FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives

Arash Tavakkol[†] Mohammad Sadrosadati[†] Saugata Ghose[‡] Jeremie S. Kim[‡†] Yixin Luo[‡]
Yaohua Wang[†§] Nika Mansouri Ghiasi[†] Lois Orosa[†*] Juan Gómez-Luna[†] Onur Mutlu[†‡]
[†]*ETH Zürich* [‡]*Carnegie Mellon University* [§]*NUDT* [*]*Unicamp*

Modern solid-state drives (SSDs) use new host–interface protocols, such as NVMe, to provide applications with fast access to storage. These new protocols make use of a concept known as the multi-queue SSD (MQ-SSD), where the SSD has direct access to the application-level I/O request queues. This removes most of the OS software stack that was used in older protocols to control how and when the I/O requests were dispatched to storage devices. Unfortunately, while the elimination of the OS software stack leads to a significant performance improvement, we show in this paper that it introduces a new problem: unfairness. This is because the elimination of the OS software stack eliminates the mechanisms that were used to provide fairness among applications in older SSDs.

To study application-level unfairness, we perform experiments using four real state-of-the-art MQ-SSDs. We demonstrate that the lack of fair scheduling mechanisms leads to high unfairness among concurrently-executing applications due to the interference among them. For instance, when one of these applications issues many more I/O requests than others, the other applications are slowed down significantly. We perform a comprehensive analysis of interference in real MQ-SSDs, and find four major interference sources: (1) the intensity of requests sent by each application, (2) differences in request access patterns, (3) the ratio of reads to writes, and (4) garbage collection.

To alleviate unfairness in MQ-SSDs, we propose the Flash-Level INterference-aware scheduler (FLIN). FLIN is a lightweight I/O request scheduling mechanism that provides fairness among requests from different applications. FLIN uses a three-stage scheduling algorithm that protects against all four major sources of interference, while respecting the application-level priorities assigned by the host. FLIN is implemented fully within the SSD controller firmware, requiring no new hardware, and has negligible (<0.06%) storage cost. Compared to a state-of-the-art I/O scheduler, FLIN improves the fairness and performance of a wide range of enterprise and datacenter storage workloads, with an average improvement of 70% and 47%, respectively.

1. Introduction

Solid-state drives (SSDs) are widely used as a storage medium today due to their high throughput, low response time, and low power consumption compared to conventional hard disk drives (HDDs). As more SSDs are deployed in datacenters and enterprise platforms, there has been a continued need to improve SSD performance. One area where SSD manufacturers have innovated on SSD performance is the *host–interface protocol*, which coordinates communication between applications and the SSD. SSDs initially adopted existing host–interface protocols (e.g., SATA [88]) that were originally designed for lower-performance HDDs. As the performance of the underlying storage technology (e.g., NAND flash memory) used by the SSD increased, these host–interface protocols became a significant performance bottleneck [114], mainly because these protocols *rely on the OS* to manage I/O requests and data transfers between the host system and the SSD.

To overcome this bottleneck, modern enterprise SSDs (e.g., [30–33, 65, 66, 87, 103, 104, 110, 111]) use new high-performance protocols, such as the Non-Volatile Memory Express (NVMe) protocol [21, 83]. These new protocols make use of the *multi-queue SSD* (MQ-SSD) [8, 48, 101] concept, where multiple host-side I/O request queues (in software) are *directly exposed* to the SSD controller. There are two benefits to directly exposing the request queues to the SSD controller: (1) there is no longer any need for the OS software stack to manage the I/O requests; and (2) the SSD can make more effective I/O request scheduling decisions than the OS, since the SSD knows exactly which of its internal resources are available or are busy serving other requests. Thus, the protocols eliminate the OS software stack, enabling MQ-SSDs to provide significantly higher performance than traditional SSDs [101, 114].

Unfortunately, eliminating the OS software stack also eliminates critical mechanisms that were previously implemented as part of the stack, such as *fairness control* [6, 8, 35, 55, 56, 84, 89, 106, 118]. Fairness control mechanisms work to equalize the effects of interference across applications when multiple applications concurrently access a shared device. Fairness is a critical requirement in multiprogrammed computers and multi-tenant cloud environments, where multiple *I/O flows* (i.e., series of I/O requests) from different applications concurrently access a single, shared SSD [36, 43, 84, 91, 94, 101].

For older host–interface protocols, the OS software stack provides fairness by limiting the number of requests that each application can dispatch from the host to the SSD. Since fairness was handled by the OS software stack, the vast majority of state-of-the-art SSD I/O request schedulers did *not* consider request fairness [20, 36, 37, 40, 77, 90, 112]. Surprisingly, even though new host–interface protocols, such as NVMe, do *not* use the OS software stack, modern MQ-SSDs still do *not* contain any fairness mechanisms. To demonstrate this, we perform experiments using four real state-of-the-art enterprise MQ-SSDs. We make two major findings. First, when two applications share the same SSD, one of the applications typically *slows down* (i.e., it takes more time to execute compared to if it were accessing the SSD alone) significantly. When we run a representative workload on our four SSDs, we observe that such slowdowns range from 2x to 106x. Second, with the removal of the fairness control mechanisms that were in the software stack, an application that makes a large number of I/O requests to an MQ-SSD can starve requests from other applications, which can hurt overall system performance and lead to denial-of-service issues [8, 84, 101]. Therefore, we conclude that there is a pressing need to introduce fairness control mechanisms within modern SSD I/O request schedulers.

In order to understand how to control fairness in a modern SSD, we experimentally analyze the sources of interference among I/O flows from different applications in an SSD using the detailed MQSim simulator [101]. Our experimental results enable us to identify four major types of interference:

1. *I/O Intensity*: The SSD controller breaks down each I/O request into multiple page-size (e.g., 4 kB) *transactions*. An

I/O flow with a greater flow *intensity* (i.e., the rate at which the flow generates transactions) causes a flow with a lower flow intensity to slow down, by as much as 49x in our experiments (Section 3.1).

2. *Access Pattern*: The SSD contains per-chip transaction queues that expose the *high internal parallelism* that is available. Some I/O flows exploit this parallelism by inserting transactions that can be performed concurrently into each per-chip queue. Such flows slow down by as much as 2.4x when run together with an I/O flow that does *not* exploit parallelism. This is because an I/O flow that does not exploit parallelism inserts transactions into only a small number of per-chip queues, which prevents the highly-parallel flow's transactions from completing at the same time (Section 3.2).
3. *Read/Write Ratio*: Because SSD writes are typically 10–40x slower than reads [62–64, 67, 92], a flow with a greater fraction of write requests unfairly slows down a concurrently-running flow with a smaller fraction of write requests, i.e., a flow that is more read-intensive (Section 3.3).
4. *Garbage Collection*: SSDs perform *garbage collection* (GC) [10, 13, 16, 23], where invalidated pages within the SSD are reclaimed (i.e., erased) some time after the invalidation. When an I/O flow requires frequent GC operations, these operations can block the I/O requests of a second flow that performs GC only infrequently, slowing down the second flow by as much as 3.5x in our experiments (Section 3.4).

To address the lack of OS-level fairness control mechanisms, prior work [36] implements a fair I/O request scheduling policy in the SSD controller that is similar to the software-based policies previously used in the OS [120]. Unfortunately, this approach has two shortcomings: (1) it *cannot* control two types of interference, access pattern interference or GC interference; and (2) it significantly decreases the responsiveness and overall throughput of an MQ-SSD [89]. **Our goal** in this work is to design an efficient I/O request scheduler for a modern SSD that (1) provides fairness among I/O flows by mitigating *all four types* of interference within an SSD, and (2) maximizes the performance and throughput of the SSD.

To this end, we propose the *Flash-Level INterference-aware scheduler* (FLIN) for modern SSDs. FLIN considers the priority class of each flow (which is determined by the OS), and carefully reorders transactions within the SSD controller to balance the slowdowns incurred by flows belonging to the same priority class. FLIN uses three stages to schedule requests. The first stage, *Intensity- and Parallelism-aware Queue Insert*, reorders transactions to mitigate the impact of different I/O intensities and different access patterns. The second stage, *Flow-level Priority Arbitration*, prioritizes transactions based on the priority class of their corresponding flow. The third stage, *Read/Write Wait-Balancing*, balances the overall stall time of read and write requests, and schedules GC activities to distribute the GC overhead proportionally across different flows. FLIN is fully implementable in the firmware of a modern SSD. As such, it requires no additional hardware, and in the worst case consumes less than 0.06% of the DRAM storage space that is already available within the SSD controller.

Our evaluations show that FLIN significantly improves fairness and performance across a wide variety of workloads and MQ-SSD configurations. Compared to a scheduler that uses a state-of-the-art scheduling policy [37] and fairness controls similar to those previously used in the OS [36], FLIN improves fairness by 70%, and performance by 47%, on average, across 40 MQ-SSD workloads that contain various combinations of concurrently-running I/O flows with different characteristics. We show that FLIN improves *both* fairness and performance across *all* of our tested MQ-SSD configurations.

In this paper, we make the following major contributions:

- We provide an in-depth experimental analysis of unfairness in state-of-the-art multi-queue SSDs. We identify four major sources of interference that contribute to unfairness.
- We propose FLIN, a new I/O request scheduler for modern SSDs that effectively mitigates interference among concurrently-running I/O flows to provide both high fairness and high performance.
- We comprehensively evaluate FLIN using a wide variety of storage workloads consisting of concurrently-running I/O flows, and demonstrate that FLIN significantly improves fairness and performance over a state-of-the-art I/O request scheduler across a variety of MQ-SSD configurations.

2. Background and Motivation

We first provide a brief background on multi-queue SSD (MQ-SSD) device organization and I/O request management (Section 2.1). We then motivate the need for a fair MQ-SSD I/O scheduler (Section 2.2).

2.1. SSD Organization

Figure 1 shows the internal organization of an MQ-SSD. The components within an SSD are split into two groups: (1) the *back end*, which consists of data storage units; and (2) the *front end*, which consists of control and communication units. The front end and back end work together to service the multiple page-size I/O transactions that constitute an I/O request.

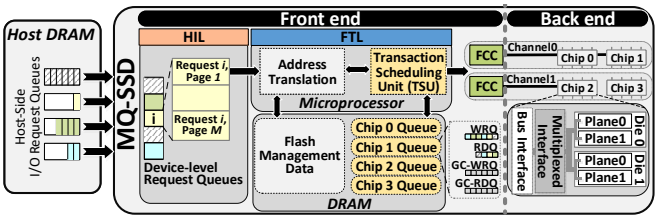


Figure 1: Internal organization of an MQ-SSD.

The back end contains multiple *channels* (i.e., request buses), where the channels can service I/O transactions in parallel. Each channel is connected to one or more *chips* of the underlying memory (e.g., NAND flash memory). Each chip consists of one or more *dies*, where each die contains one or more *planes*. Each plane can service an I/O transaction concurrently with the other planes. In all, this provides four levels of parallelism for servicing I/O transactions (channel, chip, die, and plane).

The front end manages the back end resources and issues I/O transactions to the back end channels. There are three major components within the front end [37, 61, 101]: (1) the *host–interface logic* (HIL), which implements the protocol used to communicate with the host and fetches I/O requests from the host-side request queues in a round-robin manner [83]; (2) the *flash translation layer* (FTL), which manages SSD resources and processes I/O requests [10, 13] using an embedded processor and DRAM; and (3) the *flash channel controllers* (FCCs), which are used to communicate between the other front end modules and the back end chips.

