MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices

Arash Tavakkol[†], Juan Gómez-Luna[†], Mohammad Sadrosadati[†], Saugata Ghose[‡], Onur Mutlu[†‡]
[†]*ETH Zürich* [‡]*Carnegie Mellon University*

Abstract

Solid-state drives (SSDs) are used in a wide array of computer systems today, including in datacenters and enterprise servers. As the I/O demands of these systems continue to increase, manufacturers are evolving SSD architectures to keep up with this demand. For example, manufacturers have introduced new high-bandwidth interfaces to replace the conventional SATA host–interface protocol. These new interfaces, such as the NVMe protocol, are designed specifically to enable the high amounts of concurrent I/O bandwidth that SSDs are capable of delivering.

While modern SSDs with sophisticated features such as the NVMe protocol are already on the market, existing SSD simulation tools have fallen behind, as they do not capture these new features. We find that state-of-the-art SSD simulators have three shortcomings that prevent them from accurately modeling the performance of real off-the-shelf SSDs. First, these simulators do not model *critical features of new protocols* (e.g., NVMe), such as their use of multiple application-level queues for requests and the elimination of OS intervention for I/O request processing. Second, these simulators often do not accurately capture the impact of advanced SSD maintenance algorithms (e.g., garbage collection), as they do not properly or quickly emulate *steady-state conditions* that can significantly change the behavior of these algorithms in real SSDs. Third, these simulators do not capture the full *end-to-end latency* of I/O requests, which can incorrectly skew the results reported for SSDs that make use of emerging non-volatile memory technologies. By not accurately modeling these three features, existing simulators report results that *deviate significantly* from real SSD performance.

In this work, we introduce a new simulator, called MQSim, that *accurately* models the performance of both modern SSDs and conventional SATA-based SSDs. MQSim faithfully models new high-bandwidth protocol implementations, steady-state SSD conditions, and the full end-to-end latency of requests in modern SSDs. We validate MQSim, showing that it reports performance results that are only 6%-18% apart from the measured actual performance of four real state-of-the-art SSDs. We show that by modeling critical features of modern SSDs, MQSim uncovers several real and important issues that were not captured by existing simulators, such as the performance impact of inter-flow interference. We have released MQSim as an open-source tool, and we hope that it can enable researchers to explore directions in new and different areas.

1 Introduction

Solid-state drives (SSDs) are widely used in today's computer systems. Due to their high throughput, low response time, and decreasing cost, SSDs have replaced traditional magnetic hard disk drives (HDDs) in many datacenters and enterprise servers, as well as in consumer devices. As the I/O demand of both enterprise and consumer applications continues to grow, SSD architectures are rapidly evolving to deliver improved performance.

For example, a major innovation has been the introduction of new host interfaces to the SSD. In the past, many SSDs made use of the Serial Advanced Technology Attachment (SATA) protocol [67], which was originally designed for HDDs. Over time, SATA has proven to be inefficient for SSDs, as it cannot enable the fast I/O accesses and millions of I/O operations per second (IOPS) that contemporary SSDs are capable of delivering. New protocols such as NVMe [63] overcome these barriers as they are designed specifically for the high throughput available in SSDs. NVMe enables high throughput and low latency for I/O requests through its use of the *multi-queue SSD* (MQ-SSD) concept. While SATA exposes only a single request port to the OS, MQ-SSD protocols provide *multiple* request queues to *directly expose* applications to the SSD device controller. This allows (1) an application to bypass OS intervention for I/O request processing, and (2) the SSD controller to schedule I/O requests based on how busy the SSD's resources are. As a result, the SSD can make higher-performance I/O request scheduling decisions.

As SSDs and their associated protocols evolve to keep pace with changing system demands, the research community needs simulation tools that reliably model these new features. Unfortunately, state-of-the-art SSD simulators do *not* model a number of key properties of modern SSDs *that are already on the market*. We evaluate several real modern SSDs, and find that state-of-the-art simulators do *not* capture three features that are critical to *accurately* model modern SSD behavior.

First, these simulators do not correctly model the *multi-queue approach used in modern SSD protocols*. Instead, they implement only the single-queue approach used in HDD-based protocols such as SATA. As a result, existing simulators do not capture (1) the high amount of request-level parallelism and (2) the lack of OS intervention in modern SSDs.

Second, many simulators do not adequately model *steady-state behavior* within a reasonable amount of simulation time. A number of fundamental SSD maintenance algorithms, such as garbage collection [11–13, 23], are *not* executed when an SSD is new (i.e., no data has been written to the drive). As a result, manufacturers design these maintenance algorithms to work best when an SSD reaches the steady-state operating point (i.e., after all of the pages within the SSD have been written to at least once) [71]. However, simulators that cannot capture steady-state behavior (within a reasonable

simulation time) perform these maintenance algorithms on a new SSD. As such, many existing simulators do *not* adequately capture algorithm behavior under realistic conditions, and often report unrealistic SSD performance results (as we discuss in Section 3.2).

Third, these simulators do not capture the full *end-to-end latency* of performing I/O requests. Existing simulators capture only the part of the request latency that takes place during intra-SSD operations. However, many emerging high-speed non-volatile memories greatly reduce the latency of intra-SSD operations, and, thus, the uncaptured parts of the latency now make up a significant portion of the overall request latency. For example, in Intel Optane SSDs, which make use of 3D XPoint memory [9, 25], the overhead of processing a request and transferring data over the system I/O bus (e.g., PCIe) is much higher than the memory access latency [16]. By not capturing the full end-to-end latency, existing simulators do not report the true performance of SSDs with new and emerging memory technologies.

Based on our evaluation of real modern SSDs, we find that these three features are essential for a simulator to capture. Because existing simulators do not model these features adequately, their results deviate significantly from the performance of real SSDs. **Our goal** in this work is to develop a new SSD simulator that can *faithfully* model the features and performance of both modern multi-queue SSDs and conventional SATA-based SSDs.

To this end, we introduce *MQSim*, a new simulator that provides an accurate and flexible framework for evaluating SSDs. MQSim addresses the three shortcomings we found in existing simulators, by (1) providing detailed models of both conventional (e.g., SATA) and modern (e.g., NVMe) host interfaces; (2) accurately and quickly modeling steady-state SSD behavior; and (3) measuring the full end-to-end latency of a request, from the time an application enqueues a request to the time the request response arrives at the host. To allow MQSim to adapt easily to future SSD developments, we employ a modular design for the simulator. Our modular approach allows users to easily modify the implementation of a single component (e.g., I/O scheduler, address mapping) without the need to change other parts of the simulator. We provide two execution modes for MQSim: (1) standalone execution, and (2) integrated execution with the gem5 full-system simulator [8]. We validate the performance reported by MQSim using several real SSDs. We find that the response time results reported by MQSim are very close to the response times of the real SSDs, with an average (maximum) error of only 11% (18%) for real storage workload traces.

By faithfully modeling the major features found in modern SSDs, MQSim can uncover several issues that existing simulators are unable to demonstrate. One such issue is the performance impact of *inter-flow interference* in modern MQ-SSDs. For two or more concurrent *flows* (i.e., streams of I/O requests from multiple applications), there are three major sources of interference: (1) the write cache, (2) the mapping table, and (3) the I/O scheduler. Using MQSim, we find that inter-flow interference leads to significant *unfairness* (i.e., the interference slows down each flow unequally) in modern SSDs. This is a major concern, as fairness is a first-class design goal in modern computing platforms [4, 17, 19, 31, 37, 56–60, 66, 73–76, 80, 84, 88]. Unfairness reduces the predictability of the I/O latency and throughput for each flow, and can allow a malicious flow to deny or delay I/O service to other, benign flows.

We have made MQSim available as an open source tool to the research community [1]. We hope that MQSim enables researchers to explore directions in several new and different areas.

We make the following key contributions in this work:

- We use real off-the-shelf SSDs to show that state-of-the-art SSD simulators do *not* adequately capture three important properties of modern SSDs: (1) the multi-queue model used by modern host–interface protocols such as NVMe, (2) steady-state SSD behavior, and (3) the end-to-end I/O request latency.
- We introduce MQSim, a simulator that accurately models both modern NVMe-based and conventional SATA-based SSDs. To our knowledge, MQSim is the first publicly-available SSD simulator to faithfully model the NVMe protocol. We validate the results reported by MQSim against several real state-of-the-art multi-queue SSDs.
- We demonstrate how MQSim can uncover important issues in modern SSDs that existing simulators *cannot* capture, such as the impact of inter-flow interference on fairness and system performance.

2 Background

In this section, we provide a brief background on multi-queue SSD (MQ-SSD) devices. First, we discuss the internal organization of an MQ-SSD (Section 2.1). Next, we discuss host–interface protocols commonly used by SSDs (Section 2.2). Finally, we discuss how the SSD flash translation layer (FTL) handles requests and performs maintenance tasks (Section 2.3).

2.1 SSD Internals

Modern MQ-SSDs are typically built using NAND flash memory chips. NAND flash memory [11, 12] supports read and write operations at the granularity of a flash *page* (typically 4 kB). Inside the NAND flash chips, multiple pages are grouped together into a flash *block*, which is the granularity at which erase operations take place. Flash writes can take place only to pages that are erased (i.e., *free*). To minimize the write latency, MQ-SSDs perform *out-of-place* updates (i.e., when a logical page is updated, its data is written to a different, free physical page, and the logical-to-physical mapping is updated). This avoids the need to erase the old physical page during a write operation. Instead, the old page is marked as *invalid*, and a *garbage collection* procedure [11–13, 23] reclaims invalid physical pages in the background.

