SimpleSSD: Modeling Solid State Drives for Holistic System Simulation

Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir

Abstract—Existing solid state drive (SSD) simulators unfortunately lack hardware and/or software architecture models. Consequently, they are far from capturing the critical features of contemporary SSD devices. More importantly, while the performance of modern systems that adopt SSDs can vary based on their numerous internal design parameters and storage-level configurations, a full system simulation with traditional SSD models often requires unreasonably long runtimes and excessive computational resources. In this work, we propose `SimpleSSD`, a high-fidelity simulator that models all detailed characteristics of hardware and software, while simplifying the nondescript features of storage internals. In contrast to existing SSD simulators, `SimpleSSD` can easily be integrated into publicly-available full system simulators. In addition, it can accommodate a complete storage stack and evaluate the performance of SSDs along with diverse memory technologies and microarchitectures. Thus, it facilitates simulations that explore the full design space at different levels of system abstraction.

Index Terms—Hardware, computer architecture, parallel processing, computational modeling, systems simulation, microprocessors, software

1 Introduction

In the past decade, solid state disks (SSDs) have reshaped modern memory hierarchy by replacing conventional spinning disks and/or blurring the boundary between main memory and storage systems. Thanks to their high performance and low power consumption characteristics, SSDs have already become the dominant storage type in diverse computing domains, ranging from embedded to general-purpose and high-performance computing systems. This in turn has led to a wide spectrum of research, including the comprehensive exploration of the full design space, storage stack optimization, and architecture renovation at various layers of memory and storage subsystems.

While simulations are indispensable for system designers and computer architects, very few SSD simulators have been released to the public domain [5], [7], [8], [10]. Further, these simulators have constraints that prevent them from filling the needs of design space exploration for emerging memory and storage subsystems. First, all existing SSD simulators lack system-level simulation capability, and integrating these simulators with publicly-available full-system simulators is a non-trivial task. While the execution of a CPU instruction only takes a few cycles in a simulation, a storage access requires tens of millions (even billions) of cycles for its service. Similarly, a file access in an accurate SSD simulation model can exhibit a long execution time because it needs to go through the SSD's intricate software stack and hardware architecture. Traditional SSD simulators cannot fully account for the important functionalities of the underlying firmware and model the underlying hardware in detail. Thus, they are far from capturing the critical features of contemporary high-performance SSD architectures.

In this work, we propose `SimpleSSD`, a high-fidelity simulator that models all of the detailed characteristics of hardware and software while simplifying the nondescript features of storage internals such as multi-cycle operations to address a target page on a flash interface. The proposed hardware and software simplifications allow `SimpleSSD` to accommodate a complete storage stack. Thus, system designers and computer architects can evaluate the SSDs performance along with diverse memory technologies and can explore the full design space of an SSD architecture. Moreover, `SimpleSSD` can easily be integrated with publicly-available full-system simulators and can capture relevant CPU performance characteristics impacted by different storage types employed by the system. As a case study, we integrated `SimpleSSD` with the popular full-system simulator, gem5 [6], and evaluated its system-level performance from various aspects. Note that traditional SSD simulators [5], [7], [10] capture only storage-related metrics such as bandwidth and latency by replaying block-level I/O traces; this ignores system-level interaction between the host-side CPU and storage subsystems. In contrast, the proposed `SimpleSSD`[1] can report detailed information from low-level memory to each firmware module in order to determine the host-side CPU performance while executing entire applications.

2 SSD-Enabled System Simulation Overview

Fig. 1 shows an overview of a holistic system simulation with the proposed `SimpleSSD`. Application(s) simulated on the host can place an I/O request through a virtual file system (VFS) and native file system. The VFS buffers small-sized requests through a page cache, whereas the native file system manages the data accesses and system memory. The request then arrives at a block layer that reorders and combines multiple requests into a specific order. This CPU processing part can communicate with the layered firmware of `SimpleSSD` via a disk controller. Then, the layered firmware simulates the SSD process part by interacting with an abstraction model, which simulates the given SSD hardware architecture including multiple flash dies, module interfaces, and channels. Although `SimpleSSD` leveraged gem5 running in full-system mode to simulate such CPU processing in this study, it can easily be integrated into other full-system simulators such as MARSSx86 [13].