There are seven major steps in processing an I/O request. (1) The host generates the request and inserts the request into a dedicated in-DRAM I/O queue on the host side, where each

queue corresponds to a single I/O *flow*.[1] (2) The HIL in the MQ-SSD fetches the request from the host-side queue and inserts it into a device-level queue within the SSD. (3) The HIL parses the request and breaks it down into multiple transactions. (4) The *address translation module* in the FTL performs a logical-to-physical address translation for each transaction. (5) The *transaction scheduling unit* (TSU) [101, 102] in the FTL schedules each transaction for servicing [20, 102], and places the transaction into a chip-level software queue. (6) The TSU selects a transaction from the chip-level queue and dispatches it to the corresponding FCC. (7) The FCC issues a command to the target back end chip to execute the transaction.

In addition to processing I/O requests, the SSD controller must perform maintenance operations. One such operation is *garbage collection* (GC) [10, 13, 16, 23]. Due to circuit-level limitations, many of the underlying memories used in SSDs must perform writes to a logical page *out of place* (i.e., the data is written to a new physical page). For example, in NAND flash memory, before new data can be written to a physical page, the page must first be erased. The erase operation can only be performed at the granularity of a *flash block*, which contains *hundreds* of pages [10, 13]. Since many other pages in a block may still contain valid data at the time of the update, the block containing the page that is being updated is typically *not* erased at that time, and the page is simply marked as *invalid*. Periodically, the GC procedure (1) identifies candidate blocks that are ready to be erased (i.e., blocks where most of the pages are marked as invalid), (2) generates read/write transactions to move any remaining valid data in each candidate block to another location; and (3) generates an erase transaction for each block.

The TSU is responsible for scheduling both I/O transactions and GC transactions. For I/O transactions, it maintains a read queue (RDQ in Figure 1) and a write queue (WRQ) in DRAM for each back end chip. GC transactions are kept in a separate set of read (GC-RDQ) and write (GC-WRQ) queues in DRAM. The TSU typically employs a first-come first-serve transaction scheduling policy, and typically prioritizes I/O transactions over GC transactions [51, 117]. To better exploit the parallelism available in the back end, some state-of-the-art TSUs partially reorder the transactions within a queue [37, 40].

2.2. The Need for Fairness Control in an SSD

When multiple I/O flows are serviced concurrently, the behavior of the requests of one flow can negatively interfere with the requests of another flow. In order to understand the impact of such inter-flow interference, and how the scheduler affects this type of interference, we would like to quantify how *fairly* the requests of different flows are being treated. We say that an I/O scheduler is completely fair if concurrently-running flows with equal priority experience the same *slowdown* due to interference [19, 22, 70, 73, 84]. The slowdown S_i of flow i can be calculated as:

$$S_i = RT_i^{shared} / RT_i^{alone} \quad (1)$$

where RT_i^{shared} is the average response time of flow i when it runs concurrently with other flows, and RT_i^{alone} is the same flow's average response time when it runs alone. The *response time* of each request is measured from the time the request is generated on the host side to the time the host receives a response from the SSD. We define fairness (F) [22] as the ratio of the maximum and minimum slowdowns experienced by any flow in the system:

$$F = \underset{i}{MIN}\{S_i\} / \underset{i}{MAX}\{S_i\} \quad (2)$$

As defined, $0 < F \leq 1$. A lower F value indicates a greater difference between the flow that is slowed down the most and the flow that is slowed down the least, which is more *unfair* to the flow that is slowed down the most. In other words, higher values of F are desirable.

We conduct a large set of real system experiments to study the fairness of scheduling policies implemented in four real MQ-SSDs [31, 33, 65, 111] (which we call SSD-A through SSD-D), using workloads that capture a wide range of I/O flow behavior (see Section 6). Figure 2 shows the slowdowns and fairness for a representative workload where a moderate-intensity read-dominant flow (generated by the `tpce` application; see Table 2) runs concurrently with a high-intensity flow with an even mix of reads and writes (generated by the `tpcc` application; see Table 2), as measured over a 60 s period. We make two key observations from the figure. First, the amount of interference between the two flows fluctuates greatly over the period, which leads to high variability in the slowdown of each flow. Across the four SSDs, `tpce` experiences slowdowns as high as 2.8x, 69x, 661x, and 11x, respectively, while its *average slowdowns* are 2x, 4.4x, 106x, and 5.3x. Second, the SSDs that we study do *not* provide any fairness guarantees. We observe that `tpcc` does *not* experience high slowdowns throughout the entire period. We conclude that modern MQ-SSDs focus on providing high performance at the expense of large amounts of unfairness, because they lack mechanisms to mitigate interference among flows.

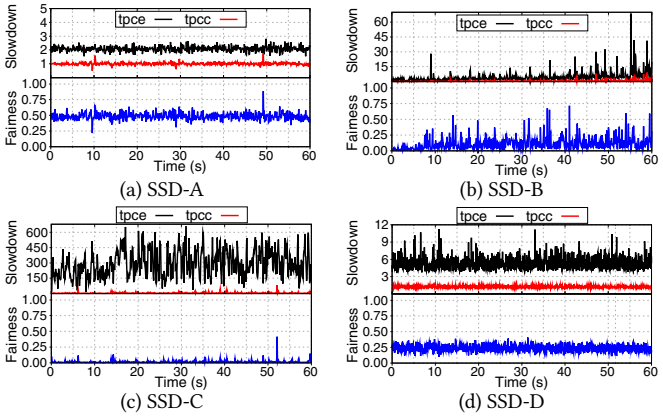


Figure 2: Slowdown and fairness of flows in a representative workload on four real MQ-SSDs.

3. An Analysis of Unfairness in MQ-SSDs

As Section 2.2 shows, real MQ-SSDs do *not* provide fairness across multiple queues. This is because state-of-the-art transaction scheduling units (TSUs) [20, 37, 40, 77, 90, 112] are designed to maximize the overall I/O request throughput of the entire SSD, and thus they do *not* take into account the throughput of *each* flow. Our goal in this work is to design a high-performance TSU for MQ-SSDs that delivers high SSD throughput *while providing fairness for each flow*.

In order to meet this goal, we must (1) identify the runtime characteristics of each I/O flow, and (2) understand how these

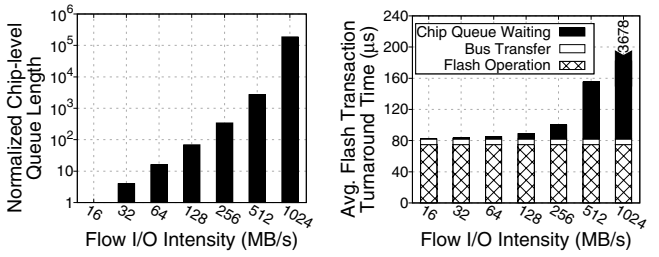
[1] In this work, we use the term *flow* to indicate a *series of I/O requests* that (1) are generated by a single user process; or (2) originate from the same host-side I/O queue (e.g., for systems running virtual machines, where fairness should be enforced at the VM level [36], and where each VM can be assigned to a dedicated I/O queue). To simplify explanations without loss of generality, we associate each flow with a dedicated host-side I/O queue in this work.

characteristics lead to unfairness. An I/O flow is made up of a stream of I/O requests, where each request is defined by its arrival time, starting address, size of the data that it requires, and access type (i.e., read or write) [42, 79, 119]. Based on the requests that make up a particular flow, we define four key characteristics of the flow: (1) I/O intensity, (2) access pattern, (3) read/write ratio of the requests in the flow, and (4) garbage collection (GC) demands of the flow. In this section, we study how differences in each of these characteristics can lead to unfairness, using the methodology described in Section 6.

3.1. Flows with Different I/O Intensities

The I/O intensity of a flow refers to the rate at which the flow generates transactions. This is dependent on the arrival time and the size of the I/O requests (as larger I/O requests generate a greater number of page-sized transactions). There are three factors that contribute to the amount of time required to service each transaction, which is known as the *transaction turnaround time*: (1) the wait time at the chip-level queue, (2) the transfer time between the front end and the back end, and (3) the time required to execute the transaction in the memory chip. As the intensity of a flow increases, the transfer time and execution time do not change, but the wait time at the chip-level queue grows.

Figure 3 shows how flow intensity affects chip-level queue length and average transaction turnaround time. We run a 100%-read synthetic flow where we vary the I/O intensity from 16 MB/s to 1024 MB/s. Figures 3a and 3b show, respectively, the average length of the chip-level queues and the breakdown of the transaction turnaround time with respect to the I/O intensity. We make three conclusions from the figures. First, the lengths of the chip-level queues increase proportionately with the I/O intensity of the running flow. Second, a longer queue leads to a longer wait time for transactions. Third, as the flow intensity increases, the contribution of the queue wait time to the transaction turnaround time increases, dominating the other two factors (shown in the figure).



(a) Chip-level queue length (b) Transaction turnaround time

Figure 3: Effect of flow I/O intensity.

When two flows with different I/O intensities execute concurrently, each flow experiences an increase in the average length of the chip-level queues, which can more adversely affect the flow with a low I/O intensity than the flow with a high I/O intensity. To characterize the effect, we execute a low-intensity 16 MB/s synthetic flow, referred to as the *base flow*, together with a synthetic flow of varying I/O intensities, referred to as the *interfering flow*. The I/O requests of the two flows are purely reads. Figure 4 depicts the slowdown and fairness results. The x-axis shows the intensity of the *interfering flow*. We observe that even when the intensity of the *interfering flow* exceeds 256 MB/s, the slowdown of the *interfering flow* is very small, while the *base flow* is slowed down by as much as 49x. The unfair slowdown of the *base flow* is due to a drastic increase in the average length of the chip-level queues compared to when the *base flow* runs alone. We conclude that *the average response time of a low-intensity flow substantially increases due to interference from a high-intensity flow.*

3.2. Flows with Different Access Patterns

The access pattern of a flow determines how its transactions are distributed across the chip-level queues. Some flows can take advantage of the parallelism available in the back end of the SSD, by distributing their transactions across all of the chip-level queues equally. Other flows do *not* benefit as much from the back-end parallelism, as their transactions are distributed unevenly across the chip-level queues. The access pattern depends on the starting addresses and sizes of each request in the flow.

To analyze the interference between concurrent flows with different access patterns, we run a *base flow* with a streaming access pattern [50, 70] together with an *interfering flow* whose access pattern changes periodically between streaming and random (i.e., we can control the fraction of time the flow performs random accesses). Figure 5 shows the slowdown of the two flows and the fairness of the SSD. The x-axis shows the randomness of the *interfering flow*. We observe that as the randomness of the *interfering flow* increases, the MQ-SSD becomes more unfair to the *base flow*. Both the probability of resource contention and the transaction turnaround time increase when the two flows are executed together. However, the *base flow*, which highly exploits and benefits from chip-level parallelism, is more susceptible to interference from the *interfering flow*. The uneven distribution of transactions from the *interfering flow* causes some, but not all, of the *base flow*'s transactions to stall. As a result, the *base flow's* parallelism gets destroyed, and the flow slows down until *its* last transaction is serviced. We conclude that *flows with parallelism-friendly access patterns are susceptible to interference from flows with access patterns that do not exploit parallelism.*

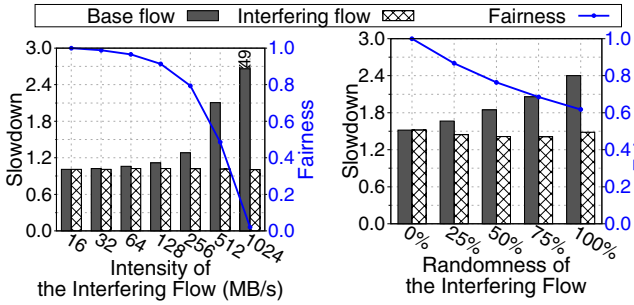


Figure 4: Effects of concurrently executing two flows with different I/O intensities on fairness.

