2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Amber*: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources

Donghyun Gouk[1], Miryeong Kwon[1], Jie Zhang[1], Sungjoon Koh[1], Wonil Choi[1,2],
Nam Sung Kim[3], Mahmut Kandemir[2] and Myoungsoo Jung[1]
Yonsei University,
[1]*Computer Architecture and Memory Systems Lab,*
[2]*Pennsylvania State University,* [3]*University of Illinois Urbana-Champaign,*
http://camelab.org

***Abstract*—SSDs become a major storage component in modern memory hierarchies, and SSD research demands exploring future simulation-based studies by integrating SSD subsystems into a full-system environment. However, several challenges exist to model SSDs under a full-system simulations; SSDs are composed upon their own complete system and architecture, which employ all necessary hardware, such as CPUs, DRAM and interconnect network. Employing the hardware components, SSDs also require to have multiple device controllers, internal caches and software modules that respect a wide spectrum of storage interfaces and protocols. These SSD hardware and software are all necessary to incarnate storage subsystems under full-system environment, which can operate in parallel with the host system.**

In this work, we introduce a new SSD simulation framework, SimpleSSD 2.0, namely *Amber*, that models embedded CPU cores, DRAMs, and various flash technologies (within an SSD), and operate under the full system simulation environment by enabling a data transfer emulation. Amber also includes full firmware stack, including DRAM cache logic, flash firmware, such as FTL and HIL, and obey diverse standard protocols by revising the host DMA engines and system buses of a popular full system simulator's all functional and timing CPU models (gem5). The proposed simulator can capture the details of dynamic performance and power of embedded cores, DRAMs, firmware and flash under the executions of various OS systems and hardware platforms. Using Amber, we characterize several system-level challenges by simulating different types of full-systems, such as mobile devices and general-purpose computers, and offer comprehensive analyses by comparing passive storage and active storage architectures.

***Index Terms*—Full-system simulator, solid state drive, non-volatile memory, memory system, storage, flash memory**

I. Introduction

Non-volatile memories (NVMs) and flash-based storage are widely used in many computing domains, and are changing the role of storage subsystems in modern memory/storage systems. For example, solid state drives (SSDs) have already replaced most conventional spinning disks in handhelds and general computing devices [1], [2]. Further, servers and high performance computers leverage SSDs as a cache or as a burst buffer in hiding long latencies imposed by the underlying disks [3]–[5], or placing hot data for satisfying quality of service (QoS) and service level agreement (SLA) constraints [6], [7].

While most of the time simulation-based studies are necessary to explore a full design space by taking into account different SSD technologies, it is non-trivial to put SSD-based subsystems into a full-system simulation environment. First, even though SSDs are considered as storage or memory subsystems, in contrast to conventional memory technologies, they are in practice incarnated on top of a complete, independent, and complex system that has its own computer architecture and system organization. Second, SSDs consist of not only a storage backend (by having multiple flash packages through internal system buses), but also a computation complex that employs embedded CPU cores, DRAM modules, and memory controllers. As all these hardware components operate in parallel with the host, the simulated evaluations can be easily far from the actual performance. Third, SSDs serve I/O requests and communicate with the host system through different types of storage interfaces, including serial ATA (SATA) [8], universial flash storage [9], NVM Express (NVMe) [10], and open-channel SSD (OCSSD) [11]. Since SSDs are subservient to the host system and OS decisions, SSD-based subsystem or full-system simulations without considering these storage interfaces and software stacks can result in ill-tuned and overly-simplified evaluations. Lastly, SSDs also employ different firmware modules optimized to reduce the performance disparity between the host interface and the internal storage complex, which also have a great impact on user-level system behavior, based on different workload executions on full-system environment.

Unfortunately, most SSD simulation infrastructures available today have no full software stack simulation through emulation of data transfers and they also lack SSD internal hardware and/or software resource models, which makes them insufficient to be put into a full-system environment. For example, [12], which is widely used for SSD simulations in both academia and industry, only captures the functionality of a specific flash firmware, called flash translation layer (FTL) without modeling any hardware resource in an SSD. Similarly, most recent simulators [13]–[15] have no model for computation complex and therefore, no detailed timing simulation for firmware execution can be carried out. Further, to the best of our knowledge, there exists no simulation model that can be integrated into a full-system environment by implementing all actual storage interfaces and data transfer emulations. For example, [15] only mimics a pointer-based multi-queue protocol (in storage) without modeling system buses and

*Amber is the project name for SimpleSSD 2.0. All the sources of our simulation framework are available to download from https://simplessd.org.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00045

data movements. As a result, it cannot be integrated into a full system setting that requires the emulating a real OS and all relevant software/hardware components. Lastly, none of the existing SSD simulators can be attached to different storage interfaces by respecting *both* functional and timing CPU modeled full-system infrastructures. For example, [13] and [14] enable a full-system simulation, but they only work on a functional CPU, which overly simplifies the host memory subsystem and CPU execution timings.

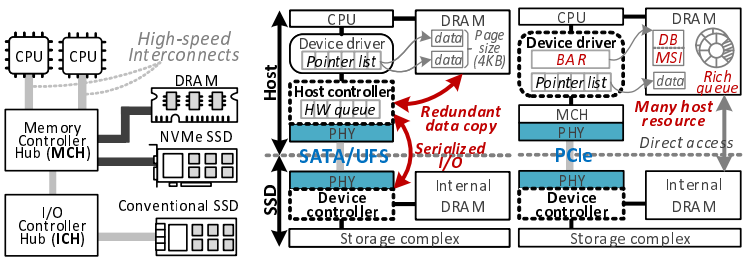
Motivated by the lack of full system simulation tools that can accommodate SSDs with all functional and timing parameters as well as firmware, we introduce a new SSD simulation framework, SimpleSSD 2.0, namely *Amber*, which accommodates all SSD resources in a full system environment and emulates all software stacks employed by both functional and timing CPU models. The proposed simulation framework modifies host-side system buses, and implements device-side controllers and a DMA engine to emulate data transfers by considering a wide spectrum of storage interfaces and protocols, such as SATA, UFS, NVMe, and OCSSD. In addition, Amber implements a diverse set of firmware modules on top of detailed SSDs computation/storage complex models, and applies different flash optimizations such as parallelism-aware readahead [16] and partial data update schemes [17], which allows it to easily mimic the performance characteristics of real systems. While most existing SSD simulators evaluate the performance of an underlying storage complex by replaying block-level traces, we evaluate/validate Amber by actually executing different microbenchmarks as well as user-level applications on real OS-enabled systems.

To the best of our knowledge, *Amber* is the first SSD simulation framework, that incorporates *both* computation and storage complexes within an SSD and covers a large and diverse set of storage interface protocols and data transfer emulation, while considering a full-system storage stack. The main contributions of this work can be summarized as follows:
• *Hardware and software co-simulation for storage.* For SSD's computation complex, we model and integrate embedded CPU cores, DRAM modules/controller and system buses, which can capture the detailed latencies and throughput of the execution of flash firmware components based on ARMv8 ISA. Amber's storage complex implements reconfigurable interconnection networks that can simulate a wide spectrum of flash technologies by considering detailed flash transaction timing models. We integrate a full firmware stack on top of the computation and storage complexes, thereby capturing diverse SSD functionalities, such as I/O caching, garbage collection, wear-leveling, and address translation. This co-simulation framework offers realistic storage latency and throughput values by incorporating all *SSD* resources into a full-system. Amber can also be used to explore various dynamics of an SSD, which are critical for monitoring power/energy usages of different storage components under real application execution scenarios.
• *Enabling SSDs in diverse full-system domains.* Real systems can attach an SSD over different locations based on their platform and user demands. We enable both I/O controller hub's hardware-driven storage (e.g., SATA and UFS) and memory controller hub's software-driven storage (e.g., NVMe and OCSSD) by implementing all their mandatory commands and data transfer mechanisms. To this end, we integrate our hardware/software co-simulation storage models into gem5 [18], and revise gem5 for tight integration with data transfer emulation. Amber modifies gem5's system bar and models a host-side DMA engine that moves data between the OS system memory and the underlying storage. This work also includes a set of protocol-specific implementations such as host/device controllers, system memory references over a pointer list, and a queue arbitration logic. Amber works with all of gem5's CPU types such as timing and functional CPU models for both in-order and out-of-order executions under different OS versions.
• *Holistic analysis for different storage subsystem designs.* Since Amber can emulate data transfers and execute a full storage stack, including host's system software and device's firmware, We implement a diverse set of SSD subsystems in Amber, and analyze their system-level challenges. Specifically, we characterize i) the performance impact of employing different operating systems, ii) mobile system challenges regarding UFS/NVMe, and iii) passive and active storage architectures. We observe that OS-level ill-tuned I/O scheduling and queue management schemes can significantly degrade the overall system performance, and that user applications in mobile computing should be revised by considering high performance SSD designs. Specifically, even though NVMe (attached ARM core) exhibits much better performance than UFS, some applications cannot take advantage of NVMe due to their default I/O operation mode. Our evaluations also reveal that the CPU and memory utilization can be problematic issues that a passive storage architecture (employing host-side FTL over OCSSD) needs to address in the future. In particular, while the active storage (having NVMe controller and full firmware stack in an SSD) consumes only 7% of the host-side CPU, the passive storage consumes most of host-side resources.