Figure 1 shows the internal organization of an MQ-SSD. The components inside the MQ-SSD are divided into two groups: (1) the *back end*, which includes the memory devices; and (2) the *front end*, which includes the control and management units. The memory devices (e.g., NAND flash memory [11, 12], phase-change

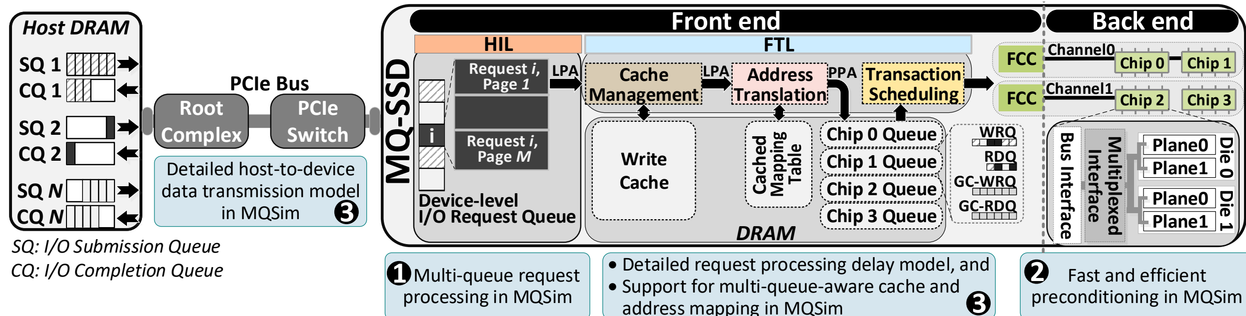


Figure 1: Organization of an MQ-SSD. As highlighted in the figure (❶, ❷, ❸), our MQSim simulator captures several aspects of MQ-SSDs not modeled by existing simulators.

memory [42], STT-MRAM [40], 3D XPoint [9]) in the back end are organized in a highly-hierarchical manner to maximize I/O concurrency. The back end contains multiple independent *bus channels*, which connect the memory devices to the front end. Each channel connects to one or more memory *chips*. For a NAND flash memory based SSD, each NAND flash chip is typically divided into multiple *dies*, where each die can independently execute memory commands. All of the dies within a chip share a common communication interface. Each die is made up of one or more *planes*, which are arrays of flash cells. Each plane contains multiple blocks. Multiple planes within a single die can execute memory operations in parallel *only* if each plane is executing the same command on the same address offset within the plane.

In an MQ-SSD, the front end includes three major components [47]. (1) The *host–interface logic* (HIL) implements the protocol used to communicate with the host (Section 2.2). (2) The *flash translation layer* (FTL) manages flash resources and processes I/O requests (Section 2.3). (3) The *flash chip controllers* (FCCs) send commands to and transfer data to/from the memory chips in the back end. The front end contains on-board DRAM, which is used by the three components to cache application data and store data structures for flash management.

2.2 Host–Interface Logic

The HIL plays a critical role in leveraging the internal parallelism of the NAND flash memory to provide higher I/O performance to the host. The SATA protocol [67] is commonly used for conventional SSDs, due to widespread support for SATA on enterprise and client systems. SATA employs Native Command Queuing (NCQ), which allows the SSD to concurrently execute I/O requests. NCQ allows the SSD to schedule multiple I/O requests based on which back end resources are currently idle [29, 50].

The NVM Express (NVMe) protocol [63] was designed to alleviate the bottlenecks of SATA [90], and to enable scalable, high-bandwidth, and low-latency communication over the PCIe bus. When an application issues an I/O request in NVMe, it bypasses the I/O stack in the OS and the block layer queue, and instead directly inserts the request into a *submission queue* (SQ in Figure 1) dedicated to the application. The SSD then selects a request from the SQ, performs the request, and inserts the request's job completion information (e.g., ack, read data) into the request *completion queue* (CQ) for the corresponding application. NVMe has already been widely adopted in modern SSD products [30, 64, 79, 85, 86].

2.3 Flash Translation Layer

The FTL executes on a microprocessor within the SSD, performing I/O requests and flash management procedures [11, 12]. Handling an I/O request in the FTL requires four steps for an SSD using NVMe. First, when the HIL selects a request from the SQ, it inserts the request into a device-level queue. Second, the HIL breaks the request down into multiple *flash transactions*, where each transaction is at the granularity of a single page. Next, the FTL checks if the request is a write. If it is, and the MQ-SSD supports *write caching*, the *write cache management unit* stores the data for each transaction in the write cache space within DRAM, and asks the HIL to prepare a response. Otherwise, the FTL *translates* the logical page address (LPA) of the transaction into a physical page address (PPA), and enqueues the transaction into the corresponding *chip-level queue*. There are separate queues for reads (RDQ) and for writes (WRQ). The *transaction scheduling unit* (TSU) resolves resource contention among the pending transactions in the chip-level queue, and sends transactions that can be performed to its corresponding FCC [20, 78]. Finally, when all transactions for a request finish, the FTL asks the HIL to prepare a response, which is then delivered to the host.

The *address translation module* of the FTL plays a key role in implementing out-of-place updates. When a transaction writes to an LPA, a *page allocation scheme* assigns the LPA to a free PPA. The LPA-to-PPA mapping is recorded in a *mapping table*, which is stored within the non-volatile memory and cached in DRAM (to reduce the latency of mapping lookups) [24]. When a transaction reads from an LPA, the module searches for the LPA's mapping and retrieves the PPA.

The FTL is also responsible for memory wearout management (i.e., *wear-leveling*) and *garbage collection* (GC) [11–13, 23]. GC is triggered when the number of free pages drops below a threshold. The GC procedure reclaims invalidated pages, by selecting a candidate block with a high number of invalid pages, moving any valid pages in the block into a free block, and then erasing the candidate block. Any read and write transactions

generated during GC are inserted into dedicated read (GC-RDQ) and write (GC-WRQ) queues. This allows the transaction scheduling unit to schedule GC-related requests during idle periods.

3 Simulation Challenges for Modern MQ-SSDs

In this section, we compare the capabilities of state-of-the-art SSD simulators to the common features of the modern SSD devices. As shown in Figure 1, we identify three significant features of modern SSDs that are *not* supported by current simulation tools: ❶ multi-queue support, ❷ fast modeling of steady-state behavior, and ❸ proper modeling of the end-to-end request latency. While some of these features are also present in some conventional SSDs, their lack of support in existing simulators is more critical when we evaluate modern and emerging MQ-SSDs, resulting in large deviations between simulation results and measured performance.

3.1 Multi-Queue Support

A fundamental difference of a modern MQ-SSD from a conventional SSD is its use of multiple queues that directly expose the device controller to applications [90]. For conventional SSDs, the OS I/O scheduler coordinates concurrent accesses to the storage devices and ensures fairness for co-running applications [66, 68]. MQ-SSDs eliminate the OS I/O scheduler, and are themselves responsible for fairly servicing I/O requests from concurrently-running applications and guaranteeing high responsiveness. Exposing application-level queues to the storage device enables the use of many optimized management techniques in the MQ-SSD controller, which can provide high performance and a high level of both fairness and responsiveness. This is mainly due to the fact that the device controller can make better scheduling decisions than the OS, as the device controller knows the current status of the SSD's internal resources.

We investigate how the performance of a flow[1] changes when the flow is concurrently executed with other flows on real MQ-SSDs. We conduct a set of experiments where we control the intensity of synthetic workloads that run on four new off-the-shelf MQ-SSDs released between 2016 and 2017 (see Table 4 and Appendix A). In each experiment, there are two flows, `Flow-1` and `Flow-2`, where each flow always keeps its I/O queue full with only sequential read accesses of 4 kB average request size. We control the intensity of a flow by adjusting its I/O queue depth. A deeper I/O queue results in a more intensive flow. We hold the I/O queue depth of `Flow-1` constant in all experiments, setting it to 8 requests. We sweep eight different values for the I/O queue depth of `Flow-2`, ranging from 8 to 1024 requests.

To quantify the I/O service fairness of each device, we measure the average *slowdown* of each executed flow, and then use the slowdown to calculate *fairness* using Equation 1. We define the slowdown of a flow f_i as $S_{f_i} = RT_{f_i}^{shared} / RT_{f_i}^{alone}$, where $RT_{f_i}^{shared}$ is the response time of f_i when it is run concurrently with other flows, and $RT_{f_i}^{alone}$ is the response time of f_i when it runs alone. Fairness (F) is calculated as [22, 56, 58]:

$$F = \frac{\underset{i}{MIN}\{S_{f_i}\}}{\underset{i}{MAX}\{S_{f_i}\}} \quad (1)$$

According to the above definition: $0 < F \leq 1$. Lower F values indicate higher differences between the minimum and maximum slowdowns of all concurrently-running flows, which we say is more unfair to the flow that is slowed down the most. Higher F values are desirable.

Figure 2 depicts the slowdown, normalized throughput (IOPS), and fairness results when we execute `Flow-1` and `Flow-2` concurrently on our four target MQ-SSDs (which we call SSD-A, SSD-B, SSD-C, and SSD-D). The x-axes in all of the plots in Figure 2 represent the queue depth (i.e., the flow intensity) of `Flow-2` in the experiments. For each SSD, we show three plots from left to right: (1) the slowdown and normalized throughput of `Flow-1`, (2) the slowdown and normalized throughput of `Flow-2`, and (3) fairness.

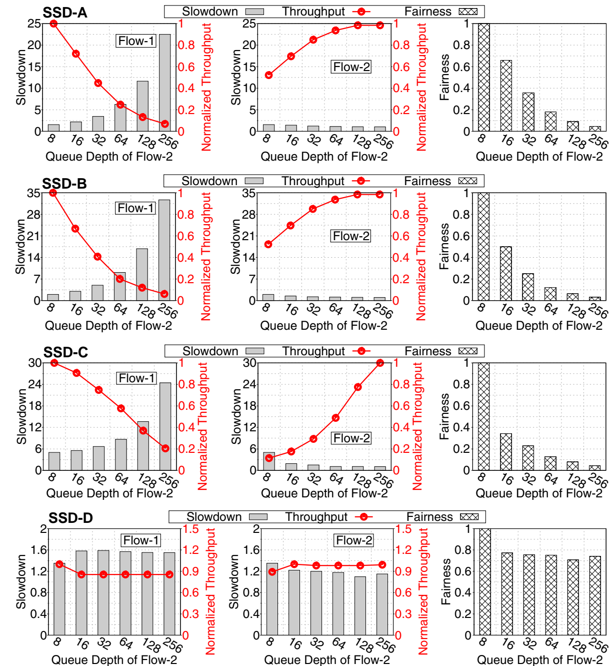


Figure 2: Performance of `Flow-1` (left) and `Flow-2` (center), and fairness (right), when flows are concurrently executed with different intensities on four real MQ-SSDs.

We make four major observations from Figure 2. First, in SSD-A, SSD-B, and SSD-C, the throughput of `Flow-2` substantially increases proportionately with the queue depth. Aside from the maximum bandwidth available from the SSD, there is no limit on the throughput of each I/O flow. Second, `Flow-1` is slowed down significantly due to interference from `Flow-2` when the I/O queue depth of `Flow-2` is much greater than that of `Flow-1`. Third, for SSD-A, SSD-B, and SSD-C, the slowdown of `Flow-2` becomes almost negligible (i.e., its

[1] We assume that each *I/O flow* uses a separate I/O queue.

value approaches 1) as the intensity of `Flow-2` increases. Fourth, SSD-D limits the maximum throughput of each flow, and thus the negative impact of `Flow-2` on the performance of `Flow-1` is well controlled. Further experiments with a higher number of flows reveal that one flow cannot exploit more than a quarter of the full I/O bandwidth of SSD-D, indicating that SSD-D has some level of internal fairness control. In contrast, one flow can unfairly exploit the full I/O capabilities of the other three SSDs.

