


BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era

Zebin Ren
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Krijn Doekemeijer
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Nick Tehrany*
BlueOne Business Software LLC
Beverly Hills, California, USA

Animesh Trivedi
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

ABSTRACT

Flash SSDs have become the de-facto choice to deliver high I/O performance to modern data-intensive workloads. These workloads are often deployed in the cloud, where multiple tenants share access to flash-based SSDs. Cloud providers use various techniques, including I/O schedulers available in the Linux kernel, such as *BFQ*, *Multiqueue-Deadline* (MQ-Deadline), and Kyber, to ensure certain performance qualities (i.e., service-level agreements, SLAs). Though designed for fast NVMe SSDs, there has not been a systematic study of these schedulers for modern, high-performance SSDs with their unique challenges. In this paper. we systematically characterize the performance, overheads, and scalability properties of Linux storage schedulers on NVMe SSDs with millions of I/O operations/s. We report 23 observations and 5 key findings that indicate that (i) CPU performance is the primary bottleneck with the Linux storage stack with high-performance NVMe SSDs; (ii) Linux I/O schedulers can introduce 63.4% performance overheads with NVMe SSDs; (iii) Kyber and BFQ can deliver 99.3% lower P99 latency than None or MQ-Deadline schedulers in the presence of multiple interfering workloads. We open-source the scripts and datasets of this work at: https://zenodo.org/records/10599514.

CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → **Secondary storage**.

KEYWORDS

Linux storage scheduler, Quality of service, Measurements, NVMe

ACM Reference Format:
Zebin Ren, Krijn Doekemeijer, Nick Tehrany, and Animesh Trivedi. 2024. BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645053

*Work done while the author was at Vrije Universiteit Amsterdam.

ICPE '24, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645053

1 INTRODUCTION

Modern flash-based NVMe solid-state drives (SSDs) are able to deliver millions of I/O operations per second (IOPS) and single-digit microsecond-level latency [7, 11]. These SSDs are widely used in multi-tenant cloud environments where their performance, bandwidth, and latency are shared among multiple tenants or workloads [3, 5, 13, 28, 32, 38, 43, 44]. In multi-tenant cloud environments, there is commonly a scheduler of I/O requests designed to deliver fairness with quality-of-service guarantees, also known as service level agreements (SLA), for the cloud services [38, 44].

Designing a fair, high-performance, low-overhead I/O scheduler has been a topic of extensive research over the past decade, with studies focusing on providing proportional performance sharing with request scheduling [1, 26, 40, 46, 52, 56, 57], low latency guarantees [27, 39, 42], and SSD-supported features acceleration [29, 31, 54]. Despite these studies, the emergence of high-performance NVMe SSDs has created multiple unique challenges for storage schedulers (or I/O schedulers) that have not been systematically studied or characterized. These unique challenges come from three distinct dimensions that we cover in this study: (i) performance overheads from the complex storage software stack on fast SSDs; (ii) scalability challenges in the presence of highly concurrent I/O operations on NVMe SSDs; and (iii) interference management among competing I/O requests at high CPU loads.

Firstly, *state-of-the-practice* I/O schedulers available in Linux (BFQ, MQ-Deadline, and Kyber) are not yet studied at the performance scale of millions of IOPS which modern NVMe SSDs can deliver. At this scale, small overheads from the Linux storage stack become a performance bottleneck [51, 55]. These overheads come from operational complexity (dispatching, merging, sorting, and staging I/O requests) that determine the maximum performance a scheduler can deliver. Various enterprise software recommends using no scheduler (also known in Linux as None) on high-performance SSDs to limit or eliminate these overheads [49, 50]. However, in this work, we demonstrate that this decision can sacrifice the quality of service and fairness among workloads. Secondly, modern server machines are highly parallel with multiple CPU cores and SSDs with parallel I/O queues. Here, overheads related to locking, synchronization, and queue management become the key performance bottlenecks. Thirdly, the widely-adopted flash-based SSDs have different read and write performance characteristics, and the reads and writes interfere with each other [20, 25]. This interference creates challenges for I/O schedulers to fairly schedule mixed read-write workloads. Furthermore, a shared environment