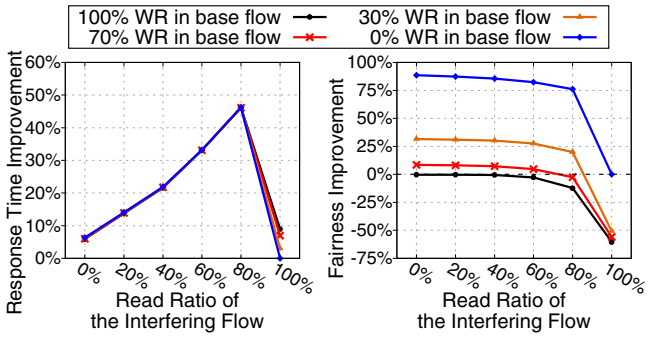
Figure 5: Effects of concurrently executing two flows with different access patterns on fairness.

3.3. Flows with Different Read/Write Ratios

As reads are 10-40x faster than writes, and are more likely to fall on the critical path of program execution, state-of-the-art MQ-SSD I/O schedulers tend to prioritize reads over writes to improve overall performance [20, 112]. To implement read prioritization, TSUs mainly adopt two approaches: (1) transactions in the read queue are always prioritized over those in the write queue, and (2) a read transaction can preempt an ongoing write transaction.

Unfortunately, read prioritization increases the wait time of write transactions, and can potentially slow down write-dominant flows in an MQ-SSD [84]. Figure 6 compares the response time and fairness of an SSD using a *read-prioritized*

scheduler (RP) with that of an SSD with a *first-come first-serve* scheduler (FCFS), which processes all transactions in the order they are received. In this experiment, we use two concurrently-running synthetic flows, where we control the rate at which the flows can issue new requests to the SSD by limiting the size of the host-side queue. We sweep the write rate of the *base flow* (0%, 30%, 70%, and 100%), shown as different curves, and set the host-side queue length to one request. We set the host-side queue length for the *interfering flow* to six requests (i.e., it has 6x the request rate of the *base flow*) to increase the probability of interference, and we sweep the *interfering flow*'s read ratio from 0% to 100%. In Figures 6a and 6b, the x-axis marks the read ratio of the *interfering flow*. We make three observations from the figures. First, RP maintains or improves the average response time for all flow configurations over FCFS. Second, RP leads to lower fairness than FCFS when the *interfering flow* is heavily read-dominant (i.e., its read ratio $> 80\%$). Third, fairness in an SSD with RP increases when the read ratio of the *interfering flow decreases*. We conclude that *when concurrently-running flows have different read/write ratios, existing scheduling policies are not effective at providing fairness.*



(a) Response time *improvement* of RP compared to FCFS

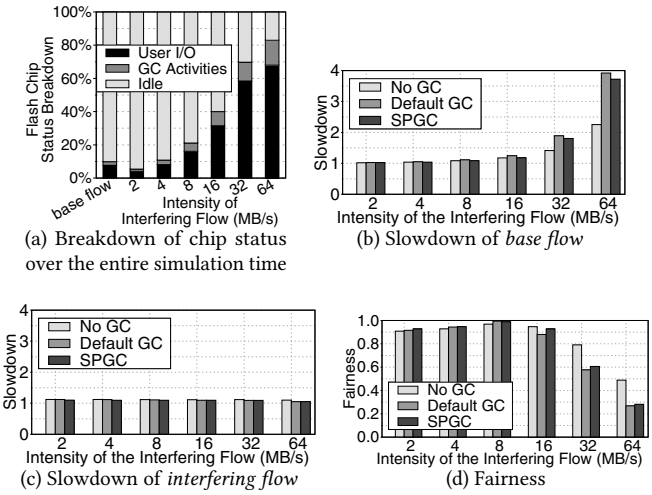
(b) Fairness *improvement* of RP compared to FCFS

Figure 6: Effects of read prioritization.

3.4. Flows with Different GC Demands

GC is one of the major sources of performance fluctuation in SSDs [26,27,38,41,51,54,108]. The rate at which GC is invoked depends primarily on the write intensity and access pattern of the flow [107]. A flow that causes GC to be invoked frequently (*high-GC*) can disproportionately slow down a concurrently-running flow that does not invoke GC often (*low-GC*). Prior works that optimize GC performance [26,27,38,41,51,54,108] can reduce interference between GC operations and transactions from I/O requests, and thus can improve fairness. However, these works *cannot* completely eliminate interference between a high-GC flow and a low-GC flow that occurs when a high-GC flow results in a *write cliff* (i.e., when the high intensity of the write requests causes GC to be invoked frequently, resulting in a large drop in performance [39]).

Figure 7 shows the effect of GC execution on MQ-SSD fairness. We run a *base flow* with moderate GC demands (i.e., 8 MB/s write rate) and an *interfering flow* where we change the GC demands by varying the write rate (2 MB/s to 64 MB/s). We evaluate three different scenarios for GC: (1) no GC execution, assuming the MQ-SSD capacity is infinite; (2) default (i.e., non-preemptive) GC execution; and (3) *semi-preemptive GC* (SPGC) [51,52] execution. SPGC tries to mitigate the negative effect of GC on user I/O requests by allowing user transactions to preempt GC activities when the number of free pages in the SSD is above a specific threshold (GC_{hard}). Figure 7a shows the fraction of time spent on I/O requests and on GC when the *base flow* (the leftmost column) and the *interfering flow* (all other columns) are executed alone. Figures 7b, 7c, and 7d show the slowdown of the *base flow*, the slowdown of the *interfering flow*, and the fairness of the MQ-SSD, respectively. The x-axis in these figures represents the write intensity of the *interfering flow*.



(a) Breakdown of chip status over the entire simulation time

(b) Slowdown of *base flow*

(c) Slowdown of *interfering flow*

(d) Fairness

Figure 7: Effects of GC demands on fairness.

We make four key observations. First, the GC activities increase proportionately with write intensity. Second, as the write intensity of the *interfering flow* increases, which triggers more GC operations, fairness decreases. Third, when the difference between the GC demands of the *base flow* and the *interfering flow* increases, the default GC mechanism leads to a higher slowdown of the *base flow*, degrading fairness. Fourth, SPGC improves fairness over the default GC mechanism, but only by a small amount, as SPGC disables preemption when there are a large number of pending writes. We conclude that *the GC activities of a high-GC flow can unfairly block flash transactions of a low-GC flow.*

4. FLIN: Fair, High-Performance Scheduling

As Section 3 shows, the four sources of interference that slow down flows all result from interference among the transactions that make up each flow. Based on this insight, we design a new MQ-SSD transaction scheduling unit (TSU) called the *Flash-Level INterference-aware scheduler* (FLIN). FLIN schedules and reorders transactions in a way that significantly reduces each of the four sources of interference, in order to provide fairness across concurrently-running flows.

FLIN consists of three stages, as shown in Figure 8. The first stage, *fairness-aware queue insertion*, inserts each new transaction from an I/O request into the appropriate read/write queue (Section 4.1). The position at which the transaction is inserted into the queue depends on the current intensity and access pattern of the flow that corresponds to the transaction, such that (1) a low-priority flow is not unfairly slowed down, and (2) a flow that can take advantage of parallelism in the back end does not experience interference that undermines this parallelism. The second stage, *priority-aware queue arbitration*, uses priority levels that are assigned to each flow by the host to determine the next read request and the next write request for each back end memory chip (Section 4.2). The arbiter aims to ensure that all flows assigned to the same priority level are slowed down equally. The third stage, *wait-balancing transaction selection*, selects the transaction to dispatch to the

flash chip controllers (FCCs, which issue commands to the memory chips in the back end; see Section 2.1) in a way that alleviates the interference introduced by read/write ratio disparities among flows and by garbage collection (Section 4.3). To achieve this, the scheduler chooses to dispatch a read transaction, a write transaction, or a garbage collection transaction, in a way that (1) balances the slowdown for read transactions with the slowdown of write transactions, and (2) distributes the overhead of garbage collection transactions fairly across all flows. FLIN is designed to be easy to implement, with low overhead, as we describe in Section 5.

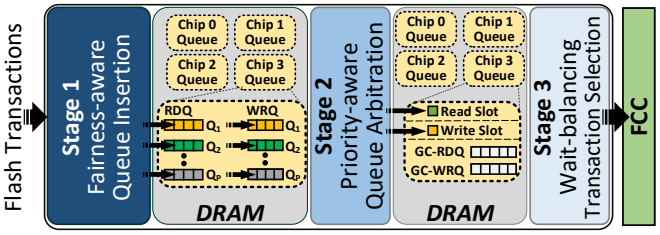


Figure 8: High-level overview of FLIN.

4.1. Fairness-Aware Queue Insertion

We design fairness-aware queue insertion, the first stage of FLIN, to relieve interference that occurs due to the intensity and access pattern of concurrently-running flows. Within the first stage, FLIN maintains separate per-chip read and write queues (RDQ and WRQ in Figure 8, respectively) for each flow priority level, since priorities are not processed until the second stage of FLIN. Fairness-aware queue insertion uses a heuristic $O(t)$ algorithm to determine where each transaction is inserted within the queue that corresponds to the access type (i.e., read or write) of the transaction and the priority level of the flow that generated the transaction. Finding the optimal ordering of transactions in each queue is a variant of the *multidimensional assignment problem* [85], which is NP-hard [82]. FLIN uses a heuristic that approximates the optimal ordering, in order to bound the time required for the first stage and to minimize the fraction of the transaction turnaround time that is spent on scheduling.

Algorithm 1 shows the heuristic used to insert a new transaction (TR_{new}) into its corresponding chip-level queue (Q). The heuristic is based on three key insights: (1) the performance of a low-intensity flow is very sensitive to the increased queue length when a high-intensity flow inserts many transactions (Section 3.1); (2) some flows can change between being high-intensity and low-intensity, based on the current phase of the application [80]; and (3) some, but not all, of the transactions from a flow that take advantage of back-end parallelism experience high queue wait times when a flow with poor parallelism inserts requests into some, but not all, of the chip-level queues (Section 3.2). The algorithm consists of two steps, which we describe in detail.