We conclude that (1) the relative intensity of each flow significantly impacts the throughput delivered to each flow; and (2) MQ-SSDs with fairness controls, such as SSD-D, perform differently from MQ-SSDs without fairness controls when the relative intensities of concurrently-running flows differ. Thus, to accurately model the performance of MQ-SSDs, an SSD simulator needs to model multiple queues and enable multiple concurrently-running flows.

3.2 Steady-State Behavior

SSD performance evaluation standards explicitly clarify that the SSD performance should be reported in the steady state [71].[2] As a consequence, *pre-conditioning* (i.e., quickly reaching steady state) is an essential requirement for SSD device performance evaluation, in order to ensure that the results are collected in the steady state. This policy is important for three reasons. First, the garbage collection (GC) activities are invoked only when the device has performed a certain number of writes, which causes the number of free pages in the SSD to drop below the GC threshold. GC activities interfere with user I/O activity and can significantly affect the sustained device performance. However, a fresh out-of-the-box (FOB) device is unlikely to execute GC. Hence, performance results on an FOB device are unrealistic as they would *not* account for GC [71]. Second, the steady-state benefits of the write cache may be lower than the short-term benefits, particularly for write-heavy workloads. More precisely, in the steady state, the write cache is filled with application data and warmed up, and it is highly likely that no free slot can be allocated to new write requests. This leads to cache evictions and increased flash write traffic in the back end [33]. Third, the physical data placement of currently-running applications is highly dependent on the device usage history and the data placement of previous processes. For example, which physical pages are currently free in the SSD depends on how previous I/O requests wrote to and invalidated physical pages. As a result, channel- and chip-level parallelism in SSDs is limited in the steady state.

[2] Based on the SNIA definition [71], a device is in the *steady state* if its performance variation is limited to a deterministic range.

Although a number of works do successfully precondition and simulate steady-state behavior, many previous studies have not explored the effect of steady-state behavior on their proposals. Instead, their simulations start with an FOB SSD, and never reach steady state (e.g., when each physical page of the SSD has been written to at least once). Most well-known storage traces are *not* large enough to fill the entire storage space of a modern SSD. Figure 3 shows the total write volume of popular storage workloads [6, 53–55, 61]. We observe that most of the workloads have a total write volume that is much smaller than the storage capacity of most SSDs, with an average write volume of 60 GB. Even for the few workloads that are large enough to fill the SSD, it is time consuming for many existing simulators to simulate each I/O request and reach steady state (see Section 5). Therefore, it is crucial to have a simulator that enables efficient and high-performance steady-state simulation of SSDs.

3.3 Real End-to-End Latency

Request latency is a critical factor of MQ-SSD performance, since it affects how long an application stalls on an I/O request. The end-to-end latency of an I/O request, from the time it is inserted into the host submission queue to the time the response is sent back from the MQ-SSD device to the completion queue, includes seven different parts, as we show in Figure 4. Existing simulation tools model only some parts of the end-to-end latency, which are usually considered to be the dominant parts of the end-to-end latency [3, 26, 27, 35, 38].

Figure 4a illustrates the end-to-end latency diagram for a small 4 kB read request in a typical NAND flash-based MQ-SSD. It includes I/O job enqueuing in the submission queue (SQ) ❶, host-to-device I/O job transfer over the PCIe bus ❷, address translation and transaction scheduling in the FTL ❸, read command and address transfer to the flash chip ❹, flash chip read ❺, read data transfer over the Open NAND Flash Interface (ONFI) [65] bus ❻, and device-to-host read data transfer over the PCIe bus ❼. Steps ❺ and ❻ are *assumed* to be the most time-consuming parts in the end-to-end request processing. Considering typical latency values for an 8 kB page read operation, the I/O job insertion (< 1 µs, as measured on our real SSDs), the FTL request processing on a multicore processor (1 µs) [47] (assuming a mapping table cache hit), and the I/O job and data transfer over the PCIe bus (4 µs) [41, 46] make negligible contributions compared to the flash read (50-110 µs) [49, 51, 52, 69] and the ONFI NV-DDR2 [65] flash transfer (20 µs).

However, the above assumption is unrealistic due to two major reasons. First, for some I/O requests, FTL re-

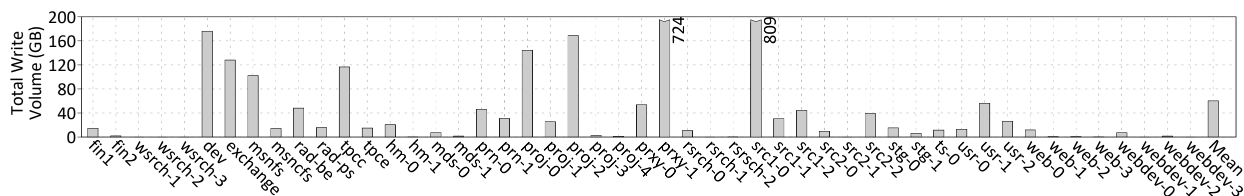
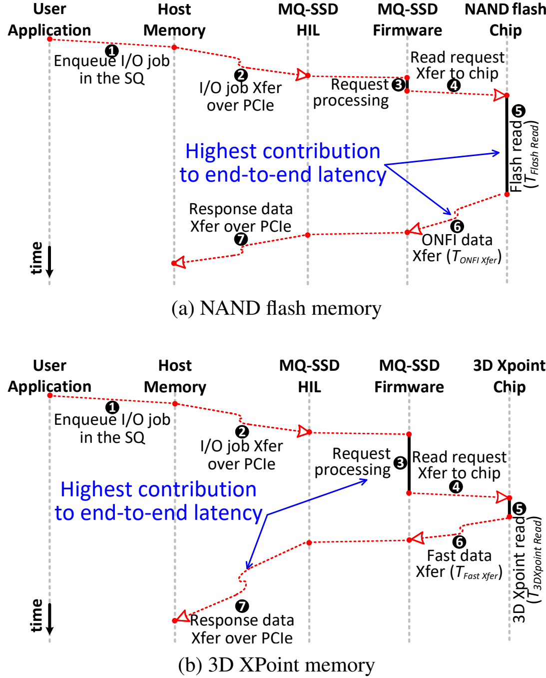


Figure 3: Total amount of data written by commonly-used storage workloads [6, 53–55, 61].



(a) NAND flash memory

(b) 3D XPoint memory

Figure 4: Timing diagram for a 4 kB read request in (a) NAND-flash and (b) 3D XPoint MQ-SSDs.

quest processing may *not* always be negligible, and can even become comparable to the flash read access time. For example, prior work [26] shows that if the FTL uses page-level address mapping, then a workload without locality incurs a large number of misses in the cached mapping table (CMT). In case of a miss in the CMT, the user read operation stalls until the mapping data is read from the SSD back end and transferred to the front end [24]. This can lead to a substantial increase in the latency of Step ❸ in Figure 4a, which can become even longer than the combined latency of Steps ❺ and ❻. In an MQ-SSD, as a greater number of I/O flows execute concurrently, there is more contention for the CMT, leading to a larger number of CMT misses.

Second, as shown in Figure 4b, cutting-edge non-volatile memory technologies, such as 3D XPoint [7, 9, 16, 48], dramatically reduce the access and data transfer times of the MQ-SSD back end, by as much as three orders of magnitude compared to that of NAND flash [25, 40, 42, 43]. The total latency of the 3D XPoint read and transfer ($< 1\,\mu s$) contributes less than 10% to the end-to-end I/O request processing latency ($<10\,\mu s$) [7, 16]. In this case, a conventional simulation tool would be inaccurate, as it does *not* model the major steps contributing to the end-to-end latency.

In summary, a detailed, realistic model of end-to-end latency is key for accurate simulation of modern SSD devices with (1) multiple I/O flows that can potentially lead to a significant increase in CMT (cached mapping table) misses, and (2) very-fast NVM technologies such as 3D XPoint that greatly reduce raw memory read/write latencies. Existing simulation tools do not provide accurate performance results for such devices.

4 Modeling a Modern MQ-SSD with MQSim

To our knowledge, there is no SSD modeling tool that supports multi-queue I/O execution, fast and efficient modeling of the SSD's steady-state behavior, and a full end-to-end request latency estimation. In this work, we present MQSim, a new simulation framework that is developed from scratch to support all of these three important features that are required for accurate performance modeling and design space exploration of modern MQ-SSDs. Although mainly designed for MQ-SSD simulation, MQSim also supports simulation of the conventional SATA-based SSDs that implement native command queuing (NCQ). Our new simulator models all of the components shown in Figure 1, which exist in modern SSDs. Table 1 provides a quick comparison between MQSim and previous SSD simulators.

MQSim is a discrete-event simulator written in C++ and is released under the permissive MIT License [1]. Figure 5 depicts a high-level view of MQSim's main components and their interaction. In this section, we briefly describe these components and explain their novel features with respect to the previous simulators.

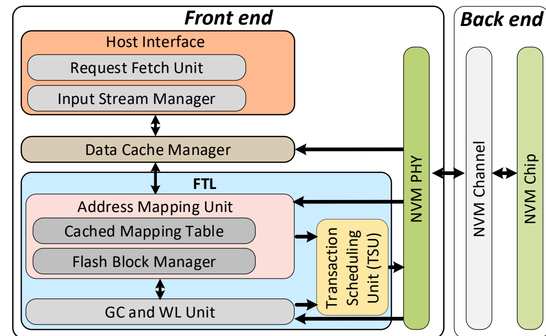


Figure 5: High-level view of MQSim components.

4.1 SSD Back End Model

MQSim provides a simple yet detailed model of the flash memory chips. It considers three major latency components of the SSD back end: (1) address and command transfer to the memory chip; (2) flash memory read/

Table 1: A quick comparison between MQSim and existing SSD modeling tools.

Tool	Multi-Queue Support	Preconditioning	End-to-end Latency	Built-in Implementation of SSD Components
MQSim	Multi-queue scheduling and prioritization	Fast and automatic (enabled by default)	Detailed model of the end-to-end latency	All major components that exist in modern SSDs
Existing Tools	Not supported	Manual, optional, and long execution time	Missing some constant- or variable-latency components	Implementation is missing for some major components

write execution for different technologies that store 1, 2, or 3 bits per cell [32]; and (3) data transfer to/from memory chips. MQSim's flash model considers the constraints of die- and plane-level parallelism, and advanced command execution [65]. One important new feature of MQSim is that it can be configured or easily modified to simulate new NVM chips (e.g., those that do not need erase-before-write). Due to decoupling of the NVM chip communication interface from the chip's internal implementation of the memory operations, one can modify the NVM chip of MQSim without the need to change the implementation of the other MQSim components.

