash` employs fewer CPU cores to execute only SSD firmware, which in turn reduces the power consumption.

8 Conclusion

In this work, we designed scalable flash firmware inspired by parallel data analysis systems, which can extract the maximum performance of the underlying flash memory complex by concurrently executing multiple firmware components within a single device. Our emulation prototype on a manycore-integrated accelerator reveals that it simultaneously processes beyond 1MIOPS, while successfully hiding long latency imposed by internal flash media.

9 Acknowledgement

The authors thank Keith Smith for shepherding their paper. This research is mainly supported by NRF 2016R1C182015312, MemRay grant (G01190170) and KAIST start-up package (G01190015). J. Zhang and M. Kwon equally contribute to the work. Myoungsoo Jung is the corresponding author.

References

[1] CNEX Labs. https://www.cnexlabs.com.

[2] Microsoft SGL Description. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-scatter-gather-dma.

[3] Nvm express. http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf.

[4] Ultra-low Latency with Samsung Z-NAND SSD. http://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf, 2017.

[5] Squid carrier board family pci express gen 3 carrier board for 4 m.2 pcie ssd modules. https://amfeltec.com/pci-express-gen-3-carrier-board-for-m-2-ssd/, 2018.

[6] Cinabro platform v1. https://www.circuitblvd.com/post/cinabro-platform-v1, 2019.

[7] Jasmin Ajanovic. PCI express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, page 22. ACM, 2013.

[9] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *FAST*, pages 359–374, 2017.

[10] Keith Busch. Linux NVMe driver. *https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Busch.pdf*, 2013.

[11] Adrian M Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[12] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices*, 44(3):217–228, 2009.

[13] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. Parallelizing garbage collection with i/o to improve flash resource utilization. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 243–254, 2018.

[14] Wonil Choi, Jie Zhang, Shuwen Gao, Jaesoo Lee, Myoungsoo Jung, and Mahmut Kandemir. An in-depth study of next generation interface for emerging non-volatile memories. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, pages 1–6. IEEE, 2016.

[15] cnet. Samsung 850 Pro SSD review. *https://www.cnet.com/products/samsung-ssd-850-pro/*, 2015.

[16] Danny Cobb and Amber Huffman. NVM Express and the PCI Express SSD revolution. In *Intel Developer Forum. Santa Clara, CA, USA: Intel*, 2012.

[17] Jae Do. SoftFlash: Programmable storage in future data centers. *https://www.snia.org/sites/default/files/SDC/2017/presentations/Storage_Architecture/Do_Jae_Young_SoftFlash_Programmable_Storage_in_Future_Data_Centers.pdf*, 2017.

[18] Alejandro Duran and Michael Klemm. The Intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE, 2012.

[19] FreeBSD. Freebsd manual pages: flock. *https://www.freebsd.org/cgi/man.cgi?query=flock&sektion=2*, 2011.

[20] Anthony Garreffa. Fadu unveils world's fastest SSD, capable of 5gb/sec. *http://tiny.cc/eyzdcz*, 2016.

[21] Arthur Griffith. *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[22] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[23] Amber Huffman. NVM Express, revision 1.0 c. *Intel Corporation*, 2012.

[24] Intel. Intel Xeon Processor E5 2620 v3. *http://tiny.cc/a1zdcz*, 2014.

[25] Intel. Intel SSD 750 series. *http://tiny.cc/qyzdcz*, 2015.

[26] Intel. Intel SSD DC P4600 Series. http://tiny.cc/dzzdcz, 2018.

[27] Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. A triple core lock-step (TCLS) ARM® Cortex®-R5 processor for safety-critical and ultra-reliable applications. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 246–249. IEEE, 2016.

[28] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[29] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 61–74, 2014.

[30] Myoungsoo Jung. Exploring design challenges in getting solid state drives closer to cpu. *IEEE Transactions on Computers*, 65(4):1103–1115, 2016.

[31] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[32] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 203–216. ACM, 2013.

[33] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535. IEEE, 2014.

[34] Myoungsoo Jung, Ellis H Wilson III, and Mahmut Kandemir. Physically addressed queueing (PAQ): improving parallelism in solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 404–415. IEEE Computer Society, 2012.

[35] Bruce Worthington Qi Zhang Kavalanekar, Swaroop and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *IISWC*, 2008.