Step 1: Prioritize Low-Intensity Flows. Transactions from a low-intensity flow are *always* prioritized over transactions from a high-intensity flow. Figure 9a shows an example snapshot of a chip-level queue when FLIN inserts a newly-arrived transaction (TR_{new}) into the queue. The heuristic checks to see whether TR_{new} belongs to a low-intensity flow (line 9 of Algorithm 1; see Section 5.1 for details). If TR_{new} is from a low-intensity flow, the heuristic inserts TR_{new} into the queue immediately after $Last_{low}$, the location of the farthest transaction from the head of the queue that belongs to a low-intensity flow (line 10), and advances to Step 2a (line 12). If

Algorithm 1 Fairness-aware queue insertion in FLIN.

```
1:  Inputs:
2:      Q: the chip-level queue to insert the new transaction
3:      TR_new: the newly arriving flash transaction
4:      Source: source flow of the new transaction
5:      F_thr: fairness threshold for high-intensity flows
6:
7:  Last_low ← farthest transaction from head of Q belonging to a low-intensity flow
8:  // Step 1: Prioritize Low-Intensity Flows
9:  if Source is a low-intensity flow then                    // more detail in Section 5.1
10:     Insert(TR_new after Last_low)
11:     // Step 2a: Maximize Fairness Among Low-Intensity Flows.
12:     Estimate waiting time of TR_new as if Source were running alone // Section 5.2
13:     Reorder transactions of low-intensity flows in Q for fairness   // Section 5.3
14: else
15:     Insert(TR_new at the tail of Q)
16:     // Step 2b: Maximize Fairness Among High-Intensity Flows.
17:     Estimate waiting time of TR_new as if Source were running alone
18:     F ← Estimate fairness from average slowdown of departed transactions (Q)
19:     if F < F_thr and Source has experienced the maximum slowdown then
20:         Move TR_new to Last_low + 1
21:     else Reorder transactions from the tail of Q to Last_low + 1 for fairness
```

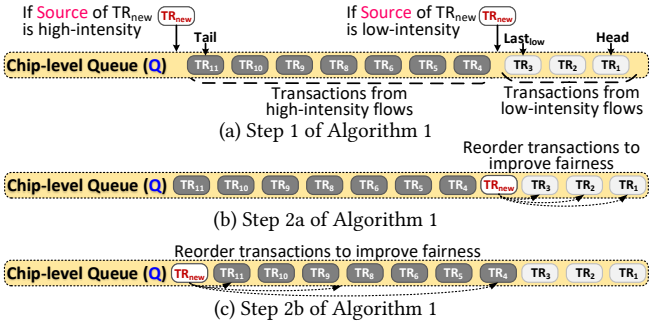


Figure 9: Fairness-aware queue insertion in FLIN.

TR_{new} belongs to a high-intensity flow, the heuristic inserts the transaction at the tail of the queue (line 15), and advances to Step 2b (line 17).

Step 2a: Maximize Fairness Among Low-Intensity Flows. The heuristic begins this step by estimating how long TR_{new} would have to wait to be serviced if its source flow were running alone (line 12). This estimate is also used to determine a transaction's slowdown and is kept alongside the transaction in the queue. The heuristic then reorders transactions of low-intensity flows in the queue for fairness (line 13). As Figure 9b shows, this involves moving TR_{new} from the tail of the low-intensity-flow transactions closer to the head, in order to maximize the ratio of the minimum slowdown to maximum slowdown for all pending transactions from low-priority flows. This reordering step prioritizes a transaction (i.e., moves it closer to the queue head) that has a greater slowdown, which indicates that its source flow either (1) has a lower intensity, or (2) better utilizes the parallelism available in the MQ-SSD back end. We describe the details of the alone wait time estimation (line 12) and the reordering step (line 13) in Sections 5.2 and 5.3, respectively.

Step 2b: Maximize Fairness Among High-Intensity Flows. Figure 9c shows a general overview of Step 2b. The heuristic begins this step by estimating the alone wait time of TR_{new} (line 17), using the same procedure in Step 2a. However, unlike the low-intensity-flow transactions, this step performs slowdown-aware queue shuffling in order to prevent any one high-intensity flow from being treated much more unfairly than the other high-intensity flows. After a new high-intensity transaction is inserted into the queue, the heuristic checks the current fairness (line 18). If the fairness is lower than a prede-

termined threshold (F_{thr}), and the new transaction belongs to the flow with the maximum slowdown (line 19), the heuristic moves the transaction ahead of all other high-intensity-flow transactions (line 20). This approach can often help flows that are transitioning from low-intensity to high-intensity, as such flows are particularly susceptible to interference from other high-intensity flows. If fairness is above the threshold, which means that all high-intensity flows are being treated somewhat fairly, the heuristic only reorders the high-intensity-flow transactions for fairness (line 21).

4.2. Priority-Aware Queue Arbitration

Many host–interface protocols, such as NVMe [83], allow the host to assign different priority levels to each flow. The second stage of FLIN enforces these priorities, and ensures that flows at the same priority level are slowed down by an equal amount. As we discuss in Section 4.1, FLIN maintains a read and write queue for each priority level (i.e., if there are P priority levels as defined by the protocol, FLIN includes P read queues and P write queues in the DRAM of the SSD, for each flash chip). Priority-aware queue arbitration selects one ready read transaction from the transactions at the head of the P read queues, and one ready write transaction from the transactions at the head of the P write queues. The two selected transactions then move to the last stage of the scheduler.

Whenever more than one queue has a transaction at the queue head that is ready to execute (i.e., the back end resources it requires are not being used by another transaction), the queue arbiter uses a *weighted round-robin* policy [76] to select the read and write transactions that move to the next scheduling stage. If the protocol defines P priority levels, where level 0 is the lowest priority and level $P-1$ is the highest priority, FLIN assigns the weight 2^i to priority level i. Under weighted round-robin, this means that out of every $\sum_i 2^i$ scheduling decisions in the second stage, transactions in the priority level i queue receive 2^i of these slots.

4.3. Wait-Balancing Transaction Selection

We design wait-balancing transaction selection, the last stage of FLIN, to minimize interference that occurs due to the read/write ratios (Section 3.3) and garbage collection demands of concurrently-running flows (Section 3.4). A transaction stalls if the back-end resources that it needs are being used by another transaction (which can be a transaction from another flow, or a garbage collection transaction). Wait-balancing transaction selection attempts to distribute this stall time evenly across all read and write transactions.

Wait-balancing transaction selection chooses one of the following transactions to dispatch to the FCC: (1) the ready read transaction determined by priority-aware queue arbitration (Section 4.2), which is stored in a *read slot* in DRAM; (2) the ready write transaction determined by priority-aware queue arbitration, which is stored in a *write slot* in DRAM; (3) the transaction at the head of the garbage collection read queue (GC-RDQ); and (4) the transaction at the head of the garbage collection write queue (GC-WRQ). Algorithm 2 depicts the selection heuristic, which consists of two steps.

Step 1: Estimate the Proportional Wait. FLIN uses a novel approach to determine *when* to prioritize reads over writes, which we call *proportional waiting*. Previous scheduling techniques [20, 112] always prioritize reads over writes, which as we show in Section 3.3 leads to unfairness. Proportional waiting avoids this unfairness.

Algorithm 2 Wait-balancing transaction selection in FLIN.

```
1:  Inputs:
2:    ReadSlot: the read transaction waiting to be issued to the SSD back end
3:    WriteSlot: the write transaction waiting to be issued to the SSD back end
4:    GC-RDQ, GC-WRQ: the garbage collection read and write queues
5:
6:  PW^read ← EstimateProportionalWait (ReadSlot)          // Section 5.4
7:  PW^write ← EstimateProportionalWait (WriteSlot)        // Section 5.4
8:  if PW^read > PW^write then
9:    Dispatch ReadSlot to FCC (flash chip controller)
10: else
11:   if number of free pages < GC_FLIN then
12:     GCMigrationCount ← AssignGCMigrations(WriteSlot) // Section 5.4
13:   while GCMigrationCount > 0 do
14:     Dispatch the transaction at the head of GC-RDQ to FCC
15:     Dispatch the transaction at the head of GC-WRQ to FCC
16:     GCMigrationCount ← GCMigrationCount - 1
17:   Dispatch WriteSlot to FCC
```

We define the *proportional wait time* (PW) of a transaction as the ratio of its wait time in the scheduler (T_{wait}), from the time that the transaction is received by the scheduler until the time the transaction is dispatched to the FCC, over the sum of the time required to perform the operation in the memory (T_{memory}) and transfer data back to the front end ($T_{transfer}$). The transaction in the read slot is prioritized over the transaction in the write slot when the read transaction's proportional wait time is greater than the write transaction's proportional wait time (lines 6–8 in Algorithm 2).

Step 2: Dispatch Transactions. If the read transaction is prioritized, it is dispatched to the flash channel controller (see Section 2.1) right away (line 9). If the write transaction is prioritized, the scheduler then considers *also dispatching* garbage collection operations, since a write transaction takes significantly longer to complete than a read transaction and the cost of garbage collection operations can be amortized by serving them together with the selected write transaction. If the number of free pages available in the memory is lower than a pre-determined threshold (GC_{FLIN}), and there are garbage collection transactions waiting in the GC-RDQ or GC-WRQ, the scheduler (1) determines how many of these transactions to perform, which is represented by *GCMigrationCount* (line 12), and (2) issues *GCMigrationCount* transactions (lines 13–16). Since garbage collection transactions read a valid page and write the page somewhere else in the memory, FLIN always executes a pair of read and write transactions. Once the garbage collection transactions are done, the scheduler dispatches the write transaction (line 17).

Optimizations. FLIN employs two optimizations beyond the basic read-write wait-balancing algorithm. First, FLIN supports *write transaction suspension* [112] whenever the proportional wait time (PW) of the transaction in the read slot is larger than the PW of the *currently-executing* write operation. Second, when the read and write slots are empty, FLIN dispatches a pair of garbage collection read/write transactions. If a transaction arrives at the read slot when the garbage collection transactions are being executed, FLIN preempts the garbage collection write and executes the transaction at the read slot, to avoid unnecessarily stalling the incoming read.

5. Implementation of FLIN

To realize the high-level behavior of FLIN (Section 4), we need to implement a number of functions, particularly for Stages 1 and 3 of the scheduler. We discuss each function in detail in this section, and then discuss the overhead required to implement FLIN in an MQ-SSD.

5.1. Classifying Flow I/O Intensity

The first stage of FLIN dynamically categorizes flows as either *low-intensity* or *high-intensity*, with separate classifications for the flow's read behavior and its write behavior. FLIN performs this classification over fixed-length intervals. During each interval, FLIN records the number of read and write transactions enqueued by each flow. At the end of the interval, FLIN uses the transaction counts to calculate the read and write arrival rates of each flow. If the read arrival rate of a flow is lower than a predetermined threshold α_C^{read}, FLIN classifies the flow as low-intensity for reads; otherwise, FLIN classifies the flow as high-intensity for reads. FLIN performs the same classification for write intensity using the write arrival rate of the flow, and a predetermined threshold α_C^{write}. We empirically set the epoch length to 10 ms, which we find is short enough to detect changes in flow intensity, but long enough to capture differences in intensity among the flows.

5.2. Estimating Alone Wait Time and Slowdown

The first stage of FLIN estimates the alone wait time of each transaction when the transaction is enqueued. Afterwards, whenever the transactions are reordered for fairness in the first stage (see Section 4.1), FLIN uses the alone wait time estimate to predict how much each individual transaction will be slowed down due to flow-level interference. The slowdown of an individual transaction, S^{TR}, is calculated as $S^{TR} = T^{TR}_{shared}/T^{TR}_{alone}$, where T^{TR}_{shared} is the transaction turnaround time when the flow corresponding to the transaction runs concurrently with other flows, and T^{TR}_{alone} is the transaction turnaround time when the flow runs alone. For both the shared and alone turnaround times, FLIN estimates T^{TR} using the following equation:

$$T^{TR} = T^{TR}_{memory} + T^{TR}_{transfer} + T^{TR}_{wait} \tag{3}$$

T^{TR}_{memory} and $T^{TR}_{transfer}$ are the times required to execute a command in the memory chip and transfer the data between the back end and the front end, respectively. Both values are constants that are determined by the speed of the components within the MQ-SSD. T^{TR}_{wait} is the amount of time that the transaction must wait in the read or write queue before it is issued to the back end.