brings new challenges to I/O schedulers [30, 32, 45] since workloads have diverse I/O requirements. For example, throughput-bound workloads like batch analytics (e.g., Spark) require high IOPS, and real-time workloads like interactive queries [16] expect low, predictable latencies. I/O schedulers need to provide SLA guarantees to satisfy these requirements across all the tenants, possibly simultaneously [37, 41, 47, 48]. There have been attempts to design SSD-aware schedulers [27, 46, 52, 56] (including Linux Kyber [8]). However, the performance of these *state-of-the-practice* schedulers (available in the Linux kernel) has not been studied or quantified on high-performance NVMe SSDs with millions of IOPS. To summarize, the high performance, scale, and interference properties of modern NVMe SSDs motivate us to systematically study and characterize the performance of Linux I/O schedulers.

In this study, we aim to measure, quantify, and analyze overheads from three widely available Linux storage I/O schedulers, BFQ [1], Kyber [8], and MQ-Deadline [9] on flash-based NVMe SSDs (§2). We start our investigation by studying the performance (expressed as IOPS or latencies) using fio, a widely used microbenchmark with the high-performance `io_uring` storage engine [19] on our 8 NVMe SSDs setup with a peak performance of 5.9 million 4KiB IOPS (see Table 1). We specifically motivate and investigate the following research questions (**RQ**) around the performance and overheads of Linux I/O schedulers with the key findings (**KF**):

(RQ1) What is the CPU overhead for modern fast NVMe SSD devices? Can the Linux storage stack saturate multiple NVMe SSDs? This research question is important to establish the baseline performance of the SSDs around which the rest of the scheduler analysis is done. The key motivation for this question is to establish under which configurations the CPU cores or SSDs become a bottleneck, and how they influence the performance of I/O schedulers. **KF-1:** *for high-performance NVMe SSDs, the CPU is the key bottleneck (even with the None scheduler), thus making the scheduling "efficiency" of I/O schedulers a key factor in the performance delivered to the workloads* (§3).

(RQ2) What are the scheduler overheads, and how do these overheads scale with the I/O and device concurrency? NVMe devices have deep I/O queues with multiple parallel queues. Hence, typically it takes multiple requests to saturate a single device. Here we study the overhead of I/O scheduling and how it scales with concurrent requests when requests come from a single workload (intra-process) and multiple workloads (inter-process). **KF-2:** *once the CPU becomes the bottleneck, the Linux I/O schedulers can induce 63.4% overheads in throughout and 50× increase in the P99 latencies over the None scheduler. We also report that the presence of multiple SSDs helps to reduce overheads associated with the device-specific locking and synchronization overheads* (§4).

(RQ3) How can I/O schedulers help to control interference in the presence of competing workloads, specifically latency-vs-bandwidth and read-vs-write? These kinds of mixed workloads are quite common in the cloud/enterprise settings. Hence, it is important to study what are the scheduler overheads in this scenario. The extent of the interference typically governs what kind of SLAs (99 percentile, worst-case performance scenarios) storage service providers can offer to their tenants. **KF-3** and **KF-4:** *Kyber and BFQ can provide good bounded performance in the presence of interference when the CPU is not the bottleneck, however, BFQ suffers from performance scalability overheads. Hence, we conclude that overall Kyber is the best fit Linux I/O schedulers for SSDs* (§5).

Table 1: Details of the benchmarking environment.

Component	*Configuration*
CPU	Single socket Intel(R) Xeon(R) Silver 4210R CPU 10 cores @ 2.40GHz, Hyper-threading disabled, Turbo disabled.
Memory	256GB, DDR4.
Storage	8× Samsung 980 PRO 1TB, Average latency (r/w): ~68/15 μs, peak random read IOPS: 1M@4KiB/device at the queue depth 32.
Software	Ubuntu 20.04 with Linux kernel `v6.3.8` (released April'23), fio v3.35, SPDK 22.09.