II. Background

A. System Architecture

Overview. SSDs can be attached to modern systems by memory controller hub (MCH) and/or I/O controller hub (ICH) via PCI Express (PCIe) or serial I/O ports, respectively. As shown in Figure 1a, MCH is directly connected to CPU via a front-side bus, and usually manages high-speed I/O components, such as DRAM, GPGPU and NVMe SSD devices. In contrast, ICH is paired with MCH, but handles relatively slow devices, such as spinning disks and conventional SSDs. Due to the different purposes of I/O connection mechanisms of MCH and ICH, in practice, SSDs and their interfaces can classified into two types of storage subsystems: i) *hardware-driven storage* (h-type) and ii) *software-driven storage* (s-type). h-type storage includes serial ATA (SATA) and universal flash storage (UFS) SSDs, whereas NVM Express SSDs (NVMe) and open-channel SSDs (OCSSD) operate as s-type storage. Overall, h-type storage is more efficient than s-type storage in terms of both I/O management efficiency and host-side resource utilization, but its latency and bandwidth are somewhat limited, preventing it from taking full advantage of the underlying flash media, due to frequent hardware interventions and queue/interrupt management complexities.



(a) Overview. (b) H-type. (c) S-type.

Fig. 1: System architecture.

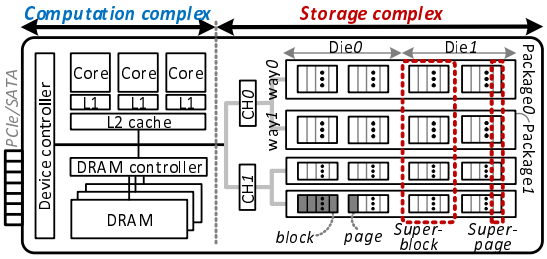


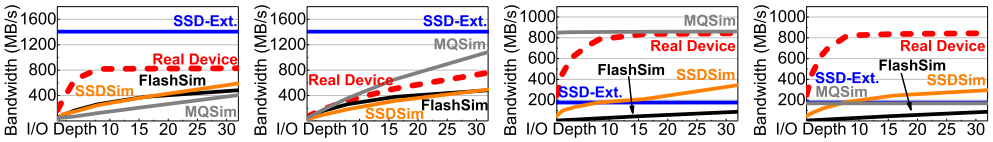
Fig. 2: SSD internal architecture.

H-type storage. As shown in Figure 1b, this type of storage employs both a "host" controller and a "device" controller for data communication over the SATA or UFS interface protocols. The host system's CPU and device drivers can issue an I/O request to the underlying storage and manage interrupt service routine (ISR) only through the host controller. This host controller resides in ICH, and unpacks/packs multiple commands and payload data by using its own buffer. The host controller sends/retrieves data to/from the device controller that exists within an SSD over its physical layer (PHY) on behalf of the host's OS drivers. There are two main architectural challenges behind modeling h-type storage. First, the data transfers between the host and underlying storage require multiple data copies since CPU only configures a set of registers to issue an I/O request. Specifically, for data communication, the drivers (running on CPU) construct a *pointer list* (each entry indicating the target system memory (DRAM) page), rather than transferring contents between the host and the storage. Therefore, the host controller should traverse all memory pointers and copy each system memory page to its own buffer. During this phase, the host controller issues a request by communicating with the target device controller through different PHYs. Second, since only the host controller manages data movements, the number I/O requests controlled by its hardware queue and interrupt mechanism is comparably small and limited. In addition, all the I/O queue management and interrupt handling of the host should be serialized with the h-type storage architecture. This single I/O path is often considered as a major performance bottleneck in many computing domains [19], [20].

S-type storage. This type of storage has no host controller, and each component of the host is managed by software. Specifically, s-type storage's device controller exposes specific parts of the internal DRAM to the host's designated system memory regions via PCIe, which is referred to as *base line address* (BAR) [21]. The software modules in the host-side storage stack, such as host NVMe or OCSSD drivers, can directly configure such memory-mapped BAR spaces. Since memory-mapped spaces are in parallel visible to the device controller, SSD can also directly pull or push data over the host-side system memory pages. In addition, interrupts in the s-type storage are managed by a request packet, referred to as *message-signaled interrupt* (MSI/MSI-X) [22]. Once the device controller completes an I/O service, it directly writes the interrupt packet into another memory-mapped region (on the host DRAM), called the MSI/MSI-X vector. While the OS driver and device controller can manage I/O request submissions and completions through BARs and MSI vector, it is non-trivial to make their I/O queues consistent and coherent between the host and storage. To address this challenge, s-type storage supports per-queue head and tail pointers, which can be written into doorbell registers by OS drivers but exposed by the device controller. With the head and tail pointer mechanisms in place, the host device driver and device controller can synchronize enqueue and dequeue status for each I/O submission and completion. This data communication method allows s-type storage to employ a large number of queues and queue entries (referred to as *rich queue*), and makes the host eliminate the I/O queue as well as the ISR serialization issues. However, since the software modules are now involved in data transfers, s-type storage requires more host-side resources than h-type storage [23]–[25].

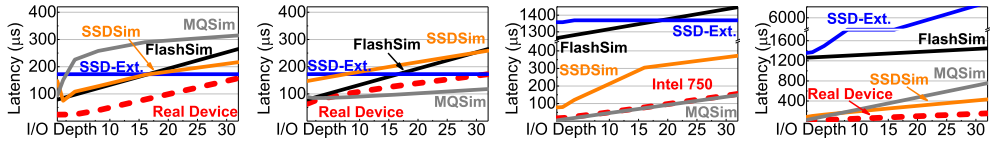
B. SSD Internals

Hardware architecture. Since the performance of flash media is much lower than the bandwidth capacity of host storage interfaces, most modern SSDs are designed based on a multi-channel and multi-way architecture to improve internal parallelism, with the goal of reducing the bandwidth gap between the host interface and flash [26], [27]. As shown in Figure 2, multiple flash packages, each containing multiple dies, are connected to the interconnection buses, referred to as *channel*. The set of packages across different channels can simultaneously operate, and in practice, flash firmware spreads a host request over multiple dies that have the same offset address, but exist across different channels. Each set of flash dies (or packages) is called a *way*. A group of multiple physical pages and blocks that span over all internal channels or ways is referred to as *super-page* or *super-block*, respectively. In this multi-channel and multi-way architecture, the *storage complex* (putting all flash media into the interconnection network) is also connected to a *computation complex* that consists of embedded CPU cores, internal DRAM, and device controller(s). Multiple cores are allocated to flash firmware that controls all I/O services and address translations within an SSD. Since the operating frequency domains between the host and storage are completely different, all data coming from the underlying flash (e.g., reads) or the host-side driver or controller (e.g., writes) should be first buffered in the internal DRAM and should then be moved to the target device. Note that the flash media of the storage complex allows no overwrite, due to the *erase-before-write* characteristic/requirement. Specifically, at the flash-level, writes/reads are served per page, whereas erases should be



(a) Sequential read. (b) Random read. (c) Sequential write. (d) Random write.

Fig. 3: Bandwidth comparison between a real device and existing SSD simulators.



(a) Sequential read. (b) Random read. (c) Sequential write. (d) Random write.

Fig. 4: Latency comparison between a real device and existing SSD simulators.

NAND Flash timing (µs)		
t_{PROG}	820.62	2250
t_R	59.975	104.956
t_{ERASE}	3000	
Storage back-end		
Channel	Package	Die
12	5	1
Plane	Block	Page
2	512	512
Internal DRAM		
Size	Channel	Rank
1GB	1	1
Bank	Chip	Bus width
8	4	8

TABLE I: Hardware configuration of real device.

processed per block. In practice, a flash block contains 128 $\sim$ 512 pages, and the latency of an erase operation is 50 times longer than that of a page write. In addition, the order of writes in a block should be *in-order* to avoid page-to-page interference/disturbance for the multi-level cell (MLC) and triple-level cell (TLC) flash technologies.