Layered Firmware. One of the main challenges of simulating an SSD is supporting diverse flash firmware versions, which greatly influences the target storage performance. We model a flexible *flash translation layer* (FTL) whose address translation mechanism can simply be reconfigured based on different associativity granularities defined by system architects. We also decouple I/O scheduling and page allocation mechanisms from the FTL so that new scheduling proposals that are aware of SSD-internal parallelism can be embedded without changing the FTL. Although we do not cover all types of potential FTLs, the implemented reconfigurable mapping algorithm can capture/support diverse operational characteristics of a block-level mapping FTL, a fully-associative FTL, and various hybrid mapping schemes that employ different levels of block and page mapping tables in their address translations. In addition, our simplified but reconfigurable layered firmware also offers diverse research opportunities where system and computer architects can simply modify some performance-critical components such as garbage collection and wear-leveling algorithms with different mapping mechanisms.

- *M. Jung, J. Zhang, and M. Kwon are with the Computer Architecture and Memory Systems Lab, Yonsei University, Seoul 03722, Republic of Korea. E-mail: {m.jung, jie}@yonsei.ac.kr, mkwon@camelab.org.*
- *A. Abulila and N. Sung Kim are with the University of Illinois Urbana-Champaign, Champaign, IL 61820. E-mail: {abulila2, nskim}@illinois.edu.*
- *N. Shahidi and M. Kandemir are with Pennsylvania State University, State College, PA 16801. E-mail: nxs314@psu.edu, kandemir@cse.psu.edu.*
- *J. Shalf is with the Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: jshalf@lbl.gov.*

Manuscript received 22 Feb. 2017; revised 29 Mar. 2017; accepted 12 May 2017. Date of publication 10 Sept. 2017; date of current version 19 Mar. 2018.
(Corresponding author: Myoungsoo Jung.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2017.2750658

1. The SimpleSSD source code can be freely downloaded from the following website: http://simplessd.camelab.org.

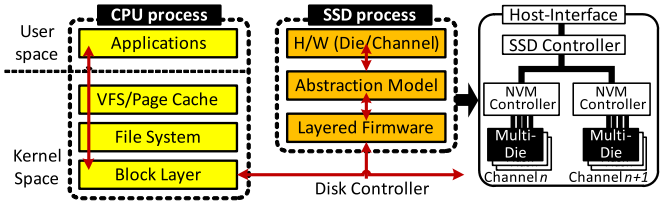


Fig. 1. Overview of SimpleSSD.

Hardware Abstraction. The performance characteristics of the underlying hardware vary based on i) the intrinsics of latency of individual flash characteristics and ii) their different levels of parallelism. A cycle-level simulation for each component can accurately evaluate all SSD internals. However, full-system simulations with an SSD at the cycle level require an unreasonably long runtime and excessive resources. In this work, we abstracted both flash-level and subsystem-level hardware characteristics. We implemented an FPGA-based memory controller built on Xilinx Spartan-6 and then used this to characterized different memory technologies. Based on the extracted characteristics, we first design a die-level latency model by simplifying the flash transactions. Specifically, we examined all flash transactions specified by the open NAND flash interface (ONFi 3.× [1]) and classified various timing components of the corresponding protocol into a few transaction activities. With this simplified latency model, the proposed SimpleSSD simulates varying numbers of flash chips over many interconnection buses by modeling the executions across different hardware resources and resource contentions. Even though this simplified model cannot account for all of the characteristics from the flash at a cycle level, it can capture the close interactions among the designs of the firmware, controller, and architecture by being aware of flash latency intrinsics and internal parallelism.

3 SimpleSSD

Fig. 2 shows a high-level view of SimpleSSD and explains how our simulator processes the incoming I/O requests. A request is first taken by the host interface layer (HIL), and the corresponding target address is translated by the flash translation layer (FTL). The parallelism allocation layer (PAL) then services the request by abstracting the physical layout of interconnection buses and flash dies. The completion of an I/O request is reported from PAL to the host-side controller via HIL.