To estimate T^{TR}_{alone}, FLIN must estimate $T^{TR}_{wait\_alone}$, which is the alone wait time. If the flow were running alone, the only other requests that the transaction would wait on are other requests in the same queue from the same flow. As a result, FLIN uses the last request inserted into the same queue *by the same flow* to determine how long the current transaction would have to wait if the other flows were not running. This can be determined by calculating how much longer the last request needs to complete:

$$T^{TR}_{wait\_alone} = T^{last}_{enqueue} + T^{last}_{alone} - T_{now} \tag{4}$$

Essentially, we take the timestamp at which the last request was enqueued ($T^{last}_{enqueue}$), add its estimated transaction turnaround time (T^{last}_{alone}) to determine the timestamp at which the request is expected to finish, and subtract the current timestamp (T_{now}) to determine the remaining time needed to complete the request. If the queue contains no other transactions from the same flow when the transaction arrives, then the transaction would not have to wait if the flow was run alone, so FLIN sets $T^{TR}_{wait\_alone}$ to 0.

To estimate T^{TR}_{shared}, FLIN estimates $T^{TR}_{wait\_shared}$ based on the current position of the transaction in the queue. For a transaction sitting at the p-th position in the queue (where the request at the head of the queue is at position 1), FLIN uses the following equation:

$$T^{TR}_{wait\_shared} = T_{now} - T_{enqueued} + T_{chip\_busy} + \sum_{i=1}^{p-1} \left(T^{i}_{memory} + T^{i}_{transfer} \right) \tag{5}$$

$T_{now} - T_{enqueued}$ gives the time that has elapsed since the transaction was enqueued (i.e., how long the transaction has already waited for). To compute $T^{TR}_{wait\_shared}$, FLIN adds the time elapsed so far to the amount of time left to service the currently-executing transaction (T_{chip_busy}), and the time required to execute and transfer data for all transactions ahead in the queue that have not yet been issued to the back end.

While FLIN calculates T^{last}_{alone} only once (when a transaction is first enqueued), it calculates T^{TR}_{alone} and T^{TR}_{shared} every time the slowdown of an individual transaction needs to be used during the first stage.

5.3. Reordering Transactions for Fairness

When the first stage of FLIN reorders transactions within a region of a transaction queue for fairness (in Steps 2a and 2b in Section 4.1), it scans through all transactions in the region to find a position for the newly-arrived transaction (TR_{new}) that maximizes the ratio of minimum to maximum slowdown of the queued transactions. This requires FLIN to perform two passes over the transactions within the queue region.

During the first pass, FLIN estimates the slowdown of each transaction in the region (see Section 5.2). Then, it calculates fairness across all enqueued requests by using Equation 2 over the per-transaction slowdowns.

During the second pass, FLIN starts with the last transaction in the queue region (i.e., the one closest to the queue tail), and uses the slowdown to determine the insertion position for the newly-arrived transaction that is expected to increase fairness by the largest amount. To do this, at every position p, FLIN estimates the change in fairness by recalculating the slowdown only for (1) the newly-arrived transaction, if it were to be inserted into position p; and (2) the transaction currently at position p, if it were moved to position $p-1$. FLIN keeps track of the position that leads to the highest fairness as it checks all of the positions in the queue region. After the second pass is complete, FLIN moves TR_{new} to the position that would provide the highest fairness.

With this two-pass approach, if TR_{new} is from a flow that has only a few transactions in the queue (e.g., the flow is low-intensity), the reordering moves the transaction closer to the head of the queue. Hence, FLIN significantly reduces the slowdown of the transaction, making its slowdown value more balanced with respect to transactions from other flows.

5.4. Estimating Proportional Wait Time

The third stage of FLIN chooses whether to dispatch the transaction in the read slot or the transaction in the write slot to the flash chip controller (FCC), by selecting the transaction that has the higher proportional wait time. In Section 4.3, we define the proportional wait time (PW) as:

$$PW = T_{wait} \div (T_{memory} + T_{transfer}) \tag{6}$$

T_{memory} and $T_{transfer}$ are the execution and data transfer time, respectively, of a read or write transaction that is issued to the back end of the SSD. T_{wait} represents how long the total wait time of the transaction would be, starting from the time the host made the I/O request that generated the transaction, if FLIN chooses to issue the transaction in the *other* slot first. In other words, if FLIN is evaluating *PW* for the transaction in the read slot, it calculates how long that transaction would have to wait if the transaction in the write slot were performed first. We can calculate T_{wait} as:

$$T_{wait} = T_{now} - T_{enqueued} + T_{other\_slot} + T_{GC} \quad (7)$$

$T_{now} - T_{enqueued}$ gives the amount of time that the transaction has already waited for, T_{other_slot} is the time required to perform the operation waiting in the other slot first, and T_{GC} is the amount of time required to perform pending garbage collection transactions. Recall from Algorithm 2 (Section 4.3) that if the transaction in the read slot is selected, no garbage collection transactions are performed, so $T_{GC} = 0$. T_{GC} can be non-zero if the transaction in the write slot is selected.

To determine T_{other_slot}, we add the time needed to execute the transaction and transfer data together with the time needed for any garbage collection transactions that would be performed along with the transaction based on Algorithm 2:

$$T_{other\_slot} = T_{memory} + T_{transfer} + T_{GC} \quad (8)$$

In Equations 7 and 8, $T_{GC} = 0$ if either (1) the transaction being considered is a read, or (2) there are no pending garbage collection operations. Otherwise, T_{GC} is estimated as:

$$T_{GC} = \sum_{i=1}^{GCM} (2T_{transfer} + T^{read}_{memory} + T^{write}_{memory}) + T^{erase}_{memory} \quad (9)$$

In this equation, *GCM* stands for *GCMigrationCount*, which is the number of page migrations that should be performed along with the transaction in the write slot to fairly distribute the interference due to garbage collection (Section 3.4). Recall from Section 2.1 that for many SSDs, such as those that use NAND flash memory, garbage collection occurs at the granularity of a block. A block that is selected for garbage collection may still contain a small number of valid pages, which must be moved to a new block before the selected block can be erased. For each valid page, this requires the SSD to (1) execute a read command on the memory chip containing the old physical page (which takes T^{read}_{memory} time), (2) two data transfers between the back end and the front end for GC read and write operations ($2T_{transfer}$), and (3) execute a write command on the memory chip containing the new physical page (T^{write}_{memory}). Hence, for each page migration, we add the time required to perform these four steps. If all GC page migrations from a block are finished, FLIN adds in the time required to perform the erase operation on the block (T^{erase}_{memory}).

FLIN determines the number of GC page movements (*GCM*) that should be executed ahead of a write transaction generated by flow f, based on the fraction of all garbage collection operations that were were previously caused by the flow. The key insight is to choose a *GCM* value that distributes a larger number of GC operations to a flow that is responsible for causing more GC operations. FLIN estimates *GCM* by determining (1) the fraction of all writes that were generated by flow f, since write operations lead to garbage collection (see Section 2.1); and (2) the fraction of valid pages that belong to flow f in a block that is selected for garbage collection (which FLIN approximates by using the fraction of *all* currently-valid pages in the SSD that were written by flow f), since each valid page in the selected block requires a GC operation:

$$GCM = \frac{NumWrites_f}{\sum_i NumWrites_i} \times \frac{Valid_f}{\sum_i Valid_i} \times length_{GC-RDQ} \quad (10)$$

where $NumWrites_f$ is the number of write operations performed by flow f since the last garbage collection operation, $Valid_f$ is the number of valid pages in the SSD for flow f (which is determined using the history of valid pages that is maintained by the FTL page management mechanism [10,13]), and $length_{GC-RDQ}$ is the number of queued garbage collection read transactions (which represents the total number of pending garbage collection migrations).

While FLIN can use fine-grained data separation techniques proposed for multi-stream SSDs [41] for more accurate GC cost estimation, we find that our estimates from Equation 10 are accurate enough for our purposes.

5.5. Implementation Overhead and Costs

FLIN can be implemented in the firmware of a modern SSD, and does not require specialized hardware. The requirements of FLIN are very modest compared to the processing power and DRAM capacity of a modern SSD controller [10–12, 14, 33, 37, 57–61, 104]. FLIN does *not* interfere with any SSD management tasks [10], such as the GC candidate block selection policy [23, 53, 107], the address translation scheme, or other FTL maintenance and reliability enhancement tasks, and hence it can be adopted independently of SSD management tasks.

DRAM Overhead. FLIN requires only a small amount of space within the in-SSD DRAM to store four pieces of information that are not kept by existing MQ-SSD schedulers: (1) the read and write arrival rates of each flow in the last interval (Section 5.1), which requires a total of $2 \times \#flows \times \#chips$ 32-bit words; (2) the alone wait time of each transaction waiting in the chip-level queues (Section 5.2), which requires a total of *max_transactions* 32-bit words; (3) the average slowdown of all high-intensity flows in each queue (Section 4.1, Step 3), which requires $2 \times \#flows \times \#chips$ 32-bit words; (4) the GC cost estimation data (i.e., $Count^i_{write}$ and $Valid_i$; Section 5.4), which requires $2 \times \#flows$ 32 bit variables. For an SSD that has 64 flash chips and 2 GB of DRAM [61] (resulting in a very conservative maximum of 262,144 8 kB transactions that the DRAM can hold), and that supports a maximum of 128 concurrent flows [30, 32, 33, 65, 66, 87, 103, 104, 110, 111], the DRAM overhead of FLIN is 1.13 MB, which is less than 0.06% of the total DRAM capacity.

Latency Impact. `ReorderForFairness` is the most time-consuming function of FLIN, as it performs two passes over the queue each time. Using our evaluation platform (see Section 6) to model a modern SSD controller with a 500 MHz multicore processor [61], we find that `ReorderForFairness` incurs a worst-case and average latency overhead of 0.5% and 0.0007%, respectively, over a state-of-the-art out-of-order transaction scheduling unit [37]. None of the FLIN functions, including `ReorderForFairness`, execute on the critical path of transaction processing. Hence, the maximum data throughput of FLIN is identical to the baseline.

6. Methodology

Simulation Setup. For our model-based evaluations, we use MQSim [101], an open-source MQ-SSD simulator that

accurately models all of the major front-end and back-end components in a modern MQ-SSD. We have open-sourced our implementation of FLIN [1] to foster research into fairness in modern SSDs. Table 1 shows the configuration that we use to model a representative MQ-SSD system in our evaluations. In our evaluation, FLIN uses α_c^{read} =32 MB/s, α_c^{write} =256 kB/s, F_{thr} = 0.6, and GC_{FLIN} = 1 K.

Table 1: Configuration of the simulated SSD.

SSD Organization	Host interface: PCIe 3.0 (NVMe 1.2) User capacity: 480 GB 8 channels, 4 dies-per-channel
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, 4096 blocks-per-plane, 2 planes-per-die
Flash Latencies [63]	Read latency: 75 µs, Program latency: 1300 µs, Erase latency: 3.8 ms

Workloads. We study a diverse set of storage traces collected from real enterprise and datacenter workloads, which are listed in Table 2 [7, 68, 69, 80]. We categorize these workloads as low-interference or high-interference, based on the average flash chip busy time, which captures the combined effect of all four types of interference. We find that a workload interferes highly with other workloads if it keeps all of the flash chips busy for more than 8% of its total execution time. We form MQ-SSD workloads using randomly-selected combinations of four low- and high-interference traces. We classify each workload based on the fraction of storage traces that are high-interference.