Software organization. Due to the aforementioned erase-before-write characteristic, all existing flash-based storage today require employing a firmware that makes it compatible with the conventional block storage and hides the complexity of flash management. *Flash Translation Layer* (FTL) is a key component of the flash firmware that prepares a number of blocks, which are erased in advance (called *reserved block*). FTL writes data into a reserved block by mapping the input address (*logical block address*, LBA) to *physical page address/number* (PPA/PPN). In cases where there is no reserved block at runtime, FTL migrates valid physical pages from old block(s) to a new block, erases the old block, and remaps the addresses of the migrated pages, thereby securing available block(s) to forward incoming write requests. This process is called *garbage collection* (GC). Since the number of erases per block is limited due to flash wear-out issues [28], [29], when FTL performs a GC task, it tries to erase flash blocks in an evenly distributed manner, referred to as *wear-leveling*. In addition, flash firmware also implements request parsing, protocol management, I/O scheduling, and data caching. Specifically, the device controller that exists for both h-type storage and s-type storage, manages the data communication based on the interface/protocol defined by the host. Then, host interface layer (HIL) transfers data and performs I/O scheduling atop other flash firmware modules. Since the data is buffered in the internal DRAM in an SSD, the flash firmware can also cache the data to leverage DRAM performance [30], [31]. This in turn can help us hide the long latency imposed the underlying flash media.

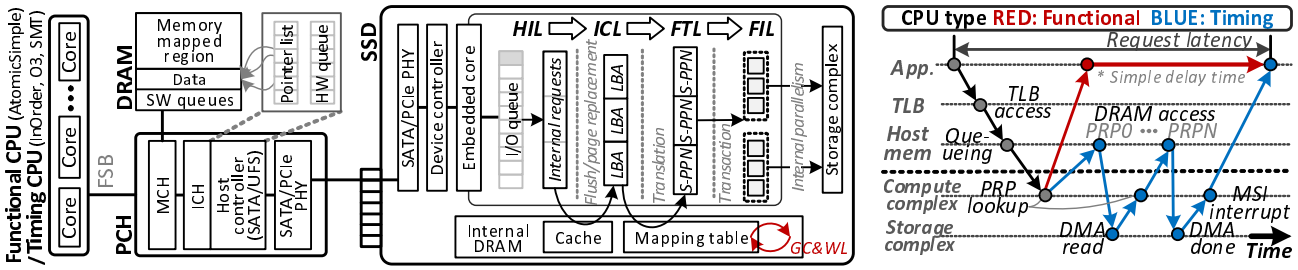
III. Enabling Hardware/Software Co-Simulation

A. Challenges in Capturing Realistic SSD Performance

Figures 3 and 4 compare SSD bandwidth and latency of a real device (Intel 750) and existing SSD simulators, namely, MQSim [15], SSDSim [32], SSD Extension for DiskSim (SSD-Extension) [12], and FlashSim [33]. To perform this analysis, we disassemble an Intel 750 device and reverse-engineer the number of channels, ways, flash dies, and DRAM sizes. We then check the part number of flash and also extract the low-level flash latencies for the read, write and erase operations from datasheets. We then configure all simulators' device parameters by referring the datailed hardware configuration information given in Table I. We extract 4KB-sized block traces from the flexible I/O tester synthetic benchmark (FIO [34]) and replay the traces on each simulator mentioned above, with I/O depths varying from 1 to 32. Since none of these existing simulators is suitable to execute FIO at user level by having full storage stack over full-system environment, *replaying or generating traces* within their simulation framework is the only way to evaluate their performance with the same device configuration.

Bandwidth trend. One can observe from Figure 3 that, all existing simulators' bandwidths are *far* from the performance of the real device. Specifically, Intel 750 keeps utilizing the bandwidth, and its performance saturates with 8 $\sim$ 16 queue entries (except for random reads). As the number of queue entries increases, most existing simulators exhibit performance curves that are completely different from that of the real device. Specifically, the existing simulators' bandwidth i) linearly increases (MQSim and SSDSim), ii) exhibits just a constant trend (SSD-Extension and FlashSim), or iii) shows a curve but does not saturate (SSDSim). In contrast, the bandwidth trend of the real device is sublinear as the number of queue entries increases. In addition to these trend differences, all the simulators tested exhibit significant performance disparity between their models and the real device. With 16 queue entries (i.e., steady-state), the error in MQSim's reads and writes ranges between 3% and 80%, compared to Intel 750. Other simulators' error ranges are even more serious. More specifically, the resulting errors with SSDSim, SSD-Extension, and FlashSim for reads and writes are as high as 54%, 176%, and 54%, and 75%, 79%, and 94%, respectively.

Latency trend. As shown in Figure 4, the latency bahavior is different from the aforementioned bandwidth trends. As the I/O depth is increased, the latency exhibited by the real device gets worse (due to the queueing delays). But, for *all* random and sequential I/O patterns, its latency is lower than 175 us. While the existing simulators also show an increased latency as the number of queue entries increases, their latencies range between 4 us and 6285 us, and exhibit different performance curves, compared to the real device. The existing simulators with varying I/O depths exhibit i) sublinear-like latency trend (MQSim, SSDSim and SSD-Extension), ii)



(a) Internal architecture.

(b) Changes for gem5 system barbus and DMA.

Fig. 5: High-level view of `Amber`.

constant latency trend (SSD-Extension), and iii) linear trend curved by unrealistic gradients (FlashSim). For example, for sequential writes, MQSim offers a latency behavior similar to that of the real device, and its latency on sequential reads exhibits a sublinear-like behavior when increasing the I/O depth. SSD-Extension shows a sublinear-like latency curve for random writes, but it does not have any latency variance when modulating the number of queue entries under the execution of all other I/O workloads. The latency errors incurred by MQSim, SSDSim, SSD-Extension and FlashSim for reads and writes are as high as 816%, 518%, 627%, and 293%, and 388%, 495%, 8492%, and 7887%, respectively.

We believe that the main reasons why most of the existing simulators exhibit significantly different performance trends and values from the real device, with varying I/O depths, are: i) lack of a computation complex, ii) incomplete firmware stack, iii) omitted storage interface and protocol management, and iv) absence of a host-side I/O initiator (e.g., host driver or controller). In contrast, Amber can simulate both computation and storage complexes, and execute FIO from user-level, employing all hardware and software components on a full-system environment. Overall, Amber exhibits performance trends that are very similar to those of the real device when varying the I/O depth, and further, its user-level latency and bandwidth are different from those observed with the real device by 20% and 14%, on average, respectively. A more detailed analysis will be given later in Section V.

B. High-Level View of Amber

Internal architecture. Figure 5a depicts the internal architecture that Amber models for an SSD. Amber models embedded CPU cores based on ARMv8 instruction set architecture (ISA) [35]; we decompose the instructions of each module's functions in a fine-granular manner, and allocate each firmware component to different CPU core. The function call procedures and control flow of SSD firmware stack can vary based on the workloads and the OS decisions made by the software emulation of higher full-system environment, Amber keeps track of all dynamic call procedures for each run, and measures instruction executions by considering arithmetic instructions, branches, loads/stores, etc. Monitoring instruction-level execution also considers the latency, overlapped with flash operations, thereby capturing very detailed *per-request* and *per-transaction* timing for both synchronous and asynchronous I/O services. In addition, we modify and integrate multicore power and area models [36] into the embedded CPU cores. This integration enables Amber to estimate the dynamic power and energy of firmware execution, based on the specific full-system environment being executed at runtime.

We also model an internal DRAM and its memory controller, which are connected to the CPU cores through the SSD system port. This internal DRAM and memory controller can capture detailed DRAM timing parameters, such as row precharge (*tRP*), row address to column address delay (*tRCD*), and CAS latency (*tCL*), considering all memory references made by the various firmware components. Specifically, this memory model contains cached data, metadata and mapping information during the SSD simulation, which are dynamically updated by flash firmware. To measure the internal power consumption of an SSD, we also incorporate a DRAM power model [37], [38] in our DRAM module and controller. This integrated DRAM power model considers *all* different DDR memory states, including the power-down and self-refresh states, and also takes into account the memory controller's open-page and close-page policies and all levels of bank-interleaving strategies.

As part of the storage complex, we modify and integrate a multi-channel and multi-way architecture [13] that can capture detailed timings, such as programming time (*tPROG*), flash memory island access time (*tR*), as well as latencies for data transfer (*tDMA*) and flash command operations. This integrated design allows Amber to accommodate highly-reconfigurable flash memory and controller models that can accurately mimic a diverse set of state-of-the-art flash technologies such as multi-level cell (MLC), triple-level cell (TLC), etc. Since this physical flash model has no built-in power model, we also modify and add a flash power measurement tool [39] into the storage complex's each flash package. This model can capture the dynamic power and energy consumption of data movements from the internal DRAMs to each package's row buffer (flash registers). In addition, it dynamically measures the actual flash access power consumed to load or store data between the buffer and flash die/plane.