(RQ4) How do a scheduler configuration parameters affect the schedulers' behavior on competing workloads? The Linux I/O schedulers provide tunable parameters, which affect the schedulers' behavior. Based on our empirical findings, we further present a detailed analysis of the Kyber scheduler which is specifically designed for modern SSDs. It has two unique configuration parameters: read and write target latencies. We perform a configuration space exploration for Kyber. **KF-5:** *Kyber can be configured to prioritize latency or total throughput by tuning its read and write target latencies, but not both* (§6).

Our key contributions in this work include:

- To the best of our knowledge, this is the first-of-its-kind systematic study about overhead quantification and characterization of *state-of-the-practice* I/O schedulers with modern NVMe SSDs, exploring their performance, scalability, and interference patterns, resulting in 23 observations and 5 key findings.
- We explore the configuration space of Kyber, an SSD-optimized scheduler. We report that Kyber has the least amount of CPU overhead, and it can provide bounded performance in the presence of read/write interference.
- To facilitate reproduction, we open-source the design and implementation of our code and datasets at https://github.com/stonet-research/icpe24_io_scheduler_study_artifact. Permanent link: https://zenodo.org/records/10599514.

2 BENCHMARKING ENVIRONMENT

In this section, we present details about the benchmarking environment, workloads used, and selected Linux storage schedulers.

2.1 Hardware and Software

We use fio [4] as the workload generator with the `io_uring` I/O engine [14, 19, 51]. Our setup is able to deliver a peak random read performance of 3.4 Million IOPS with the Linux storage stack under Linux v6.3.8 (5.9 Million IOPS with SPDK that by-passes the kernel), and an average read latency of 68 μs (4KiB, with a queue depth of 1, QD=1); hence, creating a unique opportunity to study I/O schedulers in this high-performance I/O setup. We use three metrics to evaluate the performance of the Linux I/O schedulers: throughput, latency, and CPU usage. We measure the

throughput as I/O operations completed per second (IOPS). Latency measurements focus on the tail latency where we report 99 percentile latencies (P99) with a complete CDF distribution. The average CPU usage is measured using fio, which reports "the CPU time used by the process/run time". CPU usage = 1 means that the process uses a whole CPU core. fio gets the resource usage with the Linux `getrusage` system call [6]. We precondition the flash SSDs according to [18], by sequentially writing the entire device, then randomly writing 4KiB blocks with a total of 2TiB data. Each device can deliver 1 MIOPS@4KiB random read according to the specification. However, we only get ~770 KIOPS with 4KiB random read workload after the preconditioning.

2.2 Workload Patterns and Methodology

We focus on two kinds of applications in this work, latency-sensitive and throughput-bound. We use L-app to represent latency-sensitive applications such as database queries and T-app to represent throughput-bound applications such as batch processing jobs like Map-Reduce. We use three kinds of synthesized workloads to simulate these two kinds of applications:

(1) **L-app** (latency-sensitive application) generates requests of 4KiB block sizes with an I/O depth of 1 (only one outstanding request at a time, we also refer to the I/O depth as queue depth or QD in the following sections).
(2) **T-4KiB-app** (small I/O, throughput-bound application with 4KiB block size) generates requests of 4KiB block sizes with an I/O depth of 128. With 4KiB block size, a single core can not saturate the evaluated Samsung SSD on our setup. We use T-4KiB-apps to show the effect of I/O schedulers on I/O performance when the CPU is the bottleneck.
(3) **T-64KiB-app** (large I/O, throughput-bound application with 64KiB block size) generates requests of 64KiB block sizes with an I/O depth of 128. With the 64KiB block size, the evaluated Samsung SSD can be saturated with a single CPU core in our setup for all I/O schedulers. We use T-64KiB-apps to show the effect of I/O schedulers when the SSD is the bottleneck.

Experiments with read-only workloads run for 150 seconds (2 minutes + 30 seconds warm-up time) since the read performance of flash-based SSDs is stable. For applications that issue writes, we run each experiment for 12 minutes (6 minutes + 6 minutes warm-up time) with 5 repetitions to get stable results, we report both the average value and standard deviation.