Metrics. We use four key metrics in our evaluations. The first metric, *F*, is *strict fairness* [22, 70, 73, 74], which we calculate using Eq. 2. The second metric, S_{Max}, is the *maximum slowdown* [17, 49, 50, 109] among all concurrently-running flows. The third metric, *STDEV*, is the *standard deviation of the slowdowns* of all concurrently-running flows. The fourth metric, *WS*, measures the *weighted speedup* [93] of MQ-SSD's response time (RT), which represents the overall efficiency of the I/O scheduler for all concurrent flows. Weighted speedup is calculated as:

$$WS = \sum_i \frac{RT_i^{alone}}{RT_i^{shared}} \quad (11)$$

where RT_i^{alone} is the average response time of transactions from flow *i* when the flow is run alone, and RT_i^{shared} is the average response time when the flow runs concurrently with other flows.

7. Evaluation

We compare FLIN to two state-of-the-art baselines. Our first baseline (`Sprinkler`) uses three state-of-the-art scheduling techniques: (1) Sprinkler for out-of-order transaction scheduling in the TSU [37], (2) semi-preemptive GC execution [52], and (3) read prioritization and write/erase suspension [112]. Our second baseline (`Sprinkler+Fairness`) combines Sprinkler with a state-of-the-art in-controller MQ-SSD fairness control mechanism [36].

7.1. Fairness and Performance of FLIN

Figures 10, 11, and 12 compare the fairness, maximum slowdown, and STDEV, respectively, of FLIN against the two baselines. For workload mixes comprised of 25%, 50%, 75%, and 100% high-interference flows, FLIN improves the average fairness by 1.8x/1.3x, 2.5x/1.6x, 5.6x/2.4x, and 54x/3.2x, respectively, compared to `Sprinkler`/`Sprinkler+Fairness` (Figure 10). For the same workload mixes, FLIN reduces the average maximum slowdown of the concurrently-running flows by 24x/2.3x, 1400x/5.5x, 3231x/12.x, and 1597x/18x, respectively (Figure 11), and reduces the STDEV of the slowdown of concurrently-running flows by an average of 137x/13x, 7504x/15x, 2784x/15x, and 2850x/21.5x (Figure 12). We make four key observations from the three figures. First, `Sprinkler`'s fairness decreases significantly as the fraction of high-interference flows in a workload increases, and approaches zero in workloads with 100% high-interference flows. Second, `Sprinkler+Fairness` improves fairness over `Sprinkler` due to its inclusion of fairness control, but `Sprinkler+Fairness` does *not* consider all sources of interference, and therefore has a much lower fairness than FLIN. Third, across all of our workloads, no flow has a maximum slowdown greater than 80x under FLIN, even though there are several flows that have maximum slowdowns over 500x with `Sprinkler` and `Sprinkler+Fairness`. This indicates that FLIN provides significantly better fairness over both baseline schedulers. Fourth, across all of our workloads, the STDEV of the slowdown is always lower than 10 with FLIN, while it ranges between 100 and 1000 for many flows under the baseline schedulers. This means that FLIN distributes the impact of interference more fairly across each flow in the workload. We conclude that FLIN's slowdown management scheme effectively improves MQ-SSD fairness over both state-of-the-art baselines.

Table 2: Characteristics of the evaluated I/O traces.

Trace	Read Ratio	Average Size (kB)		Avg. Inter Arrival (ms)		Interference Probability
		RD	WR	RD	WR	
dev [69]	0.68	21.4	31.5	4.1	6.1	Low
exch [68]	0.66	9.7	14.8	1.0	1.3	High
fin1 [7]	0.23	2.3	3.7	14.3	6.4	Low
fin2 [7]	0.82	2.3	2.9	11.1	10.8	Low
msncfs [69]	0.74	8.7	12.6	6.2	1.0	Low
msnfs [69]	0.67	10.7	11.2	0.9	0.4	High
prn-0 [80]	0.11	22.8	9.7	34.2	117.2	Low
prn-1 [80]	0.75	22.5	11.7	30	126.7	Low
prxy-0 [80]	0.03	8.3	4.6	9.1	49.4	Low
prxy-1 [80]	0.65	24.6	26.1	3.7	3.4	Low
rad-be [69]	0.18	106.2	11.7	3.6	13.5	Low
rad-ps [69]	0.10	10.3	8.2	54.2	21.5	Low
src1-0 [80]	0.56	36.2	52.0	11.8	21.7	High
src1-1 [80]	0.95	35.8	14.7	3.2	213.9	High
src1-2 [80]	0.25	19.1	32.5	56.5	405.6	Low
stg-0 [80]	0.15	24.9	9.2	4.6	350.3	Low
stg-1 [80]	0.64	59.5	7.9	7.4	746.2	Low
tpcc [68]	0.65	8.1	9.4	0.1	0.1	High
tpce [68]	0.92	8.0	12.6	0.1	0.1	High
wsrch [7]	0.99	15.1	8.6	3.0	1.2	Low

Figure 13 compares the weighted speedup of FLIN against the two baselines. We observe from the figure that for 25%, 50%, 75%, and 100% high-interference workload mixes, FLIN improves the weighted speedup by 38%/21%, 74%/32%, 132%/41%, and 156%/76%, on average, over `Sprinkler`/`Sprinkler+Fairness`. The improvement is a result of FLIN's fairness control mechanism, which removes performance bottlenecks resulting from high unfairness. FLIN

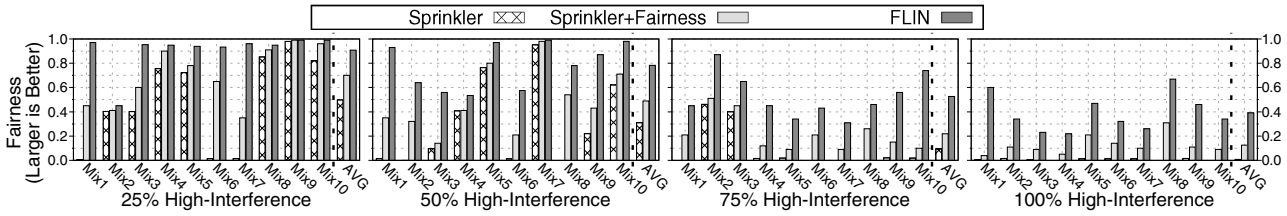


Figure 10: Fairness of FLIN vs. baseline schedulers.

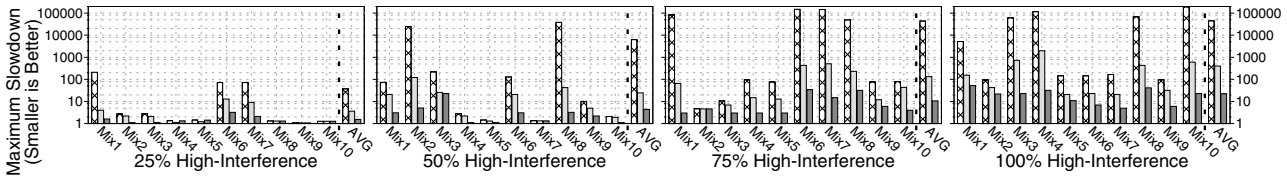


Figure 11: Maximum slowdown of FLIN vs. baseline schedulers.

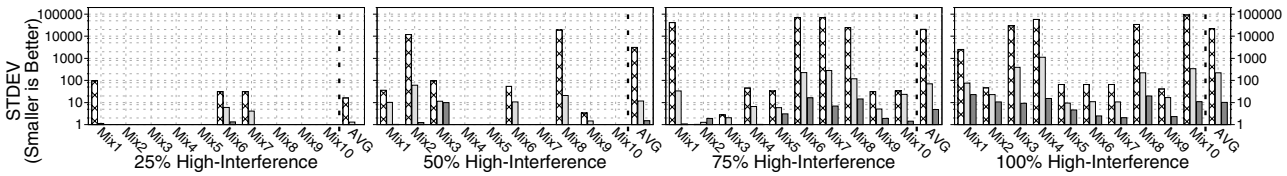


Figure 12: STDEV slowdown of FLIN vs. baseline schedulers.

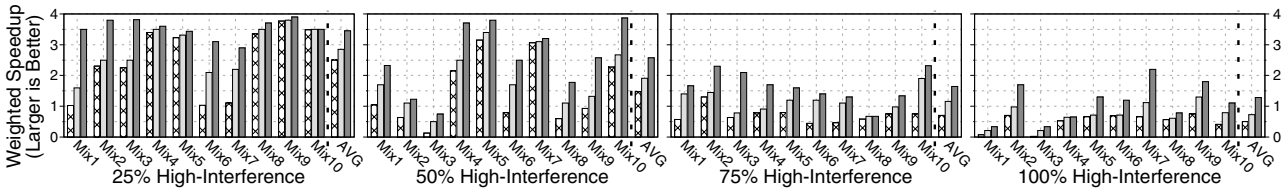


Figure 13: Weighted speedup of FLIN vs. baseline schedulers.

improves the performance of low-interference flows while throttling the performance of high-interference flows, which improves weighted speedup by enabling low-intensity flows to make fast progress. In contrast, `Sprinkler` typically prioritizes transactions from high-interference flows. This improves the performance of high-interference flows, but penalizes the low-interference flows and, thus, reduces the overall weighted speedup by causing all flows to progress relatively slowly. `Sprinkler+Fairness` improves the performance of low-interference flows by controlling the throughput of high-interference flows, but its weighted speedup remains low because the throughput control mechanism leaves many resources idle in the SSD [89]. We conclude that FLIN's fairness control mechanism significantly improves weighted speedup, an important metric for overall MQ-SSD performance, over state-of-the-art device-level schedulers.

7.2. Effect of Different FLIN Stages

We analyze the fairness, maximum slowdown, and weighted speedup improvements when using only (1) the first stage of FLIN (Section 4.1), or (2) the third stage of FLIN (Section 4.3), as these two stages are designed to reduce the four sources of interference listed in Section 3. Figure 14 shows the fairness (left), maximum slowdown (center), and weighted speedup of `Sprinkler`, Stage 1 of FLIN only, Stage 3 of FLIN only, and all three stages of FLIN. For each category, the figure shows the average value of each metric across all workloads in the category.

We make three observations from Figure 14. First, the individual stages of FLIN improve both fairness and performance over `Sprinkler`, as each stage works to reduce some sources

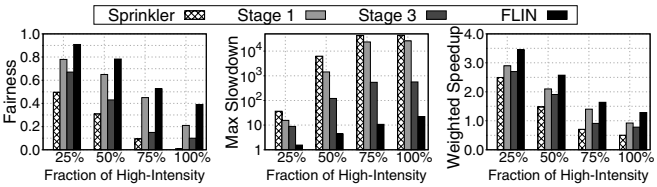


Figure 14: Effect of Stages 1 and 3 of FLIN.

of interference among concurrently-running flows, but each stage alone falls short of the improvements we observe when FLIN as a whole is used. Second, the fairness and performance improvements of Stage 1 are much higher than those of Stage 3. This is because I/O intensity is the most dominant source of interference. Since Stage 1 is designed to mitigate interference due to different I/O intensities, it delivers larger improvements than Stage 3. Third, Stage 3 reduces the maximum slowdown by a greater amount than Stage 1. This is because GC operations can significantly increase the stall time of transactions, causing a large increase in maximum slowdown, and Stage 3 manages such GC interference effectively. We conclude that Stages 1 and 3 of FLIN are effective at mitigating different sources of interference, and that FLIN makes effective use of both stages to maximize fairness and performance improvements over state-of-the-art schedulers.