Firmware stack. Figure 5a shows Amber's firmware stack. At the top of this firmware stack, HIL schedules I/O requests based on the queue protocol that the host storage interface defines. For example, HIL of h-type storage performs I/O scheduling based on first-in first-out (FIFO). However, it schedules I/O requests based on two I/O arbitration mechanisms for s-type storage, namely, round-robin (RR) and weighted round-robin (WRR). HIL fetches a host request from the device-level queue by communicating with a device controller that manages storage-side physical layer (PHY) and data movement. HIL then splits the request into multiple page-based internal requests

by considering Amber's cache entry size, which is the same as the size of a super-page. The separated requests are cached into a DRAM model of the computation complex by the underlying *internal cache layer* (ICL). ICL buffers/caches the data from the flash and/or host controller/driver. In cases where the data should be evicted due to a page replacement or flush command (coming from the host-side OS), ICL retrieves the corresponding data from the internal DRAM, composes per-page requests whose address indicates a super-page aligned LBA, and issues it to the underlying FTL. FTL then translates the request's LBA to *super-page basis PPN* (S-PPN). If there is no available reserved block to allocate S-PPN, Amber's FTL performs GC (and wear-leveling) by considering the number of valid pages to migrate (e.g., Greedy [40]) and block's access time (e.g., Cost-benefit [41]). FTL then submits the super-page requests to the underlying *flash interface layer* (FIL), which in turn schedules flash transactions and parallelizes I/O accesses across multiple channels and ways based on a parallelism method [42] that the user defines. Further, we optimize ICL and FTL by being aware of flash internal parallelism to enhance overall performance, as will be explained later in Section IV-C.

Note that all the software modules in Amber's firmware stack are highly *reconfigurable*, so that our simulation model can be incarnated as diverse storage devices under the full-system simulation environment. ICL can be configured as a fully-associative, set-associative, and direct-map cache; the number of ways and sets, cache entry size, and replacement policy (LRU, random, etc.) can all be reconfigured. Similarly, FTL can realize different types of mapping algorithms, such as block-level mapping, various hybrid mapping-algorithms, and pure page-level mapping. The request schedulers of FIL and flash controllers can also capture all possible parallelism combinations by taking into account the internal SSD resources such as a channel, way, die, and plane.

Data transfer emulation. In contrast to the existing SSD simulators that only capture latency or throughput based on timing calculator, Amber needs to handle the actual contents of all I/O requests between the host and storage. This data transfer emulation is necessary to be able to execute an OS and user-level applications on a full-system environment. To this end, we model a DMA engine and integrate it into gem5, which transfers real data from the host's system memory to the internal DRAMs of Amber's SSD architecture model. In practice, the host driver and/or controller for both h-type storage and s-type storage composes a pointer list whose each entry indicates a system memory page, as shown in Figure 5a. The DMA engine that we implement in gem5 parses the pointer list, whose actual structure varies based on the interface protocol that a storage types defines. More detailed discussion on this will be presented later in Section IV. The DMA engine then performs the data transfer from the host DRAM to the underlying storage.

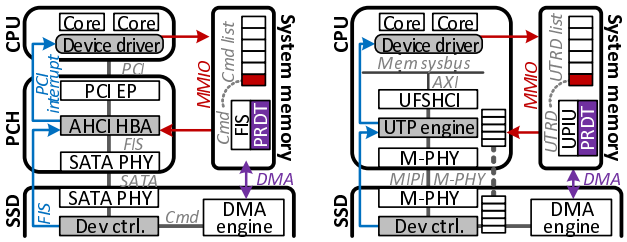
One of the challenges behind this DMA engine implementation in a full system environment is that, the different CPU models in gem5 require different memory access timings and I/O service procedures for the software modules in the storage stack. For example, a functional CPU model (i.e., `AtomicSimpleCPU`) requires to push/pull actual data at the very beginning of an I/O service (per request), but before the execution of any corresponding data communication activities. This is because, such functional CPU employs a simplified DRAM model and does not have any specific timing for OS executions. However, all other timing CPU models, including in-order and out-of-order pipelined executions, use the details of the memory access timing for software emulations. Therefore, whenever the host and device controllers of s-type and h-type storage access the memory over the system bus, the DMA engine should be involved in handling the start and end of memory accesses. We define multiple DMA handlers that service queue (mapped to system memory) accesses, queue entry references, command compositions, and pointer list traversing activities, and register them with the gem5's timing simulation engine. For the content transmission, timing CPU models should be also involved in referring each system memory page access, transferring the page between the host and storage, and performing I/O completion based on MSI/MSI-X. While the functional CPU model just aggregates data transfer activities for each request into a single I/O task (as the pointer list contains many system memory pages to transfer) in timing CPU models, the DMA engine emulates finer-granular data transfers per I/O request by arbitrating the memory references of the device/host controllers and OS drivers as many times as the number of entries that the pointer list has.

IV. Details of Full-System Storage

A. H-type Storage

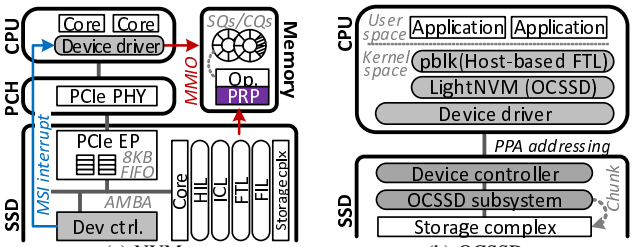
Figures 6a and 6b illustrate our SSD implementations under full system environment for SATA and UFS, respectively. In SATA, the host's functional and timing CPU models are connected to processor controller hub (PCH), which integrates MCH and ICH together via front-side bus, whereas the CPU models directly communicate with the host controller in UFS.

SATA modeling. PCH attaches to the underlying SATA through a PCI endpoint by using PCI configuration spaces [43]. Under the PCI endpoint, a *host block adapter* (HBA) generates a set of commands, pointer lists, queues and the corresponding data payloads. We implement HBA as SATA's host device controller in gem5, based on *advanced host controller interface* (AHCI) specification [44]. This HBA communicates with the OS device driver (e.g., ATA and the SCSI libraries [45], [46]), which exposes the underlying SSD simulation model to all upper-layer OS components as an actual storage volume. Specifically, HBA exposes two major sets of registers, which are mapped to the system memory: one for a *command list* and another for *communication information* [44]. The command list contains 32 entries, which is related to SATA's native command queue (NCQ) management, while the communication information handles data transfer-related packets, referred to as *frame information structure* (FIS). Each command in the command list includes a pointer list that contains system memory page addresses; in SATA, the pointer list is callled *physical region descriptor table* (PRDT) [47]. Through HBA's memory-mapped register sets, OS drivers can manage the order of I/O requests and set all the corresponding FIS and payload data. HBA then fetches a command and FISs from the register sets, and issues the I/O request through SATA PHY. In our implementation, SATA PHY exists for both full system simulator and storage simulator. Thus, the device controller parses the issued SATA



(a) SATA. (b) UFS.

Fig. 6: H-type storage.



(a) NVMe. (b) OCSSD.

Fig. 7: S-type storage.

requests, and pass them to HIL. During this phase, the DMA engine in gem5 retrieves all the target page addresses in the system memory through PRDT and emulates the data transfers, while I/O completions and interrupts (for a host CPU) are managed by both the device controller and HBA.

UFS modeling. Even though UFS employs h-type storage protocol management, which is very similar to the aforementioned SATA, the UFS datapath between the host CPU and the host controller is slightly different from that of SATA. Since UFS is designed for handheld computing, the host controller resides in CPU as an SoC, which is directly connected to the memory system bus. As shown in Figure 6b, we implement the host controller based on *UFS transport protocol* (UTP) [9], referred to as *UTP engine*. UTP engine is connected to the system bus, called *advanced extensible interface* (AXI), instead of PCIe [48]. The host CPU can communicate with the UTP engine through *UFS host controller interface* (UFSHCI) [49]. UTP engine and UFSHCI are functionally equivalent to SATA HBAs and PCI endpoint, respectively. The physical layer of UFS is defined by M-PHY [50], which exists both on the SoC side and on the storage side. To address the issue of different frequency domains between the UTP engine and the underlying device controller, we also implement a small-sized FIFO queue within those two controllers. The queuing and data transfer methods are very similar to SATA's NCQ and PRDT. Also, similar to SATA, the UTP engine exposes a set of registers via memory-mapped address spaces to the host CPU. Thus, OS drivers (e.g., OS and UFS host controller drivers [51]) can issue a command, referred to as *UTP transfer request descriptor* (UTRD) [49], over the command list which has 32 queue FIFO queue entries, similar to SATA/NCQ. Each queue entry contains PRDT and two *UFS protocol information unit* (UPIU) [9] for request and response. Once the I/O request is issued by UTP engine, the device controller, parses all commands, data payloads, and communication information, and then exposes them to HIL. The DMA engine emulates data transfers, and the corresponding I/O completion and interrupt are managed by both the device controller and UTP engine.