3.1 Fully-Functional Firmware Simulation

Host Interface Layer.In SimpleSSD, HIL first receives an incoming request from the disk controller of gem5 and enqueues the request in a device-level queue. During this phase, it parses the host-side information and translates it a logical block address (LBA), request type, number of sectors, and a host's system time information (e.g., tick). HIL then forwards this translated information to the underlying FTL through communication APIs, `ReadTransaction()` and `WriteTransaction()`. Since there are many different types of simulation models for a full system (e.g., discrete event-driven, activity-driven, and continuous), HIL exposes all request completions through a latency map table, which includes the finish time (i.e., finishTick) along with each requested address. Once the latency for each request is updated by the underlying simulation modules, HIL updates the table with the completion time, and the full-system simulator (e.g., gem5) retrieves it in an asynchronous fashion. While the current queue implementation of HIL is first-come-first-served, system and computer architects can insert their buffer cache, I/O reordering logic, or scheduler into HIL.

Flash Translation Layer. The I/O sizes requested by a host application vary and can be even larger than the page size that a single flash die could accommodate. Therefore, in this work, FTL separates the request forwarded by HIL into multiple *sub-requests*, each indicated by a logical page number (LPN). If it is a read, FTL directly translates the sub-requests' LPNs to physical page numbers (PPNs) by looking up its own address mapping table. Otherwise, FTL allocates new page(s) and updates the table with appropriate block and/or page addresses and other meta-data information. In SimpleSSD, this address translation mechanism is implemented in a functional API, called `FTLmapping()`. The translated or allocated PPNs are then issued into the underlying module's queue by calling `SendRequest()`, and FTL repeats this process until there is no waiting sub-request. When there is no available page for a write, FTL performs garbage collection (GC) to reclaim a set of new pages in flash block(s). At the beginning of GC, it selects the victim blocks and free block(s) to allocate as a new block, which can be determined by a wear-leveling algorithm. After this selection, FTL reads the data from all valid pages of the victim blocks, writes them into the new block, and updates the address table for the reclaimed blocks. Note that the additional read and write operations imposed by GC(s) are treated just like other sub-requests from PAL viewpoint, but the latency associated with all the internal I/O requests is aggregated and exhibits long tail from FTL and HIL perspectives. In this work, we consider a simple GC algorithm (cf. greedy), which selects a victim block with the maximum number of invalid pages. The number of free blocks and GC threshold can be reconfigured based on user inputs. Besides, the wear-leveling algorithm we implemented always allocates new block(s) by considering the minimum erase count among the free blocks in a reserved pool. Users can replace these algorithms by updating the `GarbageCollection()` and `WearLeveling()`.

3.2 Hardware Simulation for Scalable SSD Parallelism

Parallelism Abstraction Layer. In this work, we introduce PAL underneath FTL and decouple SSD parallelism from other flash firmware modules for improved simulation efficiency and a better research-wise structure. PAL basically stripes all incoming requests across different channels, packages and dies, based on user configurations, which is similar to the striping method employed by RAID. At the beginning, PAL dequeues the requests issued by FTL and disassembles the target page address by being aware of the underlying hardware configuration (e.g., numbers of channels, flash

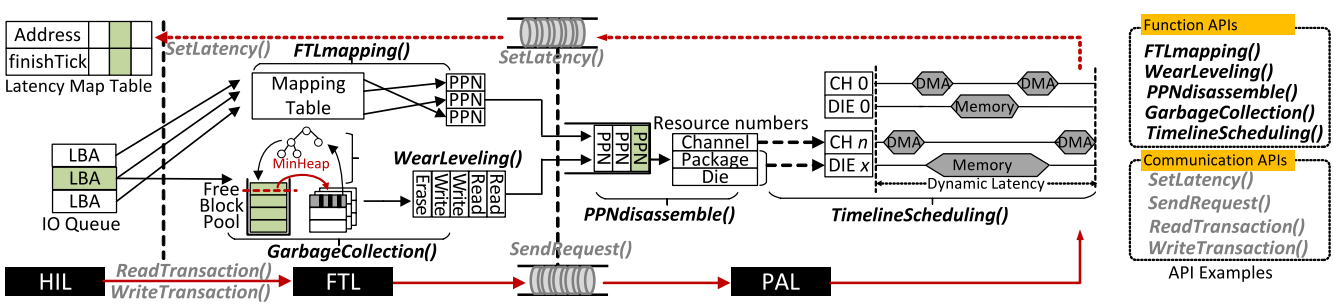
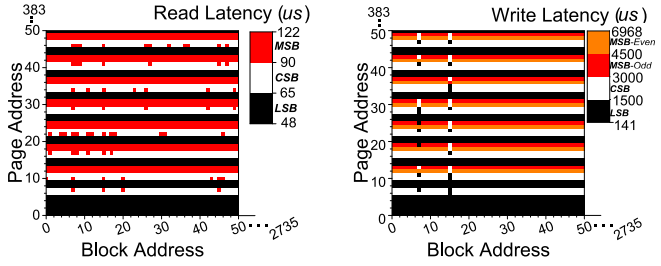