Another new feature of MQSim is that it decouples the sizes of read and write operations. This feature helps to exploit large page sizes of modern flash memory chips in that can enable better write performance, while preventing the negative effects of large page sizes on read performance. For flash chip *writes*, the operation is always page-sized [11, 12]. MQSim's data cache controller can delay writes to eliminate write-back of partially-updated logical pages (where the update size is smaller than the physical page size). When a partially-updated logical page should be written back to the flash storage, the unchanged sub-pages (sectors) of the logical page are first read from the physical page that stores page data. Then, unchanged and updated pieces of the page are merged. In the last step, the entire page data is written to a new free physical page. For flash chip *reads*, the operation could be smaller than the physical page size. When a read operation finishes, only the data pieces that are requested in the I/O request are transferred from flash chips to the SSD controller, avoiding the data transfer overhead of large physical pages.

4.2 SSD Front End Model

The front end model of MQSim includes all of the basic components of a modern SSD controller and provides many new features that do not exist in previous SSD modeling tools.

4.2.1 Host–Interface Model

The host interface component of MQSim provides both NVMe multi-queue (MQ) and SATA native command queue models for a modern SSD. To our knowledge, MQSim is the first modeling tool that supports MQ I/O request processing. There is a request fetch unit within the host interface of MQSim that fetches and schedules application I/O requests from different input queues. The NVMe host interface provides users with a parameter, called `QueueFetchSize`, that can be used to tune the behavior of the request fetch unit, in order to accurately model the behavior of real MQ-SSDs. This parameter defines the maximum number of I/O requests from each SQ that can be concurrently serviced in the MQ-SSD. More precisely, at any given time, the number of I/O requests that are fetched from a host SQ to the device-level queue is always less than or equal to `QueueFetchSize`. This parameter has a large impact on the MQ-SSD multi-flow request processing characteristics discussed in Section 3.1 (i.e., on maximum achievable throughput per I/O flow and probability of inter-flow interference). Appendix A.3 analyzes the effect of this parameter on performance.

MQSim also models different priority classes for host-side request queues, which are part of the NVMe standard specification [63].

4.2.2 Data Cache Manager

MQSim has a data cache manager component that implements a DRAM-based cache with the least-recently-used (LRU) replacement policy. The DRAM cache can be configured to cache (1) recently-written data (default mode), (2) recently-read data, or (3) both recently-written and recently-read data. A new feature of MQSim's cache manager, compared to previous SSD modeling tools, is that it implements a DRAM access model in which the contention among the concurrent accesses to DRAM chips and the latency of DRAM commands are considered. The DRAM cache models in MQSim can be extended to make use of detailed and fast DRAM simulators, such as Ramulator [2, 39], to perform detailed studies of the effect of DRAM cache performance on the overall MQ-SSD performance. We leave this to future work.

4.2.3 FTL Components

MQSim implements all the main FTL components, including (1) the address translation unit, (2) the garbage collection (GC) and wear-leveling (WL) unit, and (3) the transaction scheduling unit. MQSim provides different options for each of these components, including state-of-the-art address translation strategies [24, 78], GC candidate block selection algorithms [10, 18, 23, 45, 81, 91], and transaction scheduling schemes [34, 87]. MQSim also implements several state-of-the-art GC and flash management mechanisms, including preemptible GC I/O scheduling [44], intra-plane data movement from one physical page to another physical page using copyback read and write command pairs [27], and program/erase suspension [87] to reduce the interference of GC operations with application I/O requests. One novel feature of MQSim is that all of its FTL components support multi-flow (i.e., multi-input queue) request processing. For example, the address mapping unit can partition the cached mapping table space among the concurrently running flows. This inherent support of multi-queue-aware request processing facilitates the design space exploration of performance isolation and QoS schemes for MQ-SSDs.

4.3 Modeling End-to-End Latency

In addition to the flash operation and internal data transfer latency (steps ❸, ❹, ❺, and ❻ in Figure 4), there is a mix of variable and constant latencies that MQSim models to determine the end-to-end request latency.

Variable Latencies. These are the variable request processing times in FTL that result from contention in the cached mapping table and the DRAM write cache. Depending on the request type (either read or write) and the request's logical address, the request processing time in FTL includes some of the following items: (1) the time required to read/write from/to the data cache, and (2) the

time to fetch mapping data from flash storage in case of a miss in the cached address mapping table.

Constant Latencies. These include the times required to transmit the I/O job information, the entire user data, and the I/O completion information over the PCIe bus, and the firmware (FTL) execution time on the controller's microprocessor. The PCIe transmission latencies are calculated based on a simple packet latency model provided by Xilinx [41] that considers: (1) the PCIe communication bandwidth, (2) the payload and header sizes of the PCIe Transaction Layer Packets (TLP), (3) the size of the NVMe management data structures, and d) the size of the application data. The firmware execution time is estimated using both a CPU and cache latency model [1].

4.4 Modeling Steady-State Behavior

The basic assumption of MQSim is that *all* simulations should be executed when the modeled device is in steady state. To model the steady-state behavior, MQSim, *by default*, *automatically* executes a preconditioning function before starting the actual simulation process. This function performs preconditioning in a short time (e.g., less than 8 min when running `tpcc` [53] on an 800 GB MQ-SSD) without the need to execute additional I/O requests. During preconditioning, all available physical pages of the modeled SSD are transitioned to either a valid or invalid state, based on the steady-state valid/invalid page distribution model provided in [82] (only very few flash blocks are assumed to remain free and are added to the free block pool). MQSim pre-processes the input trace to extract the LPA (logical page address) access characteristics of the application I/O requests in the trace, and then uses the extracted information as inputs to the valid/invalid page distribution model. In addition, input trace characteristics, such as the average write arrival rate and the distribution of write addresses, are used to warm up the write cache.

4.5 Execution Modes

MQSim provides two modes of operation: (i) *standalone* mode, where it is fed a real disk trace or a synthetic workload, and (ii) *integrated* mode, where it is fed disk requests from an execution-driven engine (e.g., gem5 [8]).

5 Comparison with Previous Simulators

The increasing usage of SSDs in modern computing systems has boosted interest in SSD design space exploration. To this end, several simulators have been developed in recent years. Table 2 summarizes the features of MQSim and popular existing SSD modeling tools. The table also shows the *average error rates* for the performance of real storage workloads reported by each simulator, compared to the performance measured on four real MQ-SSDs (see Appendix A.1 for our methodology).

Existing tools either do not model some major components of modern SSDs or provide very simplistic component models that lead to unrealistic I/O request latency estimation. In contrast, MQSim provides detailed implementations for all of the major components of modern SSDs. MQSim is written in C++ and has 13K lines of code (LOC). Next, we discuss the main advantages of MQSim compared to the previous tools.

Host–Interface Logic. As Table 2 shows, most of the existing simulators assume a very simplistic HIL model with no explicit management mechanism for the I/O request queue. This leads to an unrealistic SSD model regarding the requirements of both NVMe and SATA protocols. As we mention in Section 3, the concurrent execution of I/O flows presents many challenges for performance predictability and fairness in MQ-SSDs. No ex-

Table 2: Comparison of MQSim with previous SSD modeling tools.

Simulator	HIL Protocol		Execution Mode				End-to-End Latency					Front-End Components							**LOC**[14]	Simulation Error (%)			
	NVMe MQ	SATA NCQ	Alone[1]	Full[2]	Emul[3]	Prec[4]	NVM R/W[5]	NVM Xfer	FTL Proc[6]	Cache Acc.[7]	Host Xfer[8]	Map P[9]	Map H[10]	GC	Write Cache	TSU[11]	WRL[12]	MQ FTL[13]		SSD-A	SSD-B	SSD-C	SSD-D
MQSim	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	13K	8	6	18	14
SSDModel [3]			✓				✓	✓			✓	✓		✓			✓		1K	91	155	196	136
FlashSim [38]			✓				✓	✓			✓	✓	✓	✓			✓		8K	99	259	310	138
SSDSim [27]			✓				✓	✓				✓		✓	✓				5K	70	68	74	85
NANDFlashSim [32]							✓	✓											7K	–	–	–	–
VSSIM [92]					✓		✓	✓				✓	✓	✓	✓		✓		6K	–	–	–	–
WiscSim [26]		✓	✓				✓	✓	✓			✓	✓	✓			✓		7K	95	277	324	135
SimpleSSD [35]			✓	✓			✓	✓					✓	✓		✓	✓		7K	–	–	–	–

[1] Standalone execution [2] Integrated execution with full-system simulator [3] SSD emulation for real system
[4] Fast and accurate preconditioning of the modeled SSD to enable accurate steady-state results
[5] Flash (NVM) read/write timing [6] FTL request processing overhead [7] Accurate modeling of write cache access latency
[8] Host-to-device and device-to-host data transfer delay [9] Page-level address mapping [10] Hybrid address mapping
[11] FTL transaction scheduling unit [12] FTL wear-leveling unit [13] Built-in support for multi-queue-aware request processing in FTL
[14] Lines of source code

isting simulator implements NVMe and multi-queue I/O request management, and, hence, accurately models the behavior of MQ-SSDs. Also, except for WiscSim, we find that no existing simulator implements an accurate model of the SATA protocol and NCQ request processing. This leads to unrealistic SATA device simulation, as NCQ-based I/O scheduling plays a key role in the performance of real SSD devices [15, 26].

Steady-State Simulation. To our knowledge, accurate and fast steady-state behavior modeling is not provided by many existing SSD modeling tools. Among the tools listed in Table 2, only SSDSim provides a function, called `make_aged`, to change the status of a set of physical pages to valid before starting the actual execution of an input trace. This simple method *cannot* accurately replicate the steady-state behavior of an SSD for two reasons. First, after the execution of `make_aged`, the physical blocks would include only valid pages or only free pages. This is far from the steady-state status of blocks in real devices, where each non-free block has a mix of valid and invalid pages [28, 81, 82]. Second, the steady-state status of the data cache is *not* modeled, i.e., the simulation starts with a completely empty write cache.