B. S-type Storage

Figure 7 shows the s-type storage details that Amber implements in a full-system environment. NVMe and OCSSD employ the PCIe physical interface as opposed to the ICH interface, and simplify the host-side datapath by removing the host controller. PCH's multiple PCIe lanes are connected to a PCIe endpoint (that exists on the SSD-side). We model the endpoint based on Xilinx's FPGA Gen3 Integrated Block for PCIe. Our endpoint employs inbound and outbound 8KB FIFO queues, which convert PCIe PHY to SSD on-chip interconnect system bus, called *advanced microcontroller bus architecture* (AMBA) [52], by modeling a AXI4-stream interconnect core that can connect multiple master/slave AMBA ports.

NVMe modeling. NVMe controller parses the PCIe and NVMe packets as the device-side controller, and exposes the corresponding information to HIL, so that HIL can handle the I/O requests by collaborating with other firmware modules such as ICL, FTL and FIL. We leverage a protocol/queue management module [14], but significantly modify gem5's system barbus, DMA engine. We also implement the device controller to support all mandatory NVMe commands and some optional features (such as NVMe namespace management and a scatter gather list (SGL)) for supporting diverse demands of the host-side software. The communication strategies between NVMe controller and OS driver (e.g., NVMe driver) are functionally similar to h-type storage, but the queue mechanism of NVMe (and OCSSD) is quite different from that of h-type storage. As NVMe removes the host controller, instead of h-type storage's per-controller queue, OS can create 65536 rich queues, each containing up to 65536 entries. Each queue logic is coupled with a *submission queue* (SQ) and a *completion queue* (CQ), which are synchronized between the OS drivers and the NVMe controller. Specifically, OS drivers issue an I/O service, compose 64 bytes NVMe request and put the request into an SQ entry. This 64 bytes request includes an op-code (i.e., NVMe command), metadata (e.g., namespace), and a pointer list. In NVMe, the pointer list is implemented by *physical region page* (PRP) List [10], which can contain more than 512 system memory pages (4KB). In our implementation, we also offer a conventional scatter-gather list (SGL) [53] in cases where a different version of OS drivers needs SGL. When the service is completed, an event of I/O completion is delivered by the NVMe controller to the OS drivers through 16 bytes CQ request. The queue states between the NVMe driver and controller are synchronized over MSI and a set of registers (called *doorbell*) that we implemented. During this time, the DMA engine emulates the transfer of all relevant data (presented by PRP/SGL) and NVMe packets.

OCSSD modeling. Since OCSSD overrides the NVMe interface/protocol to expose physical flash addresses to the host, the datapath and hardware architecture (e.g., PCIe endpoint) are the same as in NVMe. The difference between NVMe and OCSSD is that OCSSD unboxes SSD's storage complex and exposes the low-level flash page addresses to the host software, whereas NVMe still keeps all flash managements on the SSD side as a kind of *active* device. Specifically, OCSSD allows OS and user software to tune the underlying SSD subsystems

	PC platform	Mobile platform
CPU name	Intel i7-4790K	NVIDIA Jetson TX2
ISA	X86	ARM v8
Core number	4	4
Frequency	4.4GHz	2GHz
L1D cache	private, 32KB, 8-way	private, 32KB
L1I cache	private, 32KB, 8-way	private, 48KB
L2 cache	private, 256KB, 8-way	shared, 2MB
L3 cache	shared, 8MB, 16-way	N/A
Memory	DDR4-2400, 2 channel	LPDDR4-3733, 1 channel

TABLE II: Gem5 system configurations.

	Avg. read length (KB)	Avg. write length (KB)	Read ratio (%)	Random read (%)	Random write (%)
Authentication Server (*24HR*)	10.3	8.1	10	97	47
Back End SQL Server (*24HRS*)	106.2	11.7	18	92	43
MSN Storage metadata (*CFS*)	8.7	12.6	74	94	94
MSN Storage FS (*MSNFS*)	10.7	11.2	67	98	98
Display Ads Payload (*DAP*)	62.1	97.2	56	3	84

TABLE III: Workload characteristics.

by employing FTL and ICL on the host side, and it considers the underlying SSD as a *passive* device. To enable OCSSD in a full-system simulation environment, we first implement both OCSSD 1.2 [54] and 2.0 interfaces [55], and enable `pblk` [56] and `lightNVM` [57] modules in gem5's Linux kernel, as a host-side FTL and OCSSD driver, respectively. `LightNVM` is tightly coupled with the existing OS's NVMe driver to handle NVMe-like I/O services, but also handles the overridden OCSSD command and feature management. We also implement an OCSSD subsystem on the SSD-side, which leverages NVMe's device controller and HIL, but supports all the features required for OCSSD specification, such as chunk (similar to physical block) and other information that the host-side OS requires (e.g., memory latency data and erase count information). Since `pblk` manages flash addresses underneath file systems, the OCSSD subsystem in SSD simulations disables FTL and ICL from the datapath of the firmware stack.

C. Flash Firmware Optimizations

Internal caching. ICL can cover most of cache associativities and page replacement schemes, but the read performance of the simulated SSD can still be behind that of a real device. While writes can be immediately serviced from the internal DRAMs, reads should be serviced from the taget flash, and therefore, performance can be directly impacted by different physical data layouts and various levels of internal parallelism [26], [27]. To address this, ICL performs a *parallelism-aware readahead* that loads multiple super-pages, existing across all physical dies, but having no resource conflict at flash-level, in advance. This readahead is activated only if multiple incoming requests exhibit a high degree of data locality. ICL keeps track of multiple start offsets and page lengths of the requests, which are cached in DRAM, and increases a frequency counter if the incoming requests are sequentially accessed right after the addresses of the previous ones (but no cache hit). When the frequency counter becomes greater than a threshold, which is reconfigurable, ICL performs the readahead.

FTL mapping. Even though a super-page based mapping algorithm increases the level of internal parallelism, thereby achieving higher bandwidth, we observed that small random writes can significantly degrade overall performance. This is because small-sized writes introduce many read-and-modify operations, which give a burden on the storage complex. Specifically, if the target cache line of ICL to evict/flush is smaller than a super-page unit, ICL needs to first access other physical pages associated with the target super-page, and then write the pages (but still as a super-page) to storage complex, which in turn introduces read-and-write intermixed I/O workloads thereby making more resource conflicts. A challenge behind this optimization is that, when ICL writes a small-sized page from its eviction, the actual address of the target page should be updated by FTL, due to the erasure-before-write characteristic. If FTL maps the addresses based on the physical page unit, it needs to search all page table entries to find out other pages, which can be spread across all other channels (or ways). Thus, if there is a small-sized write, FTL selectively remaps the target page (but exists in a same channel or way) within the corresponding super-page by employing a super-page based hashmap.

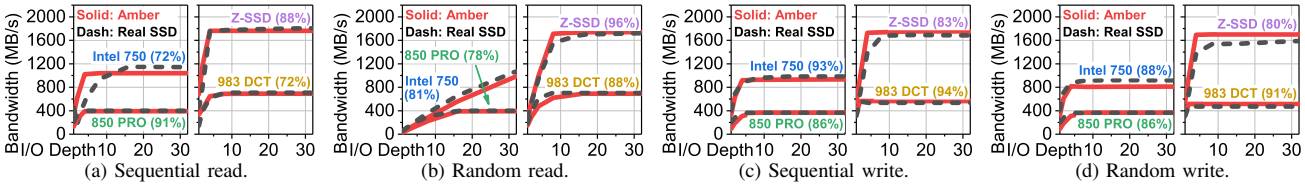
V. Evaluation

A. Methodologies

Simulation configurations. We configure the SSDs with 60 flash packages whose programming latency vary from 413 us to 1.8 ms, and the range of read latency is between 57 us and 94 us. This latency variation fundamentally exists in most MLC flash, due to the material-level nature of incremental step purse programming (ISPP) [58]. The SSD interconnection between storage complex and computation complex is implemented via AXI (250MHz) and AMBA (250MHz), and the interface linking the underlying flash is ONFi 3 (333MHz). 3 ARMv8 CPU cores are used for executing our SSD firmware. As default, we use a page-level mapping for FTL, and ICL is configured as a fully-associative cache. We model DDR3L for internal DRAM, and for SSD interfaces, we use SATA 3.0, NVMe 1.2.1, UFS 2.1 and OCSSD 2.0. The default configuration of storage complex (e.g., channels, packages, dies, etc.) is the same as that of the real device we used in Table I of Section III. To explore a full space of simulator evaluations, we also apply different storage complex configurations similar to the other testbeds that we used in Section V-B. Table II tabulates the host system configuration and timing parameters for gem5.
Workloads. The important characteristic and corresponding descriptions of our workloads [59] are listed by Table III. While most of the workloads exhibit small-sized requests (8∼10 KB), the average request sizes of *24HRS*(W2) and *MSNFS*(W5) are 28KB and 74KB, respectively. All the evaluations presented in this section are performed by *running actual storage applications at user-level* with a full system stack, instead of replaying or generating traces within the storage simulator.