Fig. 2. High-level view of SimpleSSD.



(a) Read latency. (b) Write latency.

Fig. 3. Analysis for flash intrinsic latency variation.

packages, and dies). This is implemented with `PPNdisassemble()`. Based on the disassembled information, PAL simulates SSD internal state and schedules the flash transaction at a finer granularity to capture the memory-specific latency, idle time, and even scheduling penalties imposed by resource contentions. In other words, the latency of a sub-request can be dynamically simulated in `SimpleSSD` by considering not only the hardware resource availability but also the storage media configuration. After processing the I/O request, PAL returns the simulated latency for each sub-request to FTL. FTL then collects and reevaluates them to generate an appropriate latency for the I/O request that possesses such sub-requests. By being aware of the states of the underlying hardware, users can explore new parallelism strategies and schedulers. The order for sub-request striping or management of flash transactions can be determined by modifying `PPNdisassemble()` and `TimelineScheduling()`, respectively.

Latency Variation Mapping. To make the storage denser with the same number of transistors, flash can store multiple states into a single storage cell. For example, *triple-level cell* (TLC) flash stores eight different states into a target storage core. Each state is represented by different voltage thresholds (Vth). Because a TLC core can maintain 3-bit data, the TLC technology can drastically increase the storage capacity of an SSD. However, the materials of the TLC storage core are not fundamentally different from that of a *single-level cell* (SLC) or *multiple-level cell* (MLC), which can represent 1-bit or 2-bit data per cell, respectively. Instead, the flash logic of TLC (and MLC) writes (i.e., programs) data into a target in a different manner compared with SLC flash. This is referred to as an *increment step pulse program* (ISPP [14]) and introduces significant latency variation. To characterize the latency behavior incurred by the ISPP, we built an FPGA-based controller by using Xilinx Spartan-6 and tested SLC, MLC, and TLC NAND flash devices. Figs. 3a and 3b illustrate the latency variation observed for writes and reads on TLC 25 nm flash technology [11], respectively; we provide only the TLC results owing to the page limit, but other flash technologies also exhibited the same latency trend that we observed for TLC. The evaluation data were measured for every single block and page. For writes, the latency of the *most significant bit* (MSB) pages was longer than those of the *center significant bit* (CSB) and *least significant bit* (LSB) pages by approximately 1.3 and 8 times, respectively. The reads on TLC flash also exhibited similar latency variation characteristics. Specifically, the read latency of MSB pages is longer than that of CSB pages and LSB pages by 37 and 84 percent, on average, respectively. Since the latencies between different pages exhibit a notable difference, this can have a great impact on parallelism and hardware modeling.

TABLE 1
Configurations for System Simulations

Channel	Package	Die	Plane	Block	CPU	ARMv7, 1 core, 1GHz
8	8	4	2	1024	Cache	L1I$/L1D$, 64KB, 4-way
DMA(MHz)	**Pages/blk**	**Page**	**OP**	**GC**		L2$, 512KB, 8-way
400	256	8KB	0.2	0.05	**DRAM**	DDR3, 1 channel, 2 rank

We observed that the first five pages within a block always exhibited LSB page performance, and the latency of the next three pages (i.e., after the first five) was the same as that of the CSB pages. These eight pages, referred to as *meta pages*, are usually used for storing the metadata of flash firmware, such as mapping information associated with the block. The latency for all remaining pages can be mapped with the following simple function: $f(addr) = (addr - n_{meta})/n_{plane} \bmod n_{state}$ where $addr$, n_{meta}, n_{state} and n_{plane} are the input address, number of meta pages, number of states per cell and number of planes within a flash die, respectively. If $f(addr)$ is 0, it is an LSB page. If $f(addr)$ is 1, it is a CSB page. Otherwise, the address indicates an MSB page.