In general, it *is* possible to bring these simulators to steady state. However, there is no *fast* pre-conditioning support for them, and pre-conditioning *must* be performed by executing traces. Preconditioning an existing simulator requires users to generate traces with a large enough number of I/O requests, and can significantly slow down the simulator, especially when a high-capacity SSD is modeled. For example, our studies with SSDSim show that pre-conditioning may increase the simulation time up to 80x if an 800 GB SSD is modeled.[3]

Detailed End-to-End Latency Model. As described in Section 3.3, the end-to-end latency of an application I/O request includes different components. Table 2 shows that latency modeling in existing simulators is mainly focused on the latency of the flash chip operation and the SSD internal data transfer. As we explain in Section 3.3, this is an unrealistic model of the end-to-end I/O request processing latency, even for a conventional SSD.

To study the accuracy of the existing tools in modeling real devices, we create four models for the four real SSDs shown in Table 4 in each simulator, and execute three real traces, i.e., `tpcc`, `tpce`, and `exchange`. We exclude the simulators that do *not* support trace-based execution. The four rightmost columns of Table 2 show the average error rate of each simulator in modeling the performance (i.e., read and write latency) of these four real devices. The error rates of the four evaluated simulators are almost one order of magnitude higher than that of MQSim. We believe that these high error rates are due to four major reasons: (1) the lack of write cache or inaccurate modeling of the write cache access latency, (2) the lack of built-in support for steady-state modeling, (3) incomplete modeling of the request processing latency in FTL, and (4) the lack of modeling of the host-to-device communication latency.

[3]The increase in simulation time depends on the access pattern, intensity, and mix of I/O requests (read vs. write) of the workload.

6 Research Directions Enabled by MQSim

MQSim is a flexible simulation tool that enables different studies on both modern and conventional SSD devices. In this section, we discuss two new research directions enabled by MQSim, which could not be explored easily using existing simulation tools. First, we use MQSim to perform a detailed analysis of inter-flow interference in a modern MQ-SSD (Section 6.1). We explain how sharing different internal resources in an MQ-SSD, such as the write cache, cached mapping table, and back end resources, can introduce fairness issues. Second, we explain how the full-system simulation mode of MQSim can enable detailed application-level studies (Section 6.2).

6.1 Design Space Exploration of Fairness and QoS Techniques for MQ-SSDs

As we describe in Section 1, fairness and QoS should be considered as first-class design criteria for modern datacenter SSDs. MQSim provides an accurate framework to study inter-flow interference, thus enables the ability to devise interference-aware MQ-SSD management algorithms for sharing of the internal MQ-SSD resources. As we show in Section 3.1, concurrently running two I/O flows might lead to disproportionate slowdowns for each flow, greatly degrading fairness and proportional progress. This is particularly important in high-end SSD devices, which provide higher throughput per I/O flow, as we show in Appendix A.3.

We find that this inter-flow interference is mainly the result of contention that takes place at three locations in an MQ-SSD: 1) the write cache in the front end, 2) the cached mapping table (CMT) in the front end, and 3) the storage resources in the back end. In this section, we use MQSim to explore the impact of these three points of contention on performance and fairness, which cannot be explored accurately using existing simulators.

6.1.1 Methodology

MQ-SSD Configuration. Table 3 lists the specification of the MQ-SSD that we model in MQSim for our contention studies.

Metrics. To measure performance, we use *weighted speedup* (WS) [70] of the average response time (RT), which represents the overall efficiency and system-level

Table 3: Configuration of the simulated SSD.

SSD Organization	Host interface: PCIe 3.0 (NVMe 1.2) User capacity: 480 GB Write cache: 256 MB, CMT: 4 MB 8 channels, 4 chips per channel `QueueFetchSize` = 512
Flash Communication Interface	ONFI 3.1 (NV-DDR2) Width: 8 bit, Rate: 333 MT/s
Flash Microarchitecture	8 KiB page, 448 B metadata per page, 256 pages per block, 2048 blocks per plane, 2 planes per die
Flash Access Parameters	Read latency: 75 µs, Program latency: 750 µs, Erase latency: 3.8 ms

throughput [21] provided by an MQ-SSD during the concurrent execution of multiple flows:

$$\text{WS} = \sum_i \frac{RT_i^{alone}}{RT_i^{shared}} \tag{2}$$

where RT_i^{alone} and RT_i^{shared} are defined in Section 3.1.

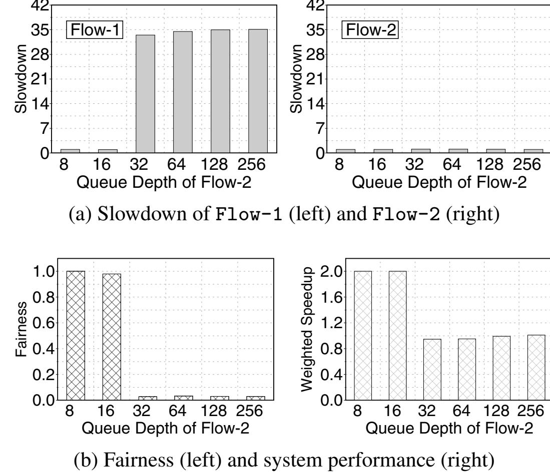
To demonstrate the effect of inter-flow interference on fairness, we report *slowdown* and *fairness (F)* metrics, as defined in Section 3.1.

6.1.2 Contention at the Write Cache

One point of contention among concurrently-running flows in an MQ-SSD is the write cache. For flows with low to moderate write intensity (where the average depth of the I/O queue less than 16), or with high spatial locality, the write cache decreases the response time of write requests, by avoiding the need for the requests to wait for the write to complete to the underlying memory. For flows with high write intensity or with highly-random accesses, the write requests fill up the limited capacity of the write cache quickly, causing significant cache thrashing and limiting the decrease in write request response time. Such flows not only do *not* benefit from the write cache themselves, but also prevent other lower-write-intensity flows from benefiting from the write cache, leading to a large performance loss for the lower-write-intensity flows.

To understand how the contention at the write cache affects system performance and fairness, we perform a set of experiments where we run two flows, `Flow-1` and `Flow-2`, both of which perform only random-access write requests. In both flows, the average request size is set to 8 kB. We set `Flow-1` to have a moderate write intensity, by limiting the queue depth to 8 requests. We vary the queue depth of `Flow-2` from 8 requests to 256 requests, to control the write intensity of the flow. In order to isolate the effect of write cache interference in our experiments, we (1) assign each flow to a dedicated subset of back end resources (i.e., `Flow-1` uses Channels 1–4, and `Flow-2` uses Channels 5–8), to avoid introducing any interference in the back end; and (2) use a perfect CMT, where all address translation requests are hits, to avoid interference due to limited CMT capacity.

Figure 6a shows the slowdown of each flow when the two flows run concurrently, compared to when each flow runs alone. Figure 6b shows the fairness and performance of the system when the two flows run concurrently. We make four key observations from the figures. First, `Flow-1` is slowed down significantly when `Flow-2` has a high write intensity (i.e., its queue depth is greater than 16), indicating that at high write intensities, `Flow-2` induces write cache trashing. Second, the slowdown of `Flow-2` is negligible, because of the low write intensity of `Flow-1`. Third, fairness degrades greatly, as a result of the write cache contention, when `Flow-2` has a high write intensity. Fourth, write cache contention causes an MQ-SSD to be inefficient at concurrently running multiple I/O flows, as the weighted speedup is reduced by over 50% when `Flow-2` has a high write intensity compared to when it has a low write intensity.



(a) Slowdown of `Flow-1` (left) and `Flow-2` (right)

(b) Fairness (left) and system performance (right)

Figure 6: Impact of write cache contention.

We conclude that write cache contention leads to unfairness and overall performance degradation for concurrently-running flows when one flow has a high write intensity. In these cases, the high-write-intensity flow (1) does *not* benefit from the write cache; and (2) *prevents* other, lower-write-intensity flows from taking advantage of the write cache, even though the other flows would otherwise benefit from the cache. This motivates the need for fair write cache management algorithms for MQ-SSDs that take inter-flow interference and flow write intensity into account.

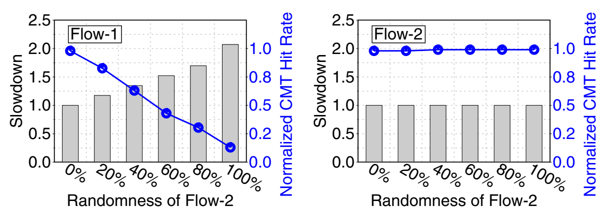
6.1.3 Contention at the Cached Mapping Table

As we discuss in Section 3.3, address translation can noticeably increase the end-to-end latency of an I/O request, especially for read requests. We find that for I/O flows with random access patterns, the cached mapping table (CMT) miss rate is high due to poor reuse of address translation mappings, which causes the I/O requests generated by the flow to stall for long periods of time during address translation. This is not true for I/O flows with sequential accesses, for which the CMT miss rate remains low due to spatial locality. However, when two I/O flows run concurrently, where one flow has a random access pattern and another flow has a sequential access pattern, the poor locality of the flow with the random access pattern may cause *both flows* to have high CMT miss rates.

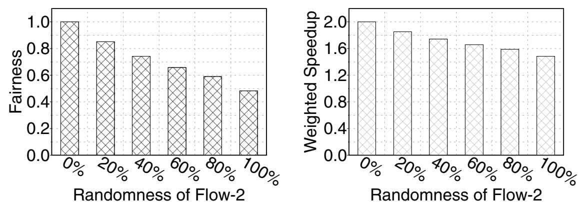
To understand how contention at the CMT affects system performance and fairness, we perform a set of experiments where we concurrently run two flows that issue read requests with an average request size of 8 kB. In these experiments, `Flow-1` has a fully-sequential access pattern, and `Flow-2` has a random access pattern for a fraction of the total execution time, and has a sequential access pattern for the remaining time. We vary the *randomness* (i.e., the fraction of the execution time with a random access pattern) of `Flow-2`. To isolate the effects of CMT contention, we assign `Flow-1` to Channels 1–4 in the back end, and assign `Flow-2` to Channels 5–8.

Figure 7a shows the slowdown and change in CMT hit rate of each flow when `Flow-1` and `Flow-2` run concur-

rently, compared to when each flow runs alone. Figure 7b shows the fairness and overall performance of the system when the two flows run concurrently. We make two observations from the figures. First, as the randomness of `Flow-2` increases, the CMT hit rate of `Flow-1` decreases, while the CMT hit rate of `Flow-2` remains constant. This indicates that the randomness of `Flow-2` introduces contention at the CMT, which hurts the CMT hit ratio of `Flow-1`. Second, as the CMT hit rate of `Flow-1` decreases, the flow experiences a greater slowdown, with a 2.1x slowdown when `Flow-2`'s access pattern is completely random. Third, as the randomness of `Flow-2` increases, both fairness and overall system performance decrease, as the interference introduced by `Flow-2` hurts the performance of `Flow-1` without providing any noticeable benefit to `Flow-2`.