7.3. Sensitivity to FLIN and MQ-SSD Parameters

We evaluate the sensitivity of fairness to four of FLIN's parameters, as shown in Figure 15: (1) the epoch length for determining I/O intensity (Section 5.1), (2) the threshold for high read intensity (α_C^{read}; Section 5.1), (3) the fairness threshold for

high-intensity transaction reordering (F_{thr}; Section 4.1), and (4) the GC management threshold (GC_{FLIN}; Section 4.3). We sweep the values of each parameter individually. We make four observations from the figure. First, for epoch length, we observe from Figure 15a that epoch lengths of 10–100 ms achieve the highest fairness, indicating that at these lengths, FLIN can best differentiate between low-intensity and high-intensity flows. Second, for α_C^{read}, we observe from Figure 15b that if α_C^{read} is too small (e.g., below 32 MB/s), fairness decreases, as low-intensity flows are *incorrectly* classified as high-intensity flows. Third, we observe in Figure 15c that for workloads where at least half of the flows are high intensity, fairness is maximized for moderate values of F_{thr} (e.g., F_{thr} = 0.6). This is because if F_{thr} is too low, FLIN does not prioritize the flow with the maximum slowdown very often (see Section 4.1, Step 2b), which keeps fairness low. If F_{thr} is too high, then FLIN almost always prioritizes the flow that currently has the maximum slowdown, which causes the other flows to stall for long periods of time, and increases their slowdown. Fourth, we find that fairness is *not* highly sensitive to the GC_{FLIN} parameter, though smaller values of GC_{FLIN} provide slightly higher fairness because they cause FLIN to defer GC operations for longer.

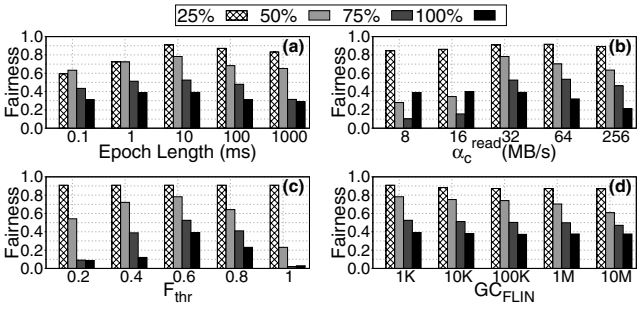


Figure 15: Fairness with FLIN configurations.

We also evaluate the sensitivity of FLIN's fairness and performance benefits to different MQ-SSD configurations. We make three observations from these studies (not shown). First, a larger page size decreases the number of transactions in the back end and, therefore, reduces the probability of interference. This reduces the benefits that FLIN can deliver. Second, newer flash memory technologies have higher flash read and write latencies [24, 100], which results in an increase in the busy time of the flash chips. This leads to a higher probability of interference, and increases both the fairness and performance benefits of FLIN. Third, having a larger number of memory chips reduces the interference probability, by distributing transactions over a larger number of chip-level queues, and thus reduces FLIN's ability to improve fairness.

7.4. Evaluation of Support for Flow Priorities

The priority-based queue arbitration stage of FLIN supports flow-level priorities. We evaluate FLIN's support for flow priorities in a variety of scenarios and present a representative case study to highlight its effectiveness. Figure 16 shows the slowdown of four `msnfs` flows concurrently running with priority levels 0, 1, 2, and 3. With FLIN, we observe that the highest-priority flow (the flow assigned priority level 3; `msnfs-3` in Figure 16) has the lowest slowdown, as the transactions of this flow are assigned more than half of the scheduling slots. The slowdown of each lower-priority version of `msnfs` increases proportionally with the flow's priority.

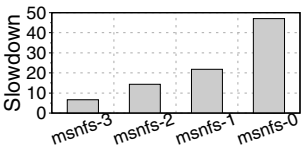


Figure 16: Effects of varying flow priorities.

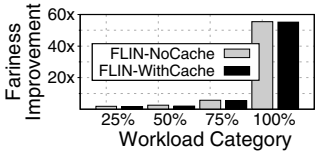


Figure 17: Effect of write caching.

7.5. Effect of Write Caching

Modern MQ-SSDs employ a front end DRAM write buffer in order to bridge the performance gap between the fast host DRAM and the slow SSD memory chips. To investigate the effects of write caching on fairness, we conduct a set of experiments assuming a 128 MB write cache per running flow (a total of 512 MB write cache). Figure 17 shows the average fairness improvement of FLIN with and without a write buffer. We observe that caching has only a minimal effect on FLIN's fairness benefits, for two reasons. First, write caching has no immediate effect on read transactions, so both I/O intensity and access pattern interference are *not* affected when write caching is enabled. Second, high-interference flows typically have high I/O intensity. When the write intensity of a flow is high, the write cache fills up and cannot cache any more requests. At this point, known as the *write cliff*, any write requests that cannot be cached are directly issued to the back end [39], eliminating the benefits of the write cache. We conclude that write caching has only a minimal effect on the performance of FLIN.

8. Related Work

To our knowledge, this is the first work to (1) thoroughly analyze the sources of interference among concurrently-running I/O flows in MQ-SSDs; and (2) propose a new device-level I/O scheduler for modern SSDs that effectively mitigates interference among concurrently-running I/O flows to provide both high fairness and performance. Providing fairness and performance isolation among concurrently-running I/O flows is an active area of research for modern SSDs [2, 28, 36, 45, 46, 84, 89, 94, 120], especially for modern MQ-SSD devices that support new host–interface protocols [36, 101], such as the NVMe protocol.

Fair Disk Schedulers. Many works [3,9,34,86,95,113,116] propose OS-level disk schedulers to meet fairness and real-time requirements in hard drives. These approaches are not suitable for SSDs as they (1) essentially estimate I/O access latencies based on the physical structure of rotating disk drives, which is different in SSDs; and (2) assume read/write latencies that are orders of magnitude longer than those of SSDs [47].

More recent OS-level schedulers have evolved to better fit the internal characteristics of SSDs [84, 89, 120]. FIOS [84] periodically assigns an equal number of time-slices to the running flows. The flows then trade these time-slices for scheduling I/O requests based on the flash read/write service time costs. BCQ [120] adopts a similar approach with a higher accuracy in determining read/write costs. Both FIOS and BCQ suffer from reduced responsiveness of the storage device and reduced overall performance [89]. More precisely, a flow that consumes its time budget early would have to wait for other flows to finish before its budget is replenished in the next round, leading to a period of unresponsiveness and resource idleness [89]. FlashFQ [89] is another OS-level scheduler that eliminates such unresponsiveness. FlashFQ estimates the progress of each running flow based on the cost of flash read/write operations, and allows a flow to dis-

patch I/O requests as long as its progress is not much further ahead of other flows. If a flow's progress is far ahead of other concurrently-running flows, FlashFQ throttles the flow that is far ahead to improve fairness.

FIOS, BCQ, and FlashFQ suffer from two major shortcomings compared to FLIN. First, they do *not* have information about the internal resources and mechanisms of an SSD, and only speculate about resource availability and the behavior of SSD management algorithms when making a scheduling decision. Such speculation-based scheduling decisions can be suboptimal for mitigating GC and parallelism interference, as both types of interference are highly dependent on internal SSD mechanisms. In contrast, FLIN has *direct information* about the status of pending requests (in the I/O queue) and the internal SSD resources/mechanisms. Second, OS-level techniques have *no direct control* over the SSD's internal mechanisms. Therefore, it is likely that there is some inter-flow interference within the SSD, *even when* the OS-level scheduler makes use of fairness control mechanisms. For example, the TSU may execute GC activities due to a high-GC flow, which can block the transactions of a low-GC flow. In contrast, FLIN takes GC activities into account (e.g., Algorithm 2, which works to minimize GC interference), and controls *all* major sources of interference in an MQ-SSD.

While our goal is to design FLIN to exploit the many advantages of performing scheduling and fairness control in the SSD instead of the OS, FLIN can also work in conjunction with OS-level I/O resource control mechanisms to exploit the best of both worlds. For example, virtualized environments require OS-level management to control I/O requests across multiple VMs/containers [43]. One could envision a system where the OS and FLIN work together to perform coordinated scheduling and fairness control. For example, the OS could make application-/VM-aware scheduling and fairness decisions, while FLIN performs resource-aware scheduling/fairness decisions. We leave an exploration of coordinated OS/FLIN mechanisms to future work.

Fair Main Memory Scheduling. There is a large collection of prior works that propose fair memory request schedulers for DRAM [4,18,19,44,49,50,71–75,81,96–99,105]. These schedulers *cannot* be directly used in SSDs, as DRAM devices (1) operate at a much smaller granularity than SSDs (64 B in DRAM vs. 8 kB in SSDs); (2) have symmetric read and write latencies; (3) do *not* perform garbage collection; (4) have row buffers in banks that significantly affect performance; and (5) perform periodic refresh operations.

SSD Performance Isolation. Performance isolation [15, 25, 28, 29, 46, 78, 91, 94] aims to provide predictable performance by eliminating interference from different tenants that share an SSD. Inter-tenant interference can be reduced by (1) assigning different hardware resources to different tenants [28, 29, 46, 78, 94] (e.g., different tenants are mapped to different channels/dies), or (2) using isolation-aware algorithms for time sharing the hardware resources between tenants [15, 25, 91, 94] (e.g., credit-based algorithms).

These techniques have three main differences from FLIN. First, they do *not* provide fairness control. Instead, they reduce the interference between flows *without* considering any fairness metric (e.g., slowdown), and do not evaluate fairness quantitatively. Second, they are *not* designed to optimize the utilization of the MQ-SSD resources, which limits their performance. Third, no existing approach considers all four interference sources considered by FLIN.

High-Performance Device Schedulers. The vast majority of device-level I/O schedulers [20, 37, 40, 77, 90] try to effectively exploit SSD internal parallelism with out-of-order execution of flash transactions. These schedulers are designed for performance, and are vulnerable to inter-flow interference.

Performance Evaluation of NVMe SSDs. A number of prior works focus on the performance evaluation and implementation issues of MQ-SSDs [5, 114, 115]. Xu et al. [114] analyze the performance implications of MQ-SSDs for modern database applications. Awad et al. [5] explore different implementation aspects of NVMe via simulation, and show how these aspects affect system performance. None of these works consider the fairness issues in MQ-SSDs.

9. Conclusion

We propose FLIN, a lightweight transaction scheduler for modern multi-queue SSDs (MQ-SSDs), which provides fairness among concurrently-running flows. FLIN uses a three-stage design to protect against all four major sources of interference that exist in real MQ-SSDs, while enforcing application-level priorities that are assigned by the host. Our extensive evaluations show that FLIN effectively improves both fairness and system performance compared to state-of-the-art device-level schedulers. FLIN is simple to implement within the SSD controller firmware, requires no additional hardware, and consumes less than 0.06% of the storage space available in the in-SSD DRAM. We conclude that FLIN is a promising scheduling mechanism that enables the design of fair and high-performance MQ-SSDs.

Acknowledgments

We thank the anonymous referees of ISCA 2018 and ASPLOS 2018. We thank SAFARI group members for feedback and the stimulating research environment. We thank our industrial partners, especially Alibaba, Google, Huawei, Intel, Microsoft, and VMware, for their generous support. Lois Orosa was supported by FAPESP fellowship 2016/18929-4.