4 Evaluation

System Devices and Software Configurations. We configure a host that employs an eight-bank DDR3-1600 DRAM and 1 GHz CPU (ARM). The underlying storage is configured as an eight-channel high performance SSD device. Each channel connects eight packages, each with four TLC flash dies. FTL of this baseline is configured with a set-associative mapping algorithm, which associates eight log blocks with a single physical block. FTL has 20 percent overprovisioning (OP) space, and its GC threshold is set to 5 percent. The detailed information for system configurations, including CPU, SSD and flash, are given by Table 1. Lastly, we simulate SSDs with Linux 3.13.0 and EXT2 file system driver.

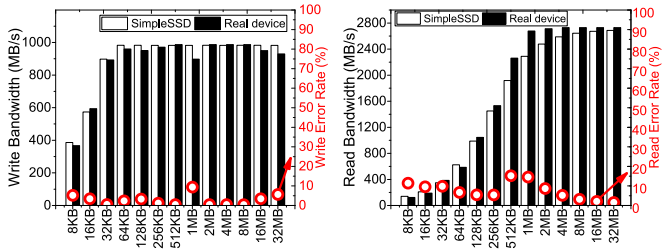
Workloads. In this evaluation, we use 13 different workloads. Specifically, *Apache*Bench [4] is used to measure the performance of an HTTP web server, where a specified URL is processed by generating heavy storage reads for the corresponding HTTP file(s). *Filebench* [15] includes several storage-centric workloads; each creates, writes and reads a few thousand files. In addition to these basic file I/Os, *fileserver* appends data and performs several file-sync operations with multiple threads, whereas *varmail* and *webserver* repeatedly read 1,000 small-sized files and write logs. Compared with *webserver*, *varmail* has extra I/O operations related to file deletion and creation. Finally, *Iozone* [12] evaluates a file system with a given automatic mode, and *mmap* [3] keeps reading and writing many files over POSIX library's APIs. Table 2 lists the important characteristics of these workloads.

4.1 Performance Validation

We compare the performance of `SimpleSSD` simulations in standalone mode with that of a real device (Intel 750). Specifically, we use multiple storage traces of ATTO [2] to analyze the disk-level characteristics in detail. Basic read and write tests were performed with varying I/O request sizes. Fig. 4 shows the results. For all requests ranging in size from 8 KB to 32 MB, the percentage difference (i.e., error rate) between the results of `SimpleSSD` and Intel

TABLE 2
Important Characteristics of the Tested Benchmarks

Workloads	apache1	fileserver1	fileserver2	fileserver3	fileserver4	varmail1	varmail2	varmail3	varmail4	webserver1	webserver2	iozone	mmap
Storage/K Instruction	26	82	127	86	126	8	6	7	6	5	4	57	109
SSD Read Ratio (%)	99	5.5	2.2	6.1	2.3	60	74	60	73	99	99	4	51
Max Instructions (B)	5	18	5	17	5	3	3	3	3	3	3	4	3



(a) Writes. (b) Reads.

Fig. 4. Set of evaluations for performance validation.

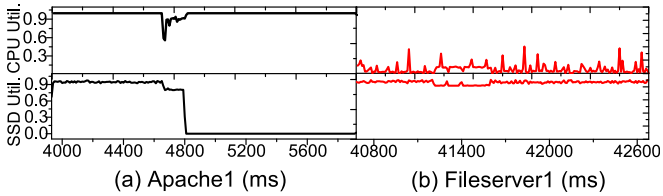


Fig. 6. Time series analysis.