(a) Slowdown and CMT hit rate (normalized to the hit rate when `Flow-2` randomness is 0%) for `Flow-1` (left) and `Flow-2` (right)



(b) Fairness (left) and system performance (right)

Figure 7: Impact of CMT contention.

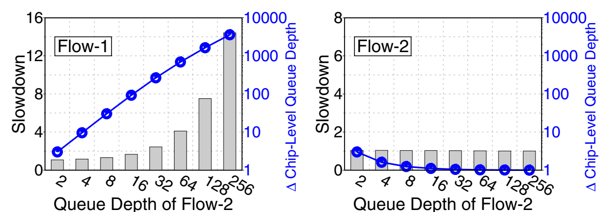
We conclude that the CMT contention induced by an I/O flow with a random access pattern disproportionately slows down concurrently-running flows with sequential access patterns, which would otherwise benefit from the CMT, leading to high unfairness and system performance degradation. To avoid such unfairness and performance loss, an MQ-SSD should use CMT management algorithms that are aware of inter-flow interference.

6.1.4 Contention at the Back End Resources

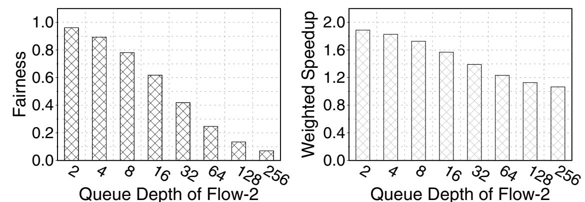
A third point of contention is at the back end resources within an MQ-SSD (see Section 2.1). A high-intensity flow can use up most of the back end resources if the flow issues a large number of requests in a short period of time. This stalls the requests issued by a low-intensity concurrently-running flow, as the requests cannot be serviced before the back end resources finish servicing requests from the high-intensity flow.

To understand how contention at the back end resources affects system performance and fairness, we perform a set of experiments where we concurrently run two I/O flows that issue random reads with a request size of 8 kB. `Flow-1` is a low-intensity I/O flow, as we limit its submission queue size (see Section 2.2) to 2 requests. We vary the submission queue size of `Flow-2` from 2 requests to 256 requests, to control the flow intensity. In order to isolate the effect of back end resource contention, we disable the write cache, and simulate a CMT where address translation requests always hit.

Figure 8a shows the slowdown when `Flow-1` and `Flow-2` run concurrently, and the change in the average chip-level queue depth (i.e., the number of requests waiting to be serviced by the back end; see Section 2.3) for each flow during concurrent execution, compared to the depth when each flow runs alone. Figure 8b shows the fairness and overall performance of the system when the two flows run concurrently. We make four observations from the figures. First, the average chip-level queue depth of `Flow-1` increases significantly when the intensity of `Flow-2` increases. Second, `Flow-1` is slows down significantly when we increase the host-side queue depth of `Flow-2` beyond 16. For example, when `Flow-2` is at the highest intensity that we test (with a host-side queue depth of 256 requests), `Flow-1` slows down by 14.4x. Third, the effect of inter-flow interference on `Flow-2` is negligible, as its slowdown is almost equal to 1 for host-side queue depths larger than 4. Fourth, the asymmetric slowdowns (i.e., the large slowdown for `Flow-1` and the lack of slowdown for `Flow-2`) cause both fairness and the overall system performance to decrease.



(a) Slowdown and average chip-level queue depth of `Flow-1` (left) and `Flow-2` (right)



(b) Fairness (left) and system performance (right)

Figure 8: Impact of back end resource contention.

We conclude that a high-intensity flow can significantly increase the depth of the chip-level queues and thus lead to a large slow-down for concurrently-running low-intensity flows. The FTL transaction scheduling unit must be aware of the inter-flow interference at the MQ-SSD back end to make the per-flow performance more fair and thus keep the overall performance high.

6.2 Application-Level Studies

To study the effect of SSD device-level design choices on application-level performance metrics, such as instructions per cycle (IPC), an SSD simulator must be integrated and run together with a full-system simulator. We integrate MQSim with gem5 [8] to provide a complete

model of multi-queue I/O execution and a complete computer system. As Table 2 shows, among existing SSD simulators, only SimpleSSD [35] is integrated with a full-system simulator, and SimpleSSD does *not* simulate multi-queue I/O execution. In this section, we show the effectiveness of our integrated simulator, by studying how changes to `QueueFetchSize` (see Section 4.2.1) affect the IPC of concurrently-executing applications due to storage-level interference.

We conduct a set of experiments, running instances of file server (`fs`) [77], mail server (`ms`) [77], web server (`ws`) [77], and IOzone large file access (`io`) [62] applications using the integrated execution mode of MQSim. We first execute each application alone (i.e., without interference from other applications), and then concurrently execute the application with a second application to study the effect of inter-application interference. To isolate the effect of inter-flow interference, where each flow belongs to one application, we assign each application to a single processor core and a single memory channel. We test two different values of `QueueFetchSize` (16 entries and 1024 entries) to examine how `QueueFetchSize` affects inter-application interference. For these experiments, we measure *application slowdown* (S_{app}), which is calculated as $S_{app_i} = IPC^{alone}_{app_i}/IPC^{shared}_{app_i}$, and use application slowdown to determine fairness using Equation 1.

Figure 9 shows the slowdown of each application and the system fairness for six pairs of concurrently-executing applications. On the x-axis, we list the applications used in each pair, along with the value of `QueueFetchSize` that we use. We make two observations from the figure. First, for application pairs where one of the applications is `ms` or `ws`, the impact of `QueueFetchSize` on fairness is negligible. Both `ms` and `ws` benefit mainly from caching a large part of their data set in main memory, and hence issue very few requests to the SSD. This keeps storage-level interference low, as `ms` and `ws` do not contend often for access to the SSD with the other applications that they are paired with. Second, `fs` and `io` have high storage access intensities, and hence interfere significantly when they are paired together. In this case, we observe that a large `QueueFetchSize` value leads to 60% fairness reduction.

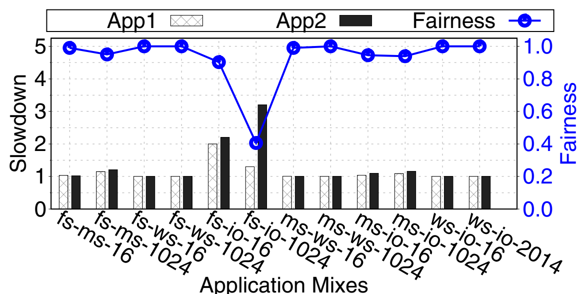


Figure 9: Application-level impact of `QueueFetchSize`.

We conclude that full-system behavior can greatly impact the fairness and performance of I/O flows on an MQ-SSD, as it affects the storage-level intensity of each flow.

7 Related Work

To our knowledge, MQSim is the first simulator that (1) accurately simulates both modern and conventional SSDs, (2) faithfully models modern host–interface protocols such as NVMe, and (3) supports the accurate simulation of SSDs that use emerging ultra-fast memory technologies. We compare MQSim to existing state-of-the-art SSD simulation tools in Section 5, and show that MQSim provides greater capabilities and accurate results. In this section, we provide a brief summary of other related works.

A number of prior works consider the performance and implementation challenges of MQ-SSDs [5, 31, 89, 90]. Xu et al. [89] analyze the effect of MQ-SSDs on the performance of modern hyper-scale and database applications. Awad et al. [5] evaluate the impact of different NVMe host-interface implementations on the system performance. Vučinić et al. [83] show that the current NVMe protocol will be a performance bottleneck in future PCM-based storage devices. The authors modify the NVMe standard in order to improve its performance for future PCM-based SSDs.

Other works [31, 72] focus on managing multiple flows in modern SSDs. Song and Yang [72] partition the SSD back end resources among concurrently-running I/O flows to provide performance isolation and alleviate inter-flow interference. Jun and Shin [31] propose a device-level scheduling technique for MQ-SSDs with built-in virtualization support.

None of these previous studies provide a simulation framework for MQ-SSDs or study the sources of inter-flow interference inside MQ-SSDs.

8 Conclusion

We introduce MQSim, a new simulator that accurately captures the behavior of both modern multi-queue SSDs and conventional SATA-based SSDs. MQSim faithfully models a number of critical features absent in existing state-of-the-art simulators, including (1) modern multi-queue-based host–interface protocols (e.g., NVMe), (2) the steady-state behavior of SSDs, and (3) the end-to-end latency of I/O requests. MQSim can be run as a standalone tool, or integrated with a full-system simulator. We validate MQSim against real off-the-shelf SSDs, and demonstrate that it provides highly-accurate results. By accurately modeling modern SSDs, MQSim can uncover important issues that cannot be modeled accurately using existing simulators, such as the impact of inter-flow interference. We have released MQSim as an open-source tool [1], and we hope that MQSim enables researchers to explore new ideas and directions.

Acknowledgments

We thank our shepherd Haryadi Gunawi and the anonymous referees for their feedback on this work. We thank our industrial partners, especially Google, Huawei, Intel, and VMware, for their generous support.

References

[1] MQSim GitHub Repository. https://github.com/CMU-SAFARI/MQSim.

[2] Ramulator GitHub Repository. https://github.com/CMU-SAFARI/ramulator.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX ATC* (2008).

[4] AUSAVARUNGNIRUN, R., CHANG, K. K.-W., SUBRAMANIAN, L., LOH, G. H., AND MUTLU, O. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA* (2012).

[5] AWAD, A., KETTERING, B., AND SOLIHIN, Y. Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems. In *ISPASS* (2015).

[6] BATES, K., AND MCNUTT, B. UMass Rrace Repository. http://traces.cs.umass.edu/.

[7] BILLI, E. How NVMe and 3D XPoint Will Create a New Datacenter Architecture. In *FMS* (2016).

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

[9] BOURZAC, K. Has Intel Created a Universal Memory Technology? *IEEE Spectrum* (2017).

[10] BUX, W., AND ILIADIS, I. Performance of Greedy Garbage Collection in Flash-Based Solid-State Drives. *Perform. Eval.* (2010).

[11] CAI, Y., GHOSE, S., HARATSCH, E. F., LUO, Y., AND MUTLU, O. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. IEEE* (2017).

[12] CAI, Y., GHOSE, S., HARATSCH, E. F., LUO, Y., AND MUTLU, O. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. arXiv:1711.11427 [cs:AR], 2017.