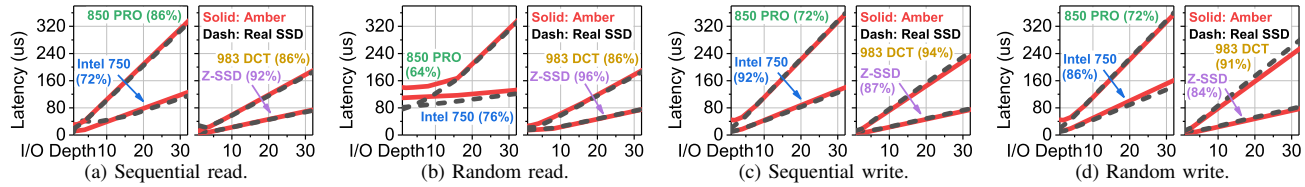
B. Validation

Real devices. In addition to the real device (Intel 750), we prepare three more real SSD devices and compare their performance with simulation results: i) Samsung 850 PRO (H-type), ii) Samsung Z-SSD prototype (S-type), and iii) Samsung 983 DCT SSD prototype (S-type). We evaluated other NVMe and SATA devices, but their performance significantly fluctuate/untenable and is far away from the above, and therefore, we carefully select those four devices as comparison target testbeds. 850 PRO is composed by multiple MLC flash over 8 interconnectors. 983DCT is similar to 850 PRO, but



(a) Sequential read. (b) Random read. (c) Sequential write. (d) Random write.

Fig. 8: Bandwidth trend and accuracy comparisons for real devices' and Amber's simulation results.



(a) Sequential read. (b) Random read. (c) Sequential write. (d) Random write.

Fig. 9: Latency trend and accuracy comparisons for real devices' and Amber's simulation results.

it supports the multi-stream feature (newly added in NVMe). Z-SSD uses NVMe interface and protocol, but their backend media is replaced with new flash that supports 3us and 100us I/O latency for read and write, respectively [60]. Note that *all configurations of our Amber that we evaluate for this validation will be available to download with the simulation source codes.*

Bandwidth comparison. Figure 8 compares the user-level performance trend of four different SSDs simulated by Amber (solid lines) and real devices (dashed lines). The figure also includes the average accuracy of our simulation for each real target device. One can observe from these plots that, for all micro-benchmark evaluations (e.g., reads, writes, sequential and random patterns) the Amber's bandwidth curve exhibits a very similar trend compared to that of the real devices, as the I/O queue depth increases. Specifically, the average accuracy of Amber's simulated bandwidth for Intel 750 ranges from 88% and 93% for sequential and random writes, and for reads, it also exhibits 72% $\sim$ 81% accuracy while closely following the bandwidth curve of the real device. Even though real devices exhibit different curve patterns with different bandwidth levels, the accuracy of Amber is in the range of 72% $\sim$ 96%.

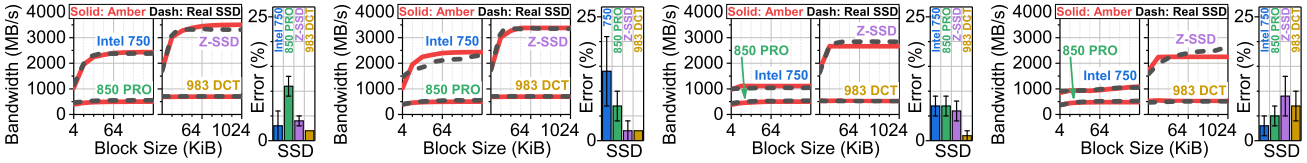
Latency comparison. Figure 9 shows the latency trend and accuracy of simulated results compared to each real target device when varying I/O queue depth. As shown in the figure, the user-level latency curve simulated by Amber almost overlaps with that of the real device under all micro-benchmarks as the I/O depth increases, and the accuracy of simulation ranges between 64% and 94% for all real devices. The difference between simulation results and real device latency (especially Intel 750 and 850 PRO) is relatively more notable at a low I/O queue depth (1 $\sim$ 8), compared to the cases with higher queue depths. We believe that this is because Intel 750 and 850 PRO have several optimizations, which operate at a specific I/O pattern. Even though the optimization techniques such as caching data with battery-backed DRAM [61] or dumping them to the underlying flash by only using fastest pages of MLC are not published in a public domain, the varying and unsustainable performance of such devices with a specific I/O pattern are reported in various publicly available articles [62], [63].

Validation with different block sizes. Figure 10 shows bandwidths of the real device and Amber. In this evaluation, we increase the block size from 4 KB to 1024 KB. The figure also includes error rate results for each device and the corresponding range (from minimum errors to maximum error rates), as an evaluation summary. The error range is calculated by $|(Perf_{real} - Perf_{sim})|/Perf_{real}$, where $Perf_{real}$ and $Perf_{sim}$ are the real devices' performance and Amber's simulation results, respectively. As shown in the figure, the performance curve from simulation is very similar to the performance curves of all real devices with the tests of varying block sizes. While Amber simulation results catch up the trend of real device performance, the error rates (measured at user-level of the host) are in still reasonable range of 6% for all sizes of I/O request blocks across all SSDs. In these sensitivity tests in terms of different block sizes, Intel 750 shows higher error rates than those of other real devices (14% in random read, on average). We believe that this is because of the internal optimizations that we mentioned in the latency comparisons.

Over-provisioning. Note that all of above validations are performed after fully filling the target storage spaces with sequential writes for each test set (e.g., STEADY-STATE). Even though we perform all evaluations with the steady-state, the write performance can vary based on the ratio of over-provision (OP). Thus, we evaluate the write performance with different block sizes (4KB $\sim$ 1024KB) by reducing the OP ratio from 20% (default as same with Intel 750's OP rate) to 5%, and the results are plotted in Figure 11. To create a worst-case scenario as a stress test, we randomly write data for the entire space into the steady-state SSDs, which means that the amount of written data is 2$\times$ greater than the actual target volume. The simulation results with 15% and 10% and 5% OP rates show significant performance drops (by 87.9%, 62.1%, and 33.7%, respectively). This is mainly because of too many invocations, which lead to frequent migrations of data over different flash packages, thereby introducing not only long I/O latencies but also frequent resource conflicts.

C. Operating System Impacts

Figure 12 shows the performance of real workloads (executed at a user-level) when employing different versions of OS (4.4 and 4.14). Interestingly, the read and write performances of Kernel 4.4 are worse than that of Kernel 4.14 by 63% and 69%, on average, respectively. The major difference between those two versions of Linux kernel from the viewpoint of



(a) Sequential read. (b) Random read. (c) Sequential write. (d) Random write.

Fig. 10: Performance validation with different block sizes ranging from 4KB to 1024KB.

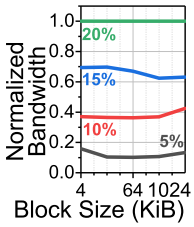


Fig. 11: OP.

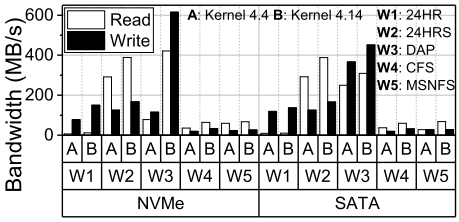
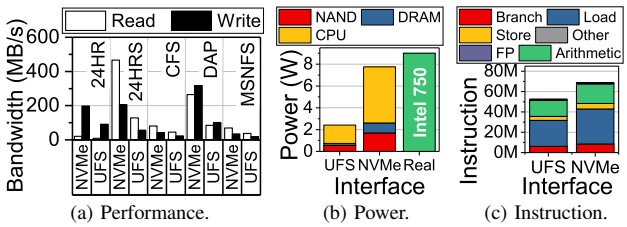


Fig. 12: Performance impacts on OS.



(a) Performance. (b) Power. (c) Instruction.