750 is 2.7 percent on average, and the performance trends are similar. When the request size is increased, the bandwidth of both drives quickly increased and saturates at the 64 KB. On the other hand, the percentage difference of the reads is 7.1 percent on average. While the performance trends of the two devices are similar, the `SimpleSSD` performance increases more gradually than that of the real device; this makes the read error rate slightly higher than the write error rate. We conjecture that the real device has vendor-specific optimization, such as read-ahead or caching. Note that the current version of `SimpleSSD` has no specific buffer caching algorithm or acceleration model, which can introduce a greater performance disparity (compared to Intel 750) for small-sized I/O request tests. In addition to these microbenchmark tests, we also validate `SimpleSSD` by comparing its performance with that of a real device when executing 14 real storage workloads [9], [16], which includes real storage access patterns of a web server, database, and enterprise cluster. We observed that the performance trend of `SimpleSSD` with these workloads is similar to that of the real device. More practically, for these real workload evaluations, the difference between them is 9 percent on average.

4.2 SSD-Enabled Full System Evaluation

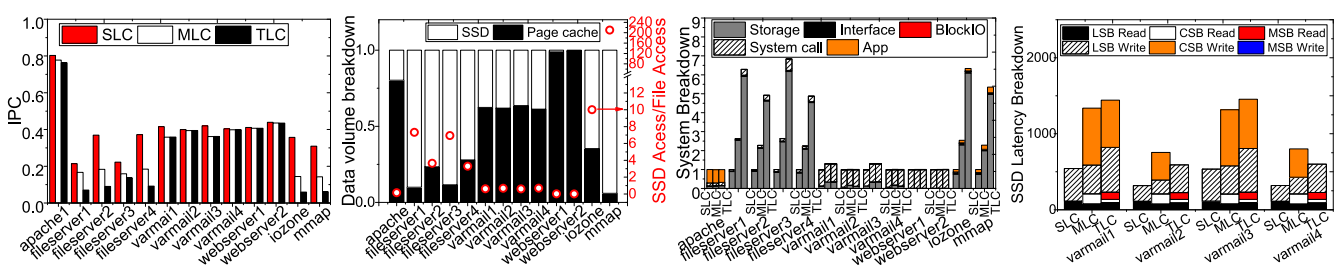
Overall CPU Performance. Fig. 5a shows the CPU performance (IPC) of hosts that employ different flash technologies (i.e., SLC/MLC/TLC) as their storage subsystems. All IPCs are normalized to those of the SLC version. As expected, the SLC-equipped system has better IPC than the MLC- and TLC-equipped systems by averages of 44 and 141 percent, respectively. Interestingly, *apache* and *webserver* show small or almost no performance benefit over SLC. As shown in Fig. 5b, even though these servers read many files, most of them are served from VFS's page cache. In contrast, *fileserver*, *iozone* and *mmap* have poor locality regarding the target (i.e., they touch once and never refer again), and have many fsync and/or flush operations, which make the page cache inefficient. A total of the 19 percent of I/O accesses is served by the page cache, on average. Even though *varmail* also exhibit many reads like *webserver*, it has slightly different performance characteristics. We explain the reason shortly.

Storage Stack Analysis. Fig. 5c decomposes the execution time spent for each component. It excludes overlaps of time with the latency consumed by the underlying component. For a better comparison, all MLC and TLC values are normalized to SLC ones. As expected, file-intensive benchmarks including *fileserver*, *iozone* and *mmap*, spend the most time accessing the underlying storage. Thus, the SLC-equipped system performs better than the MLC- and TLC-equipped systems by around $2.5\times$ and $5.8\times$, respectively. However, *apache* shows a completely different performance behavior than *fileserver*. Specifically, it consumes more CPU cycles at the user application level (68 percent of the total time) rather than storage accesses. This is because most of the cycles consumed by a block layer and system call overlap with those of underlying storage services, while processing the HTTP service keeps the entire CPU busy. For better understanding, we analyze the time series of CPU utilization and SSD utilization, which are measured at the end of benchmark executions for 2s (cf. Fig. 6). Compared to *fileserver1*, which utilizes the CPU 11 percent of the time on average while utilizing the SSD almost 100 percent of the time, *apache* activates CPU constantly. It has many overlaps with the SSD activities. Even after the SSD completes all read services, *apache* continues to process their data, which exhibit a high IPC.