[13] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems. *TECS* (2004).

[14] CHEN, F., HOU, B., AND LEE, R. Internal Parallelism of Flash Memory-Based Solid-State Drives. *TOS* (2016).

[15] CHEN, F., LEE, R., AND ZHANG, X. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-Speed Data Processing. In *HPCA* (2011).

[16] COULSON, R. 3D XPoint Technology Drives System Architecture. In *SNIA Storage Industry Summit* (2016).

[17] DAS, R., MUTLU, O., MOSCIBRODA, T., AND DAS, C. R. Application-Aware Prioritization Mechanisms for On-Chip Networks. In *MICRO* (2009).

[18] DESNOYERS, P. Analytic Modeling of SSD Write Performance. In *SYSTOR* (2012).

[19] EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. N. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS* (2010).

[20] ELYASI, N., ARJOMAND, M., SIVASUBRAMANIAM, A., KANDEMIR, M. T., DAS, C. R., AND JUNG, M. Exploiting Intra-Request Slack to Improve SSD Performance. In *ASPLOS* (2017).

[21] EYERMAN, S., AND EECKHOUT, L. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* (2008).

[22] GABOR, R., WEISS, S., AND MENDELSON, A. Fairness and Throughput in Switch on Event Multithreading. In *MICRO* (2006).

[23] GAL, E., AND TOLEDO, S. Algorithms and Data Structures for Flash Memories. *CSUR* (2005).

[24] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS* (2009).

[25] HANDY, J. 3D XPoint: Speed at What Cost? In *FMS* (2017).

[26] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The Unwritten Contract of Solid State Drives. In *EuroSys* (2017).

[27] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *ICS* (2011).

[28] ILIADIS, I. Rectifying Pitfalls in the Performance Evaluation of Flash Solid-State Drives. *Perform. Eval.* (2014).

[29] INTEL CORPORATION. Intel SSD DC S3500 Series Datasheet, 2015.

[30] INTEL CORPORATION. Intel 3D NAND SSD DC P4500 Series Datasheet, 2017.

[31] JUN, B., AND SHIN, D. Workload-Aware Budget Compensation Scheduling for NVMe Solid State Drives. In *NVMSA* (2015).

[32] JUNG, M., CHOI, W., GAO, S., WILSON III, E. H., DONOFRIO, D., SHALF, J., AND KANDEMIR, M. T. NANDFlashSim: High-Fidelity, Microarchitecture-Aware NAND Flash Memory Simulation. *TOS* (2016).

[33] JUNG, M., AND KANDEMIR, M. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *SIGMETRICS* (2013).

[34] JUNG, M., AND KANDEMIR, M. T. Sprinkler: Maximizing Resource Utilization in Many-Chip Solid State Disks. In *HPCA* (2014).

[35] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. SimpleSSD: Modeling Solid State Drives for Holistic System Simulation. *CAL* (2017).

[36] KIM, J., KIM, J., PARK, P., KIM, J., AND KIM, J. SSD Performance Modeling Using Bottleneck Analysis. *CAL* (2017).

[37] KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO* (2010).

[38] KIM, Y., TAURAS, B., GUPTA, A., AND URGAONKAR, B. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *SIMUL* (2009).

[39] KIM, Y., YANG, W., AND MUTLU, O. Ramulator: A Fast and Extensible DRAM Simulator. *CAL* (2016).

[40] KÜLTÜRSAY, E., KANDEMIR, M., SIVASUBRAMANIAM, A., AND MUTLU, O. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS* (2013).

[41] LAWLEY, J. Understanding Performance of PCI Express Systems. XILINX White Paper, 2014.

[42] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA* (2009).

[43] LEE, B. C., ZHOU, P., YANG, J., ZHANG, Y., ZHAO, B., IPEK, E., MUTLU, O., AND BURGER, D. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* (2010).

[44] LEE, J., KIM, Y., SHIPMAN, G. M., ORAL, S., AND KIM, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *TC* (2013).

[45] LI, Y., LEE, P. P., AND LUI, J. Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization. In *SIGMETRICS* (2013).

[46] LIU, J., MAMIDALA, A., VISHNU, A., AND PANDA, D. K. Performance Evaluation of InfiniBand with PCI Express. In *CONNECT* (2004).

[47] MARVELL. Marvell 88SS1093 Flash Memory Controller, 2017.

[48] MICRON TECHNOLOGY, INC. Breakthrough Nonvolatile Memory Technology. `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`.

[49] MICRON TECHNOLOGY, INC. NAND Flash Memory - MT29E64G08CECBB Datasheet, 2009.

[50] MICRON TECHNOLOGY, INC. M500 2.5-Inch SATA NAND Flash SSD Series Datasheet, 2013.

[51] MICRON TECHNOLOGY, INC. NAND Flash Memory - MLC+ MT29F256G08CKCAB Datasheet, 2014.

[52] MICRON TECHNOLOGY, INC. NAND Flash Memory - MT29E128G08CBCCB Datasheet, 2016.

[53] MICROSOFT CORPORATION. Microsoft Enterprise Traces. `http://iotta.snia.org/traces/130`.

[54] MICROSOFT CORPORATION. Microsoft Production Server Traces. `http://iotta.snia.org/traces/158`.

[55] MICROSOFT CORPORATION. Microsoft Research Cambridge Traces. `http://iotta.snia.org/traces/388`.

[56] MOSCIBRODA, T., AND MUTLU, O. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security* (2007).

[57] MUTLU, O. Memory Scaling: A Systems Architecture Perspective. In *IMW* (2013).

[58] MUTLU, O., AND MOSCIBRODA, T. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO* (2007).

[59] MUTLU, O., AND MOSCIBRODA, T. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA* (2008).

[60] MUTLU, O., AND SUBRAMANIAN, L. Research Problems and Opportunities in Memory Systems. *SUPERFRI* (2015).

[61] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys* (2009).

[62] NORCOTT, W. D., AND CAPPS, D. IOzone Filesystem Benchmark, 2003.

[63] NVM EXPRESS WORKGROUP. NVM Express Specification, Revision 1.2, 2014.

[64] OCZ. RD400/400A Series Datasheet, 2016.

[65] ONFI WORKGROUP. Open NAND Flash Interface Specification, Revision 4.0, 2014.

[66] PARK, S., AND SHEN, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST* (2012).

[67] SATA-IO. Serial ATA Revision 3.3. `http://www.sata-io.org`, 2016.

[68] SHEN, K., AND PARK, S. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *USENIX ATC* (2013).

[69] SK HYNIX INC. F26 32Gb MLC NAND Flash Memory TSOP Legacy, 2011.

[70] SNAVELY, A., AND TULLSEN, D. M. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS* (2000).

[71] SNIA TECHNICAL POSITION. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise, version 1.1, 2013.

[72] SONG, X., YANG, J., AND CHEN, H. Architecting Flash-Based Solid-State Drive for High-Performance I/O Virtualization. *CAL* (2014).

[73] SUBRAMANIAN, L., LEE, D., SESHADRI, V., RASTOGI, H., AND MUTLU, O. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *ICCD* (2014).

[74] SUBRAMANIAN, L., LEE, D., SESHADRI, V., RASTOGI, H., AND MUTLU, O. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *TPDS* (2016).

[75] SUBRAMANIAN, L., SESHADRI, V., GHOSH, A., KHAN, S., AND MUTLU, O. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *MICRO* (2015).

[76] SUBRAMANIAN, L., SESHADRI, V., KIM, Y., JAIYEN, B., AND MUTLU, O. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA* (2013).

[77] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login* (2016).

[78] TAVAKKOL, A., MEHRVARZY, P., ARJOMAND, M., AND SARBAZI-AZAD, H. Performance Evaluation of Dynamic Page Allocation Strategies in SSDs. *TOMPECS* (2016).

[79] TOSHIBA CORPORATION. PX04PMB Series Datasheet, 2016.

[80] USUI, H., SUBRAMANIAN, L., CHANG, K. K.-W., AND MUTLU, O. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *TACO* (2016).

[81] VAN HOUDT, B. A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives. In *SIGMETRICS* (2013).

[82] VAN HOUDT, B. On the Necessity of Hot and Cold Data Identification to Reduce the Write Amplification in Flash-Based SSDs. *Perform. Eval.* (2014).

[83] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., LE MOAL, D., BUNKER, T., XU, J., SWANSON, S., AND BANDIĆ, Z. DC Express: Shortest Latency Protocol for Reading Phase Change Memory Over PCI Express. In *FAST* (2014).

[84] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *OSDI* (1994).

[85] WESTERN DIGITAL CORPORATION. HGST Ultrastar SN200 Series Datasheet, 2017.

[86] WESTERN DIGITAL CORPORATION. SanDisk Skyhawk & Skyhawk Ultra NVMe PCIe SSD Datasheet, 2017.

[87] WU, G., AND HE, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *FAST* (2012).

[88] XIANG, X., GHOSE, S., MUTLU, O., AND TZENG, N.-F. A Model for Application Slowdown Estimation in On-Chip Networks and Its Use for Improving System Fairness and Performance. In *ICCD* (2016).

[89] XU, Q., SIYAMWALA, H., GHOSH, M., AWASTHI, M., SURI, T., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Characterization of Hyper-Scale Applications on NVMe SSDs. In *SIGMETRICS* (2015).

[90] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *SYSTOR* (2015).

[91] YANG, M.-C., CHANG, Y.-M., TSAO, C.-W., HUANG, P.-C., CHANG, Y.-H., AND KUO, T.-W. Garbage Collection and Wear Leveling for Flash Memory: Past and Future. In *SMARTCOMP* (2014).

[92] YOO, J., WON, Y., HWANG, J., KANG, S., CHOIL, J., YOON, S., AND CHA, J. VSSIM: Virtual Machine Based SSD Simulator. In *MSST* (2013).

A MQSim Validation

A.1 Evaluation Methodology

To validate the accuracy of MQSim, we compare its performance results to four state-of-the-art MQ-SSDs (SSD-A, SSD-B, SSD-C, and SSD-D) manufactured between 2016 and 2017. Table 4 lists key properties of the four MQ-SSDs. We precondition each device with full-load write traffic to write to 70% of the available logical space [71]. The device preconditioning process includes two 4-hour phases. In the first phase, we perform sequential writes, while in the second phase, we perform random writes. We perform real-system experiments on a server that contains an Intel Xeon E3-1240 v6 3.70GHz processor and 32 GB of DDR4 main memory. The system uses Ubuntu 16.04.2 with version 2.6.27 of the Linux kernel, and the OS is stored in a 500 GB Western Digital HDD. We run the `fio` benchmark tool for performance evaluations, and all storage devices are connected to the PCIe bus as add-in cards.