References

[1] "MQSim GitHub Repository," https://github.com/CMU-SAFARI/MQSim.
[2] S. Ahn *et al.*, "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-Core Systems," in *HotStorage*, 2016.
[3] W. G. Aref *et al.*, "Scalable QoS-Aware Disk-Scheduling," in *IDEAS*, 2002.
[4] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[5] A. Awad *et al.*, "Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems," in *ISPASS*, 2015.
[6] J. Axboe, "Linux Block I/O—Present and Future," in *Ottawa Linux Symp.*, 2004.
[7] K. Bates and B. McNutt, "UMass Trace Repository."
[8] M. Bjørling *et al.*, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems," in *SYSTOR*, 2013.
[9] J. Bruno *et al.*, "Disk Scheduling with Quality of Service Guarantees," in *ICMCS*, 1999.
[10] Y. Cai *et al.*, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, 2017.
[11] Y. Cai *et al.*, "Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation," in *DSN*, 2015.
[12] Y. Cai *et al.*, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.
[13] Y. Cai *et al.*, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs:AR], 2017.
[14] Y. Cai *et al.*, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
[15] D.-W. Chang *et al.*, "VSSD: Performance Isolation in a Solid-State Drive," *TODAES*, 2015.
[16] L.-P. Chang *et al.*, "Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems," *TECS*, 2004.
[17] R. Das *et al.*, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.
[18] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[19] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.
[20] N. Elyasi *et al.*, "Exploiting Intra-Request Slack to Improve SSD Performance," in *ASPLOS*, 2017.
[21] G2M Research, "NVMe Market Forecast & Vendor Report Abstract," 2017.

[22] R. Gabor *et al.*, "Fairness and Throughput in Switch on Event Multithreading," in *MICRO*, 2006.
[23] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *CSUR*, 2005.
[24] L. M. Grupp *et al.*, "The Bleak Future of NAND Flash Memory," in *FAST*, 2012.
[25] A. Gulati *et al.*, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *FAST*, 2009.
[26] J. Guo *et al.*, "Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance," in *IPDPS*, 2017.
[27] S. S. Hahn *et al.*, "To Collect or Not to Collect: Just-in-Time Garbage Collection for High-performance SSDs with Long Lifetimes," in *DAC*, 2015.
[28] J. Huang *et al.*, "FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs," in *FAST*, 2017.
[29] S.-M. Huang and L.-P. Chang, "Providing SLO Compliance on NVMe SSDs Through Parallelism Reservation," *TODAES*, 2018.
[30] Huawei Technologies Co., Ltd., "Huawei ES3000 V3 NVMe SSD White Paper," 2017.
[31] Intel Corp., "Intel Solid-State Drive DC P3600 Series, Product Specification," 2014.
[32] Intel Corp., "Intel Solid-State Drive DC P3500 Series, Product Specification," 2015.
[33] Intel Corp., "Intel 3D NAND SSD DC P4500 Series, Product Brief," 2018.
[34] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," in *SOSP*, 2001.
[35] W. Jin *et al.*, "Interposed Proportional Sharing for a Storage Service Utility," in *SIGMETRICS*, 2004.
[36] B. Jun and D. Shin, "Workload-aware Budget Compensation Scheduling for NVMe Solid State Drives," in *NVMSA*, 2015.
[37] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing Resource Utilization in Many-chip Solid State Disks," in *HPCA*, 2014.
[38] M. Jung *et al.*, "HIOS: A Host Interface I/O Scheduler for Solid State Disks," in *ISCA*, 2014.
[39] M. Jung and M. Kandemir, "Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications," in *SIGMETRICS*, 2013.
[40] M. Jung *et al.*, "PAQ: Physically Addressed Queuing for Resource Conflict Avoidance in Solid State Disk," in *ISCA*, 2012.
[41] J.-U. Kang *et al.*, "The Multi-streamed Solid-state Drive," in *HotStorage*, 2014.
[42] S. Kavalanekar *et al.*, "Characterization of Storage Workload Traces from Production Windows Servers," in *IISWC*, 2008.
[43] H.-J. Kim *et al.*, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSD," in *HotStorage*, 2016.
[44] H. Kim *et al.*, "Bounding Memory Interference Delay in COTS-Based Multi-Core Systems," in *RTAS*, 2014.
[45] J. Kim *et al.*, "I/O Scheduling Schemes for Better I/O Proportionality on Flash-Based SSDs," in *MASCOTS*, 2016.
[46] J. Kim *et al.*, "Towards SLO Complying SSDs Through OPS Isolation," in *FAST*, 2015.
[47] J. Kim *et al.*, "Disk Schedulers for Solid State Drivers," in *EMSOFT*, 2009.
[48] T. Y. Kim *et al.*, "Improving Performance by Bridging the Semantic Gap Between Multi-queue SSD and I/O Virtualization Framework," in *MSST*, 2015.
[49] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[50] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[51] J. Lee *et al.*, "Preemptible I/O Scheduling of Garbage Collection for Solid State Drives," *TCAD*, 2013.
[52] J. Lee *et al.*, "A Semi-preemptive Garbage Collector for Solid State Drives," in *ISPASS*, 2011.
[53] Y. Li *et al.*, "Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization," in *SIGMETRICS*, 2013.
[54] M. Lin and S. Chen, "Efficient and Intelligent Garbage Collection Policy for NAND Flash-based Consumer Electronics," *TCE*, 2013.
[55] Linux Kernel Organization, Inc., "Block IO Priorities," https://www.kernel.org/doc/Documentation/block/ioprio.txt.
[56] Linux Kernel Organization, Inc., "CFQ (Complete Fairness Queueing)," https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.
[57] Y. Luo *et al.*, "WARM: Improving NAND Flash Memory Lifetime With Write-Hotness Aware Retention Management," in *MSST*, 2015.
[58] Y. Luo *et al.*, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
[59] Y. Luo *et al.*, "HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness," in *HPCA*, 2018.
[60] Marvell Technology Group, Ltd., "Marvell 88NV9145 PCIe-NAND controller," 2012.
[61] Marvell Technology Group, Ltd., "Marvell 88SS1093 Flash Memory Controller," 2017.
[62] Micron Technology, Inc., "64Gb, 128Gb, 256Gb, 512Gb Asynchronous/Synchronous NAND," 2009.
[63] Micron Technology, Inc., "NAND Flash Memory MLC MT29F256G08CKCAB Datasheet," 2014.
[64] Micron Technology, Inc., "128Gb, 256Gb, 512Gb Async/Sync Enterprise NAND," 2016.
[65] Micron Technology, Inc., "9100 U.2 and HHHL NVMe PCIe SSDs," 2016.
[66] Micron Technology, Inc., "Micron 9200 NVMe SSDs," 2016.
[67] Micron Technology, Inc., "MLC 128Gb to 2Tb Enterprise Async/Sync NAND," 2016.
[68] Microsoft Corp., "Microsoft Enterprise Traces," http://iotta.snia.org/traces/130.
[69] Microsoft Corp., "Microsoft Production Server Traces," http://iotta.snia.org/traces/158.
[70] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[71] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-Core DRAM Controllers," in *PODC*, 2008.
[72] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *MICRO*, 2011.
[73] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[74] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[75] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2015.
[76] J. Nagle, "On Packet Switches with Infinite Storage," *TCOM*, 1985.
[77] E. H. Nam *et al.*, "Ozone (O3): An Out-of-order Flash Memory Controller Architecture," *TC*, 2011.
[78] M. Nanavati *et al.*, "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage," in *NSDI*, 2017.
[79] D. Narayanan *et al.*, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ToS*, 2008.
[80] D. Narayanan *et al.*, "Migrating Server Storage to SSDs: Analysis of Tradeoffs," in *Eurosys*, 2009.
[81] K. J. Nesbit *et al.*, "Fair Queuing Memory Systems," in *MICRO*, 2006.
[82] D. M. Nguyen *et al.*, "Solving the Multidimensional Assignment Problem by a Cross-Entropy Method," *JCO*, 2014.
[83] NVM Express Workgroup, "NVM Express Specification, Revision 1.2," 2014.
[84] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *FAST*, 2012.
[85] W. P. Pierskalla, "The Multidimensional Assignment Problem," *OR*, 1968.
[86] P. E. Rocha and L. C. E. Bona, "A QoS Aware Non-Work-Conserving Disk Scheduler," in *MSST*, 2012.
[87] Samsung Electronics Co., Ltd., "Samsung SSD 960 PRO M.2, Data Sheet," 2017.
[88] SATA-IO, "Serial ATA Revision 3.3," http://www.sata-io.org, 2016.
[89] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *ATC*, 2013.
[90] J.-Y. Shin *et al.*, "Exploiting Internal Parallelism of Flash-based SSDs," *CAL*, 2010.
[91] D. Shue *et al.*, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *OSDI*, 2012.
[92] SK Hynix Inc., "F26 32Gb MLC NAND Flash Memory TSOP Legacy," 2011.
[93] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.
[94] X. Song *et al.*, "Architecting Flash-Based Solid-State Drive for High-Performance I/O Virtualization," *CAL*, 2014.
[95] M. J. Stanovich *et al.*, "Throttling On-Disk Schedulers to Meet Soft-Real-Time Requirements," in *RTAS*, 2008.
[96] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[97] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
[98] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[99] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
[100] K. Suzuki and S. Swanson, "A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014," in *IMW*, 2015.
[101] A. Tavakkol *et al.*, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices," in *FAST*, 2018.
[102] A. Tavakkol *et al.*, "Performance Evaluation of Dynamic Page Allocation Strategies in SSDs," *ToMPECS*, 2016.
[103] Toshiba Corp., "OCZ RD400/400A Series, Product Brief," 2016.
[104] Toshiba Corp., "PX04PMC Series, Data Sheet," 2016.
[105] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *TACO*, 2016.
[106] P. Valente and F. Checconi, "High Throughput Disk Scheduling with Fair Bandwidth Distribution," *TC*, 2010.
[107] B. Van Houdt, "A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives," in *SIGMETRICS*, 2013.
[108] B. Van Houdt, "On the Necessity of Hot and Cold Data Identification to Reduce the Write Amplification in Flash-based SSDs," *Perform. Eval.*, 2014.
[109] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-Threaded Processors," *CAL*, 2011.
[110] Western Digital Technologies, Inc., "Skyhawk & Skyhawk Ultra NVMe PCIe SSD, Data Sheet," 2017.
[111] Western Digital Technologies, Inc., "Ultrastar SN200 Series, Data Sheet," 2017.
[112] G. Wu and X. He, "Reducing SSD Read Latency via NAND Flash Program and Erase Suspension," in *FAST*, 2012.
[113] J. C. Wu *et al.*, "Hierarchical Disk Sharing for Multimedia Systems," in *NOSSDAV*, 2005.
[114] Q. Xu *et al.*, "Performance Characterization of Hyper-scale Applications on NVMe SSDs," in *SIGMETRICS*, 2015.
[115] Q. Xu *et al.*, "Performance Analysis of NVMe SSDs and their Implication on Real World Databases," in *SYSTOR*, 2015.
[116] Y. Xu and S. Jiang, "A Scheduling Framework That Makes Any Disk Schedulers Non-Work-Conserving Solely Based on Request Characteristics," in *FAST*, 2011.
[117] S. Yan *et al.*, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *FAST*, 2017.
[118] S. Yang *et al.*, "Split-Level I/O Scheduling," in *SOSP*, 2015.
[119] J. Zhang *et al.*, "Synthesizing Representative I/O Workloads for TPC-H," in *HPCA*, 2004.
[120] Q. Zhang *et al.*, "An Efficient, QoS-Aware I/O Scheduler for Solid State Drive," in *HPCC EUC*, 2013.