2.3 I/O Schedulers Under Study

The Linux kernel has four multi-queue enabled I/O schedulers: None, BFQ [1], Kyber [8], and MQ-Deadline [9]. For this study, we use the default configuration of each scheduler as they are the most likely used configurations for real workloads. For Kyber we further explore the configuration space to synthesize guidelines (§6).

None is the default I/O scheduler for NVMe devices that is recommended often to reduce the scheduling overheads with fast NVMe storage devices [49, 50]. Technically, None is not an I/O scheduler since it dispatches I/O requests to the NVMe driver immediately when it gets a new request without reordering the requests. Due to its simplicity, None has the lowest overheads among all I/O schedulers. Hence, we select it as the baseline scheduler.

MQ-Deadline is the multi-queue version of the Deadline scheduler [2]. The main goal of MQ-Deadline is to guarantee the start service time for a request. MQ-Deadline maintains two read-write queue pairs, a sorted queue pair and a FIFO queue pair. MQ-Deadline issues I/O requests in increasing sector orders (from the sorted queue) unless there is a request that violates the service deadline (from the FIFO queue). When the service deadline is violated, MQ-Deadline issues the request from the FIFO queue.

BFQ (Budget Fair Queuing) is a proportional-sharing I/O scheduler that is designed to provide fair bandwidth sharing and low latency for latency-sensitive applications. BFQ associates each process with an internal request queue and a budget according to each process' weight in the number of sectors. BFQ uses worst-case fair weighted fair queuing+ (WF2Q+) [15] to select the next queue to service and exclusive device access is given to the selected queue until its I/O budget is used up or a timeout happens. To provide low latency for real-time applications such as video players, BFQ uses heuristics to detect applications that are sensitive to latency and gives them higher priority. BFQ is the most complex Linux kernel I/O scheduler (BFQ is ~10,000 LOC, against ~1,000 LOC for Kyber and MQ-Deadline), and is believed to have the highest overhead among the Linux I/O schedulers [1, 12].

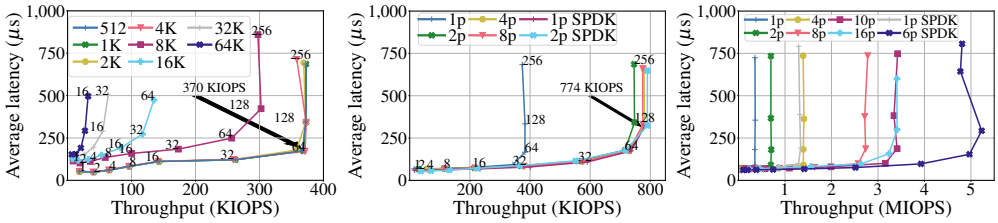
Kyber is designed for fast multi-queue devices to deliver low latency for reads. It is based on a heuristic that a process that issues a read I/O request usually waits for the request to finish and the data to be available (synchronous completion). In contrast, a process that issues a write I/O request usually can continue executing before the writes are finished (asynchronous completion). Thus, Kyber prioritizes reads over writes, but not to the extent where writes are starved. To achieve high responsiveness, Kyber prevents requests from building up on the device side with tokens. A detailed description of how Kyber works is presented in §6.

3 BOTTLENECK ANALYSIS: CPU OR NVME

We start our analysis by characterizing the performance of Samsung NVMe devices with the None scheduler to explore how CPU or NVMe devices become a bottleneck. Figure 1 shows our results. In the graphs, each line (same block size, increasing QD) has a hook shape: as QD increases, the throughput grows fast (x-axis), while the latency remains stable. At a certain turning point, the latency starts to grow fast with throughput remaining the same (because of queuing delays). This turning point is called the **saturation point** or **knee point**. At the saturation point, either the CPU or the SSD becomes the bottleneck.