Table 4: Key characteristics of real MQ-SSDs.

Code	Production Year	Capacity	Flash Technology
SSD-A	2016	800 GB	MLC
SSD-B	2016	256 GB	MLC
SSD-C	2017	1 TB	TLC
SSD-D	2016	512 GB	TLC

We validate our simulator with four different configurations that correspond to our four real MQ-SSDs. To this end, we extract the main structural parameters of each real SSD using a microbenchmarking program. This program analyzes and estimates the SSD's internal configuration (e.g., NAND flash page size, NAND flash read/write latency, number of channels in the SSD, address mapping strategy, write cache size) based on the methods described in prior SSD modeling studies [14, 15, 36]. We have open-sourced our microbenchmark [1]. For garbage collection (GC) management, we enable all of the advanced GC mechanisms described in Section 4.2.3, except write suspension, in MQSim. According to the specifications of the flash chips used in two of the SSD devices, write suspension is not supported.

A.2 Performance Validation

We validate MQSim against real devices using both synthetic and real workloads. Our synthetic workloads issue random accesses, and consist of only read requests or only write requests, where we set the queue depth to 1 request.

Figure 10 compares the read and write request response time[4] measured on our four real MQ-SSDs with the latencies reported by MQSim for our synthetic workloads. The plots in Figure 10a and 10b show the read and the write latencies, respectively. The x-axes reflect different I/O request sizes, ranging from 4 kB to 1 MB. The blue curves show the error percentage of the simulation model. We observe that across all request sizes, the response times reported by MQSim match very closely

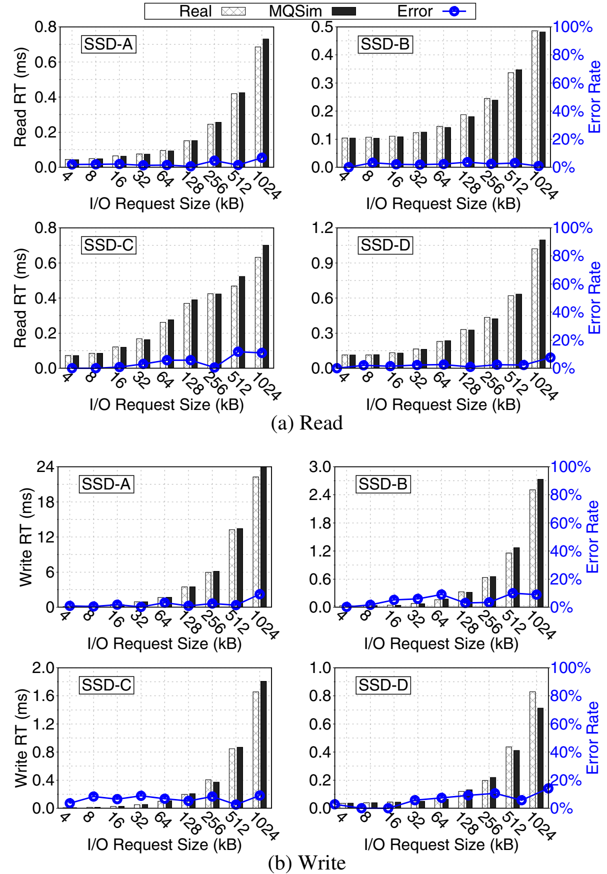


Figure 10: Average response time (RT) for read (a) and write (b) requests, reported by MQSim, compared to RT measured on four real MQ-SSD devices, for synthetic workloads. The blue curves show the error rates of MQSim's reported latency.

with the measured response times of the real devices, especially for SSD-B and SSD-D. Averaged across all four MQ-SSDs and all I/O request sizes, the error rates for read and write requests are 2.9% and 4.9%, respectively.

Figure 11 shows the accuracy of the request response time reported by MQSim as a cumulative distribution function (CDF), for three real workloads [53]: `tpcc`, `tpce`, and `exchange`. We observe that MQSim's reported response times are very accurate when compared to the response times measured on the real MQ-SSDs. The average error rates for SSD-A, SSD-B, SSD-C, and SSD-D are 8%, 6%, 18%, and 14%, respectively.

We conclude that MQSim accurately models the performance of real MQ-SSDs.

A.3 Multi-Queue Simulation

To validate the accuracy of the multi-queue I/O execution model in MQSim, we conduct a set of simulation experiments using two I/O flows, `Flow-1` and `Flow-2`, where each flow generates only sequential read requests. We maintain a constant request intensity for `Flow-1`, by setting its I/O queue depth to 8 requests. We vary the intensity of `Flow-2` across our experiments, by varying the I/O queue depth between 8 entries and 256 entries.

[4] Response time is defined as the time from when a host request is enqueued in the submission queue to when the SSD response is enqueued in the completion queue.

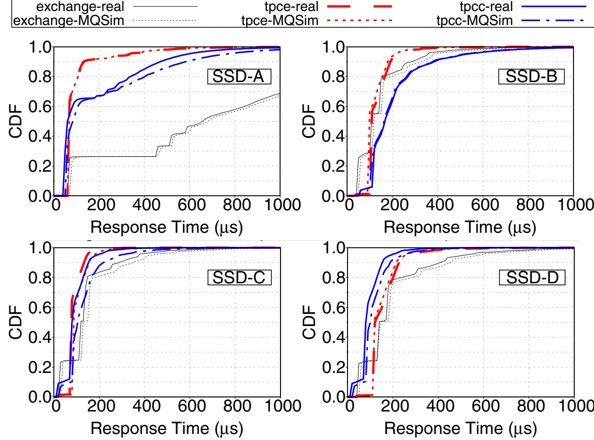
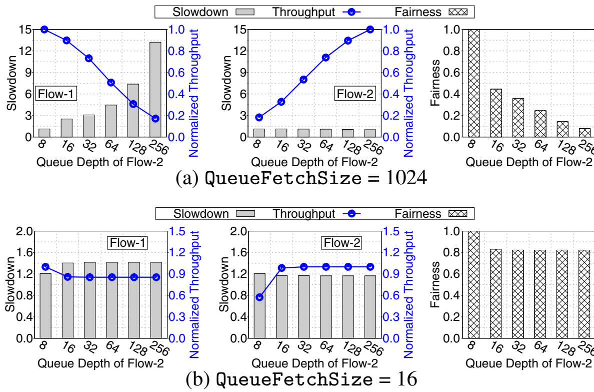


Figure 11: Comparison of response time CDF when running real workloads on MQSim and on real MQ-SSDs.

For each `Flow-2` I/O queue depth, we test two different values (16 and 1024) of `QueueFetchSize` (see Section 4.2.1).

Figure 12 shows the slowdown and normalized throughput (IOPS) of `Flow-1` (left) and `Flow-2` (center), and the fairness (see Section 3.1) of the system (right). We make two key observations from the figure.



(a) `QueueFetchSize` = 1024

(b) `QueueFetchSize` = 16

Figure 12: MQSim successfully models the multi-queue I/O processing model in (a) SSD-A, SSD-B, and SSD-C, and (b) SSD-D (compare with Figure 2).

First, we find that MQSim successfully models the behavior of real MQ-SSDs that are optimized for higher per-flow throughput (e.g., SSD-A, SSD-B, SSD-C) when the value of `QueueFetchSize` is equal to 1024. Figure 12a shows similar trends for slowdown, throughput, and fairness to the measurements we perform on real MQ-SSDs, which we show in Figure 2. When `QueueFetchSize` is set to 1024, a higher number of I/O requests from each flow are fetched into the device-level queue of the MQ-SSD. In both our MQSim results and the measured results on real MQ-SSDs, we observe that as the intensity of `Flow-2` increases, its throughput increases significantly with little slowdown, while the throughput of `Flow-1` decreases significantly, causing `Flow-1` to slow down greatly. This occurs because when `Flow-2` has a high intensity, it unfairly uses most of the back end resources in the MQ-SSD, causing requests from `Flow-1` to wait for longer latencies before they can be serviced.

Second, MQSim accurately models the behavior of real MQ-SSD products that implement mechanisms to control inter-flow interference, such as SSD-D, when `QueueFetchSize` is set to 16. We see that the trends in Figure 12b are similar to those observed in our measured results from SSD-D in Figure 2. When `QueueFetchSize` is set to 16, only a limited number of I/O requests for each concurrently-running flow are serviced by the back end, preventing any one flow from unfairly using most of the resources within the MQ-SSD. As a result, even when `Flow-2` has a high intensity, `Flow-1` does not experience significant slowdown.

We conclude that by adjusting `QueueFetchSize`, MQSim successfully models different multi-queue I/O processing mechanisms in modern MQ-SSD devices.

A.4 Steady-State Behavior Modeling

As we discuss in Section 4.4, MQSim pre-conditions the flash storage space and warms up the SSD data cache based on the characteristics of the co-running workloads. To validate the steady-state model in MQSim, we conduct a set of experiments using MQSim under high write intensity, and compare the results to those from real MQ-SSD devices. Figure 13 plots the read and write response times for (1) actual I/O execution on SSD-B (which is representative of the general behavior of the state-of-the-art SSDs we examine); (2) *MQSim-NoPrec*, where MQSim is run *without* pre-conditioning, and (3) *MQSim-Prec*, where MQSim is run *with* pre-conditioning.

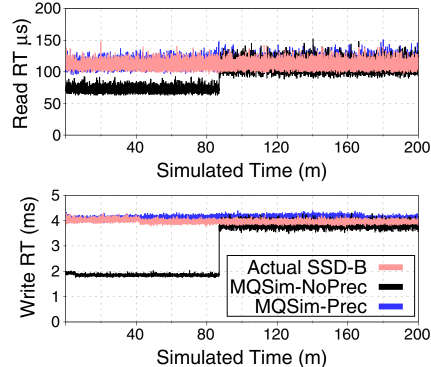


Figure 13: MQSim accurately models the steady-state read and write response times (RT) of SSD-B, using fast preconditioning.

We make two observations from the figure. First, MQSim with pre-conditioning successfully follows the response time results extracted from SSD-B. Second, MQSim without pre-conditioning reports lower response time results at the beginning of the experiment, since the simulated SSD is not yet in steady state. Once the whole storage space is written, the response time results become similar to the real device, as garbage collection and write cache evictions now take place in simulation at a rate similar to the rate measured on SSD-B. We conclude that MQSim's pre-conditioning quickly and accurately models the steady-state behavior of real MQ-SSDs.