Fig. 13: General computing vs. handheld computing

storage stack is the disk scheduler. The newer version of Kernel (4.14) employs a refined Budget Fair Queueing (BFQ) [64], which assigns a budget to each time slice for scheduling I/O requests by considering the number of sectors (i.e., request length). This refined BFQ is different from the original BFQ as it employs per-process queues optimized for SSDs. BFS also uses a unified mechanism to merge incoming requests to improve the performance through sequential access patterns. In contrast, the older version of Kernel (4.4) changes the disk scheduler to Completely Fair Queuing (CFQ) [65], which removes an anticipatory scheduling mechanism [66], and allows the other kernel scheduler to dispatch the I/O requests thereby improving throughput. Even though one would expect the newer version of Kernel that reflects the latest features of NVMe and SATA, compared BFQ, CFQ is not able to generate enough I/O requests to saturate the SSD performance and consumes CPU cycles in I/O scheduling. To dig deeper into these OS-bound issues that prevent the system from serving enough I/O requests, we increase Amber's host-side CPU clock frequency from 2 GHz to 8 GHz by employing the fastest SSD (Z-SSD), and the results are given in Figure 14. Thanks to the new flash technology, the performance at the device-level now reaches at 4.3 GB/s. However, Kernel execution (User-level) and protocol management (Interface-level) with 2 GHz CPU degrades the performance by 41% compared to the device-level performance. When we run the same Kernel with a higher frequency (8GHz), the user-level performance is enhanced by 12%. Note that Kernel (with 8GHz operations) still slashes device-level performance by 29%, but we believe that future efforts that tune the Kernel and scheduler to make them better aware of the storage stack can reduce the performance loss more, and Amber can help such efforts as a research vehicle with the full functionality of holistic system simulation.

D. Handheld vs. General Computing

Figure 13a compares the user-level performance of a mobile system and that of a personal computing device, which use UFS and NVMe protocols, respectively. One can observe from this figure that NVMe shows 1.81 times better performance than UFS. Even though the performance of NVMe exceeds that of UFS, for *24HR*, *CFS* and *MSNFS* workloads. The difference between them is less than 15%. This is because of the low computing power in the mobile system, as described in the previous section; the low power mobile cores cannot generate enough I/O requests to enjoy all potential benefits of NVMe performance. As one would expect that several handheld devices such as tablets will have higher computing power in the future, NVMe might be a better solution to take off the storage accesses from the critical path. However, to this end, NVMe devices also require a significant technology enhancement. Specifically, Figures 13b and 13c breakdown the power and instruction sets for the SSD executions, respectively. The CPU that exists at the SSD-side is the most power hungry component. Considering the power budget of handheld computing, it requires significantly optimization of the underlying SSD's power usage. As shown in Figure 13c, load and store are most dominant instructions, which account for 60% of the total executions executed. We also observe that NVMe can execute more instructions than UFS within a same time period (by 5.45 times). This is because the core that handles NVMe queue should be involved everytime the doorbell rings. Note that, as shown in the figure, the power that Amber reports for NVMe, including internal CPU, flash-backend (NAND) and DRAM, is similar to that of the NVMe real device (Intel 750). While the power simulated by UFS is around 2W, the majority of such power is consumed by the internal CPU, and therefore, it can be reduced to hundreds *m*W with hardware automation in the mobile storage.

E. Passive vs. Active Storage

Figure 15a shows the performance of NVMe SSD (Active approach) and OCSSD 2.0 (Passive approach). As it can be observed, passive approach is better to service I/O requests than active approach in case of small I/O accesses (4KB-sized) with both random and sequential patterns. Specifically, OCSSD shows a 30% better throughput than NVMe SSD for 4KB-sized requests. This is mainly because OCSSD is in a better position to utilize host-side buffer cache with better information, which can in turn directly serve the request from the host rather than going through the underlying storage. However, in case of large-sized I/O accesses (64KB), NVMe exhibits a 20% better average throughput, due to the limited system buffer

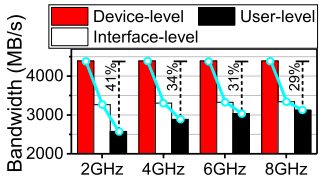
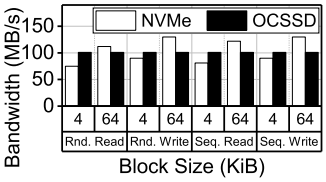
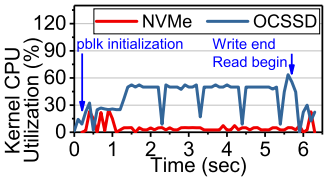


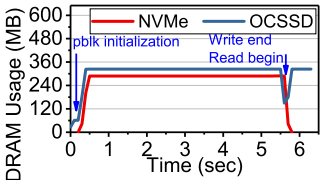
Fig. 14: Performance with CPU frequency changes.



(a) Overall performance.



(b) Kernel CPU Utilization.



(c) Total DRAM usage.

Fig. 15: Overall performance and system utilization of Active and Passive SSD.

	Mode		H-CPU(ARM)						Host interface				C-Cplx[3]		S-Cplx[4]			Cache			FTL		Power			Dynamics		Sup[13]
	SA[1]	FS	Atomic	Timing	Minor	HPI[2]	DerivO3	O3_v7a	SATA	UFS	NVMe	OCSSD	CPU	DRAM	Tranx[5]	SP/SB[6]	ISPP[7]	Config[8]	RA[9]	Full[10]	Hybrid	Page[11]	CPU	DRAM	NAND	Energy	Exec[12]	Queue
Amber	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SimpleSSD 1.x [13] [14]	✓	✓	✓								✓			✓	✓	✓	✓	✓	✓	✓	✓				✓			✓
MQSim [15]	✓								✓		✓			✓	✓		✓			✓	✓	✓						✓
SSDSim [32]	✓																			✓	✓	✓						
SSD-Extension [12]	✓															✓						✓						
FlashSim [33]	✓															✓					✓	✓						

[1] Standalone [2] High perf. In-Order [3] Computation complex [4] Storage complex [5] Transaction scheduling [6] Super page/block [7] Incremental step pulse programming [8] Configurable cache [9] Readahead [10] Fully associative [11] Page-level mapping [12] Dynamic firmware execution [13] Support

TABLE IV: Feature comparison across various simulators.

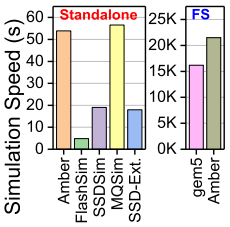


Fig. 16: Execution time.

capacity that kernel-level drivers can use. Note that, in contrast to user-level memory, kernel memory is not freely used as it directly allocates the buffer from the physical memory address space instead of virtual memory space.

Figures 15b and 15c plot the CPU utilization and memory requirement of OCSSD and NVMe devices, respectively. At the beginning of I/O processing, FIO consumes similar CPU cycles for both OCSSD and NVMe SSD due to write initialization for warming up processes. However, after the initialization, OCSSD consumes 50% of all the cores (four) whereas NVMe SSD only uses 10% of CPU. This is because, OCSSD requires to run `pblk` and `LightNVM` drivers to perform address translations, memory caching, and physical flash media management. On the other hand, `pblk` and `LightNVM` drivers do not use DRAM memory too much compared to NVMe SSD. As they are kernel drivers, OCSSD allocates system memory at the initialization phase, and reuses them for the entire execution phase (64 MB). Since FIO and NVMe protocol management basically requires about 280 MB of memory, the driver's memory requirement pressure can be ignored.

VI. Simulator Comparisons and Related Work

In literature, only a few simulators exist for high-performance SSD modeling. SSD Extension for DiskSim (SSD-Extension) [12] is the most popular simulator, which extends DiskSim [67] by adding SSD models. This simulator simulates a page mapping FTL built upon a simplified flash model. FlashSim [33] tries to extend the mapping algorithm from the page-level to different associativities, but has no flash model or queue at its storage complex. In contrast to SSD-Extension, SSDSim [32] can capture the details of internal parallelism by considering flash dies, planes and channel resources extracted by an in-house FPGA platform. However, SSDSim has no model for storage interface and the corresponding queue control mechanisms. MQSim [15] models a storage complex and enhances SSD simulation in comparison, by adding simple DRAM/flash and protocol management latency models, which partially consider the latency of the computation complexity. However, it cannot capture any internal embedded core latency, and has no capability of SSD emulation (that store/load actual data). As real contents and data movements are omitted in its model, MQSim cannot have a full storage stack on the host side; a system emulation mode of gem5, which only captures the system timing without having file system or storage stack might be possible to execute, but an SSD-enabled full system by having all software/hardware components for both host and storage cannot be explored. We checked all the repositories of their simulation framework and verified the unavailability of full-system simulations. There are a few SSD simulations built on QEMU [68], which allow the SSD simulators communicate with a host emulation system. Unfortunately, FEMU [69] removes many necessary components of the storage stack to achieve a reasonable simulation speed, and VSSIM [70] only supports IDE-based storage. In contrast, SimpleSSD [13] can be attached to gem5 over IDE and emulate data transfers, and its extension [14] employs NVMe queue protocols that handle a pointer-based communication between host and storage-side simulations. However, these simulators cannot accommodate diverse storage interfaces and protocols atop different CPU timing and detailed memory models, which require all responses for software/hardware modules in a very-fine granular manner. In addition, none of the aforementioned simulators has the capability of capturing the SSD's computation complex, including dynamic firmware executions, and power/energy measurement.