3.1 What throughput and latency can a single SSD and a single CPU core deliver?

To answer this question, we configure a single SSD with a single CPU core and issue a random read workload. We measure the throughput and latency as we increase the number of outstanding requests (i.e., queue depth). Figure 1a shows the throughput as IOPS (x-axis) with the average latency (y-axis, the lower the better) for multiple queue depths (points on the lines). In this setting, we report that the performance of request sizes smaller than or equal to 4KiB are similar in nature, all saturating at a queue depth of 64 with 370 KIOPS as the peak performance. After this point,



(a) Vary request sizes, 1 SSD. **(b) Vary # processes with 1 SSD.** **(c) Vary # processes with 8 SSDs.**

Figure 1: Throughput and latency of a Samsung 980 PRO 1TB with varying queue depth and number of processes.

increasing the queue depth leads to further queuing delays, hence an increase in the latency. *At this point, the CPU is the primary bottleneck as it is 100% utilized, yet the SSD device itself is not saturated* (**Observation-1, O-1**). For request sizes greater than 4KiB, the throughput decreases proportionally, and the CPU gets saturated at shallow queue depths, hence, the latencies start to increase.

3.2 How does the performance scale as we increase the number of CPU cores or SSDs?

In the subsequent experiments, we increase the number of CPU cores with one SSD (Figure 1b) and then the number of SSDs (Figure 1c). We observe in Figure 1b that with 4 cores (denoted as 4p, or processes), a single SSD is saturated with the peak performance of 774 KIOPS. In Figure 1c, we continue to scale the number of CPU cores with all 8 SSDs (the max possible). The figure shows that the peak performance shifts from 774 KIOPS (from Figure 1b) to 3.4 MIOPS for 10 CPU cores (and 16 as well). In this configuration, all 10 CPU cores are 100% utilized and this is the peak performance the Linux kernel can deliver with the `io_uring` engine (bounded by the CPU performance). We also plot the SPDK performance, which is a high-performance, kernel-bypass storage stack [10]. With SPDK, it takes 6 CPU cores ("6p SPDK" line) to deliver 5.2 MIOPS, saturating the SSDs (within 85% of the peak possible 6.2 MIOPS (8×774 KIOPS) that our hardware should deliver). *This demonstrates that SPDK (1.3 MIOPS/core) is still the state-of-the-art storage stack, which is at least 3.6× more efficient (single core peak IOPS performance with 8 SSDs, "*`1p`*" and "*`1p SPDK`*" lines in Figure 1c) than the Linux storage stack (370 KIOPS/core)* (**O-2**). SPDK can deliver higher throughput than the Linux storage stack when the CPU is the bottleneck because SPDK is more CPU efficient than the Linux storage stack. Previous study shows that SPDK needs fewer CPU instructions to process each I/O request than the Linux storage stack [51].

3.3 Summary

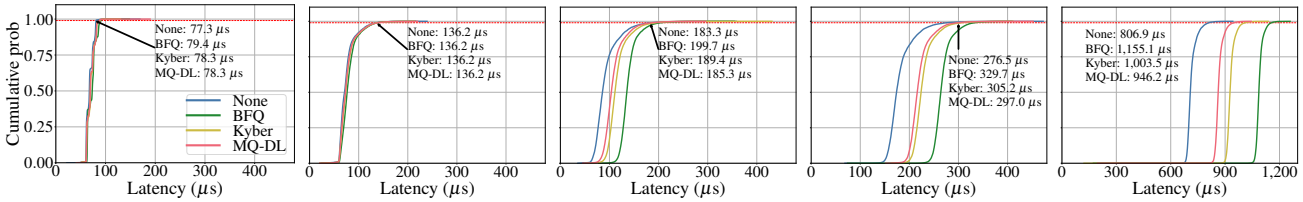
What is the key resource bottleneck for the L-app, T-4KiB-app, and T-64KiB-app? The key finding (**Key Finding, KF1**) here is that as the NVMe device speeds are improving, the CPU becomes the primary bottleneck. Modern fast NVMe storage devices like Samsung 980 PRO 1TiB, can require more than a single CPU core to saturate the performance of the SSD. On our setup, L-apps (QD=1), and T-4KiB-apps (QD=128) are bounded by the CPU performance. We report that the T-4KiB-app needs at least 4 T-4KiB-apps on 4 CPU cores to saturate a single SSD. The T-64KiB-app can saturate a single SSD with only one CPU core (not shown). *Hence, the age-old mantra of "CPU is fast, I/O devices are slow" does not hold anymore for modern fast NVMe SSDs* (**O-3**). We answer **RQ1** by identifying when the CPU or the SSD becomes a bottleneck (the inflection or saturation points). With the Linux I/O stack, the CPU can only deliver 51.6% of the peak hardware throughput (3.2 MIOPS out of a possible 6.2 MIOPS) before it becomes the bottleneck. When we introduce an I/O scheduler to deliver a quality-of-service (QoS) in this setting, it also competes for the CPU cycles. The operational complexity of the scheduler determines the raw performance loss that is traded to deliver a quality-of-service (QoS). In the following section, we study the impact of I/O schedulers (§4), and quantify the performance loss and QoS in the presence of competing tenants with interference (§5).