Device Analysis. Fig. 5d shows the page-level latency breakdown for four *varmail* workloads. Interestingly, the write patterns of *varmail2* and *varmail4* have no address associated with CSB and MSB pages. Because all of the writes are served from the LSB pages, the TLC-based SSD has 34 and 32 percent shorter latencies on average, respectively, than the MLC-based SSD. However, these performance benefits are not directly reflected in the IPC, as shown in Fig. 5a. This is because, as shown in Fig. 5c, most of the time spent by *varmail* is consumed by system calls, which are primarily related to handling the page cache. This time consumed by the system calls, which does not overlap with the underlying device operations, accounts for more than 90 percent of the overhead for all executions.

4.3 Related and Future Work

There are very few SSD simulators in literature that are publically available for download [5], [7], [8], [10]. Even with these simulators, constraints prevent design space exploration for emerging memory/storage hierarchies. First, the hardware organization of existing simulators [5], [10] is unfortunately overly-simplified and far from capturing the critical features of high-performance contemporary SSD architectures. There is neither a specific flash microarchitecture nor an internal parallelism model. In addition, these simulators cannot fully reflect the important functionalities



(a) IPC. (b) Data access breakdown. (c) System breakdown. (d) SSD latency breakdown.

Fig. 5. System-level performance analysis with three different non-volatile memory technologies.

of the underlying flash firmware, which also have a great impact on system performance. The simulators have no FTL [7], [8] or an ideal FTL [5]. Note that none of these existing SSD simulators can be directly used for full system simulations.

In contrast, our `SimpleSSD` not only models contemporary SSDs by employing a complete storage stack and detailed hardware parallelism but also enables system-level simulation by considering different flash memory technologies. Thus it enables researchers to study diverse system performance characteristics from a holistic viewpoint.

Future Work. Computer Architecture and Memory Systems Laboratory (CAMEL) is extending the current simulation framework by implementing new features such as PCIe-enabled system/IO crossbars, message-signaled interrupts, internal DRAM models, NVMe interfaces and memory power models.

5 Conclusion

We proposed a high-fidelity SSD simulator that builds a complete storage stack from scratch and models all detailed characteristics of SSD internal hardware and software. This simulator can be integrated into publicly-available full system simulators.

Acknowledgments

This research is mainly supported by NRF 2016R1C1B2015312. This work is also supported in part by IITP-2017-2017-0-01015, NRF-2015M3C4A7065645, DOE DE-AC02-05CH 11231, and MemRay grant (2015-11-1731). Dr. Kim is supported in part by US National Science Foundation 1640196 and SRC/NRC NERC 2016-NE-2697-A. Dr. Kandemir is supported in part by US National Science Foundation grants 1439021, 1439057, 1409095, 1626251, 1629915, 1629129 and 1526750.

References

[1] Open NAND Flash Interface Specification Revision 3.0., ONFI Workgroup Mar, 2011.
[2] ATTO Disk Benchmark, 2014. [Online]. Available: www.atto.com/disk-benchmark
[3] mmap-benchmark, 2014. [Online]. Available: github.com/exabytes18/mmap-benchmark
[4] Apache HTTP server benchmark tool, 2014. [Online]. Available: httpd.apache.org
[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.
[6] N. Binkert, et al., "The gem5 simulator," *ACM SIGARCH Comp. Archit. News*, vol. 39, pp. 1–7, 2011.
[7] 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 *Proc. Int. Conf. Supercomputing*, 2011, 96–107.
[8] M. Jung, et al., "NANDFlashSim: Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level," in*Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.
[9] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008. pp. 119–128.
[10] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for NAND Flash-based solid-state drives," in *1st Int. Conf. Advances Syst. Simul.*, 2009, pp. 125–131.
[11] MICRON, Mt29f64g08, 2014. [Online]. Available: http://goo.gl/SdvjyV
[12] W. D. Norcott and D. Capps, "Iozone filesystem benchmark," 2003. [Online]. Available: http://www.iozone.org
[13] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore X86 CPUs," in *Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf.*, 2011, pp. 1050–1055.
[14] K.-D. Suh, et al., "A 3.3 v 32 mb NAND flash memory with incremental step pulse programming scheme," *IEEE J. Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, Nov. 1995.
[15] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX*, vol. 41, no. 1, 2016, https://www.usenix.org/node/195598
[16] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "Srcmap: Energy proportional storage using dynamic consolidation," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, pp. 20–20.