Table IV summarizes the differences between the existing SSD simulators and our proposed *Amber*. *Amber* can capture embedded CPU cores' performance such as CPI and break down firmware executions into "branch", "load/store" and "arithmetic" operations, which none of the existing simulators is able to perform. In addition, *Amber* can report dynamic power consumption of firmware stack executions, by taking into account the embedded cores, internal DRAMs, and flash devices. By modifying system barbus and implementing a DMA engine (that enables SSD emulation), *Amber* can operate under both functional CPU and timing CPU models in full-system environment. It also models different host controllers (SATA/UFS) and drivers (NVMe/OCSSD), which can enable all

s-type storage and h-type storage under diverse computing from mobile and general computing systems. Figure 16 compares simulation speeds of diverse standalone simulators (left), and pure gem5 and Amber (right). The simulation speed of Amber is slightly better than that of MQSim, and it captures all SSD-enabled full system characteristics with a reasonable simulation speed.

VII. Acknowledgement

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. The simulator is designed, developed, and maintained by Computer Architecture and MEmory systems Laboratory (CAMELab). Myoungsoo Jung is the corresponding author.

VIII. Conclusion

We propose a new SSD simulation framework, Amber, which incorporates a full-featured SSD model into a full system environment and emulates all software stacks employed in diverse OS and hardware computing platforms. Amber modifies host buses, and implements plenty of storage interfaces and protocols such as SATA, UFS, NVMe, and OCSSD.

References

[1] D. E. Merry Jr and M. S. Diggs, "Solid state storage subsystem for embedded applications," Mar. 23 2010. US Patent 7,685,337.

[2] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 279–289, ACM, 2009.

[3] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–11, IEEE, 2012.

[4] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, *et al.*, "Accelerating science with the nersc burst buffer early user program," tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2016.

[5] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at los alamos national lab," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–5, IEEE, 2012.

[6] I. Ari, M. Gottwals, D. Henze, *et al.*, "Performance boosting and workload isolation in storage area networks with sancache," in *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 263–273, 2006.

[7] G. Zhang, L. Chiu, and L. Liu, "Adaptive data migration in multi-tiered storage based cloud environment," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 148–155, IEEE, 2010.

[8] "Serial advanced technology attachment (sata)." http://sata-io.org/.

[9] "Universal flash storage (ufs)." https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

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

[11] "Open-channel solid state drives." https://openchannelssd.readthedocs.io/en/latest/.

[12] V. Prabhakaran and T. Wobber, "Ssd extension for disksim simulation environment," *Microsoft Reseach*, 2009.

[13] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "Simplessd: modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 37–41, 2018.

[14] CAMELab, "Simplessd 1.0." https://simplessd.org/.

[15] A. Tavakkol, J. G. Luna, M. Sadrosadati, S. Ghose, O. Mutlu, and G. Juan, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, USENIX, 2018.

[16] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, pp. 181–192, ACM, 2009.

[17] M. Abraham, T. Tanaka, K. Kawai, and Y. Einaga, "Partial page memory operations," Apr. 19 2016. US Patent 9,318,199.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[19] D. H. Walker, "A comparison of nvme and ahci," *The Serial ATA International Organization*, 2012.

[20] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–11, IEEE, 2015.

[21] I. Livny, "Storage over pcie protocol analysis and traffic generation techniques." https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140807_I31_Livny.pdf.

[22] Xilinx, "Introduction interrupt types in pci express." https://www.xilinx.com/Attachment/Xilinx_Answer_58495_PCIe_Interrupt_Debugging_Guide.pdf.

[23] M. Jung, "Exploring design challenges in getting solid state drives closer to cpu," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1103–1115, 2016.

[24] M. Jung and M. Kandemir, "Challenges in getting flash drives closer to cpu," *Future*, vol. 2, no. 4.00, pp. 8–00, 2013.

[25] J. Zhang, M. Shihab, and M. Jung, "Power, energy, and thermal considerations in ssd-based i/o acceleration.," in *HotStorage*, 2014.

[26] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 266–277, IEEE, 2011.

[27] M. Jung and M. T. Kandemir, "An evaluation of different page allocation strategies on high-speed ssds.," in *HotStorage*, 2012.

[28] Y. Pan, G. Dong, and T. Zhang, "Exploiting memory device wear-out dynamics to improve nand flash memory system performance.," in *Fast*, vol. 11, pp. 18–18, 2011.

[29] I. Super Talent Technology, "Slc vs. mlc: an analysis of flash memory." http://www.supertalent.com/datasheets/SLC_vs_MLC%20whitepaper.pdf.

[30] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, pp. 177–190, ACM, 2015.

[31] H. Shim, B.-K. Seo, J.-S. Kim, and S. Maeng, "An adaptive partitioning scheme for dram-based cache in solid state drives," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–12, IEEE, 2010.

[32] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*, pp. 96–107, ACM, 2011.

[33] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pp. 125–131, IEEE, 2009.

[34] J. Axboe, "Flexible io tester." https://github.com/axboe/fio.

[35] ARM, "Arm limited: Arm architecture reference manual (armv8, for armv8-a architecture profile) (2013)." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.h/index.html.

[36] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009.

[37] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of ddr sdrams," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pp. 99–108, IEEE, 2011.

[38] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "Drampower: Open-source dram power & energy estimation tool." http://www.drampower.info.
[39] M. Jung, W. Choi, S. Gao, E. H. Wilson III, D. Donofrio, J. Shalf, and M. T. Kandemir, "Nandflashsim: High-fidelity, microarchitecture-aware nand flash memory simulation," *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, p. 6, 2016.
[40] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172–1186, 2010.
[41] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system.," in *USENIX*, pp. 155–164, 1995.
[42] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 524–535, IEEE, 2014.
[43] PCI-SIG, "Generic capability structure for sata host bus adapters." https://pcisig.com/sites/default/files/specification_documents/SATA_CAP_Ptr_ECN_-_final.pdf.
[44] QEMU, "Qemu the fast! processor emulator." https://www.qemu.org/.
[45] Linux, "libata developer's guide." https://01.org/linuxgraphics/gfx-docs/drm/driver-api/libata.html.
[46] Linux, "Scsi interface guide." https://01.org/linuxgraphics/gfx-docs/drm/driver-api/scsi.html#scsi-lower-layer.
[47] D. A. Deming, *The Essential Guide to Serial ATA and SATA Express*. CRC Press, 2014.
[48] PCI-SIG, "Pci express base specification revision 3.0," 2010.
[49] "Ufs host controller interface specification." https://www.jedec.org/sites/default/files/docs/JESD223.pdf.
[50] mipi, "A performance-driven phy for multimedia and chip-to-chip inter-processor communication (ipc) applications." https://mipi.org/specifications/m-phy.
[51] "Ufs 2.0 host stack and driver." https://www.design-reuse.com/hds/ufs-2-0-host-stack-and-driver-ip-790/.
[52] D. Flynn, "Amba: enabling reusable on-chip designs," *IEEE micro*, vol. 17, no. 4, pp. 20–27, 1997.
[53] "Microsoft sgl description." https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-scatter-gather-dma.
[54] "Open-channel solid state drives specificaion - revision 1.2." http://lightnvm.io/docs/Open-ChannelSSDInterfaceSpecification12-final.pdf.
[55] "Open-channel solid state drives specificaion - revision 2.0." http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.
[56] LightNVM, "pblk: Host-based ftl for open-channel ssds." http://lightnvm.io/pblk-tools/.
[57] LightNVM, "User space i/o library for open-channel ssds." http://lightnvm.io/liblightnvm/.
[58] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, *et al.*, "A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, 1995.
[59] M. Kwon, J. Zhang, G. Park, W. Choi, D. Donofrio, J. Shalf, M. Kandemir, and M. Jung, "Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, IEEE, 2017.
[60] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu, J. Kim, J. Park, K. W. Song, K. T. Park, S. Cho, H. Oh, D. D. G. Lee, J. H. Choi, and J. Jeong, "A flash memory controller for 15 µs ultra-low-latency ssd using high-speed 3d nand flash with 3 µs read time," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 338–340, Feb 2018.
[61] J. W. Stockdale, S. G. LeMay, and D. R. Nelson, "High performance battery backed ram interface," Oct. 12 2004. US Patent 6,804,763.
[62] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds.," in *HotStorage*, 2016.
[63] B. Nikkel, "Nvm express drives and digital forensics," *Digital Investigation*, vol. 16, pp. 38–45, 2016.
[64] "The bfq i/o scheduler." https://lwn.net/Articles/601799/.
[65] "Completely fair queuing (cfq)." https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.
[66] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, (New York, NY, USA), pp. 117–130, ACM, 2001.
[67] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," *Parallel Data Laboratory*, p. 26, 2008.
[68] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.
[69] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, (Oakland, CA), pp. 83–90, USENIX Association, 2018.
[70] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, May 2013.