[36] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing array of ssds when the storage device is no longer the performance bottleneck. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[37] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

[38] Nathan Kirsch. Phison E12 high-performance SSD controller. http://tiny.cc/91zdcz, 2018.

[39] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds. *arXiv preprint arXiv:1912.06998*, 2019.

[40] Ricardo Koller et al. I/O deduplication: Utilizing content similarity to improve I/O performance. *TOS*, 2010.

[41] Linux. Mandatory file locking for the linux operating system. https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt, 2007.

[42] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[43] marvell. Marvell 88ss1093 flash memory controller. https://www.marvell.com/storage/assets/Marvell-88SS1093-0307-2017.pdf, 2017.

[44] Micron. Mt29f2g08aabwp/mt29f2g16aabwp NAND flash datasheet. 2004.

[45] Micron. Mt29f256g08cjaaa/mt29f256g08cjaab NAND flash datasheet. 2008.

[46] Micron. Mt29f1ht08emcbbj4-37:b/mt29f1ht08emhbbj4-3r:b NAND flash datasheet. 2016.

[47] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H Noh. Enabling cost-effective flash based caching with an array of commodity ssds. In *Proceedings of the 16th Annual Middleware Conference*, pages 63–74. ACM, 2015.

[48] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.

[49] Seon-yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based SSDs. *IEEE Computer Architecture Letters*, 9(1):9–12, 2010.

[50] Chris Ramseyer. Seagate SandForce SF3500 client SSD controller detailed. http://tiny.cc/f2zdcz, 2015.

[51] Tim Schiesser. Correction: PCIe 4.0 won't support up to 300 watts of slot power. http://tiny.cc/52zdcz, 2017.

[52] Windows SDK. Lockfileex function. https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-lockfileex, 2018.

[53] Hynix Semiconductor et al. Open NAND flash interface specification. *Technical Report ONFI*, 2006.

[54] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 561–572. IEEE, 2016.

[55] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 389–394. IEEE, 2013.

[56] Mustafa M Shihab, Jie Zhang, Myoungsoo Jung, and Mahmut Kandemir. Revenand: A fast-drift-aware resilient 3d nand flash design. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):1–26, 2018.

[57] 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. ACM, 2009.

[58] S Shin and D Shin. Power analysis for flash memory SSD. *Work-shop for Operating System Support for Non-Volatile RAM (NVRAMOS 2010 Spring)(Jeju, Korea, April 2010)*, 2010.

[59] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.

[60] Wei Tan, Liana Fong, and Yanbin Liu. Effectiveness assessment of solid-state drive used in big data services. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 393–400. IEEE, 2014.

[61] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[62] Linus Torvalds. Linux kernel repo. https://github.com/torvalds/linux, 2017.

[63] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: Energy proportional storage using dynamic consolidation. In *FAST*, volume 10, pages 267–280, 2010.

[64] Shunzhuo Wang, Fei Wu, Zhonghai Lu, You Zhou, Qin Xiong, Meng Zhang, and Changsheng Xie. Lifetime adaptive ecc in nand flash page management. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1253–1556. IEEE, 2017.

[65] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. WAFTL: A workload adaptive flash translation layer with data partition. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12. IEEE, 2011.

[66] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ANViL: Advanced virtualization for modern non-volatile memory devices. In *FAST*, pages 111–118, 2015.

[67] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[68] Norbert Werner, Guillermo Payá-Vayá, and Holger Blume. Case study: Using the xtensa lx4 configurable processor for hearing aid applications. *Proceedings of the ICT. OPEN*, 2013.

[69] ONFI Workgroup. Open NAND flash interface specification revision 3.0. *ONFI Workgroup, Published Mar*, 15:288, 2011.

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

[71] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.

[72] Jie Zhang, Gieseo Park, Mustafa M Shihab, David Donofrio, John Shalf, and Myoungsoo Jung. OpenNVM: An open-sourced fpga-based nvm controller for low level memory characterization. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 666–673. IEEE, 2015.

[73] Jie Zhang, Mustafa Shihab, and Myoungsoo Jung. Power, energy, and thermal considerations in SSD-based I/O acceleration. In *HotStorage*, 2014.

[74] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *FAST*, pages 59–72, 2013.

[75] Da Zheng, Randal Burns, and Alexander S Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 69. ACM, 2013.

[76] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. An efficient page-level FTL to optimize address translation in flash memory. In *Proceedings of the Tenth European Conference on Computer Systems*, page 12. ACM, 2015.