4 I/O SCHEDULER SCALABILITY

In the previous section, we show the baseline performance and overhead between a CPU and NVMe devices without I/O schedulers with multiple concurrent requests and processes. We report that a single process gets 370 KIOPS at the queue depth (QD) of 64, and the Linux storage stack can not fully saturate 8 SSDs even with 16 T-4KiB-apps (100% CPU utilization). In this section, we introduce I/O schedulers and answer the **RQ2** about how the performance and overheads scale with increasing amounts of concurrency with NVMe devices with I/O schedulers. We evaluate the scalability by analyzing workload throughput (IOPS) and tail latency (P99) with a varying number of hardware resources. Specifically, we measure the scalability of both latency-sensitive and throughput-bound workloads across 2 resource axes: number of SSDs and CPU cores. We look at both intra- and inter-process scalability. Intra-process scalability refers to configurations where we increase the concurrency within a single process by increasing the number of concurrently issued outstanding I/O requests. With inter-process scalability, we increase the concurrency by increasing the number of parallel processes while keeping the concurrency within each process fixed. The expectation here is that multiple processes exercise the scheduling, locking, and synchronization overheads within the scheduler.

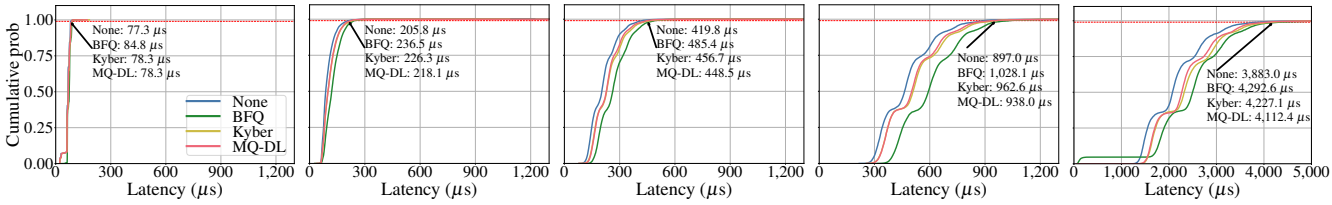
4.1 Scheduler Overheads on Latency

What are the scheduler overheads, and how do they scale with increasing I/O concurrency? In this section, we study the impact of I/O concurrency overheads from schedulers on latency-sensitive applications (L-app). For this, we study the intra- (Figure 2) and inter-process (Figure 3) concurrency overheads. For **intra-process concurrency**, we have a single process (pinned to a core) that



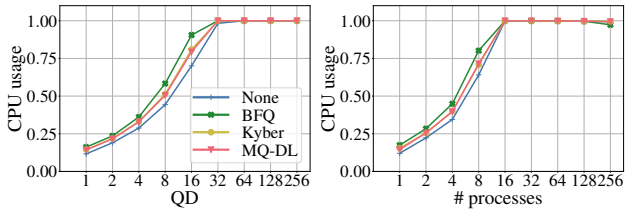
(a) QD = 1 (b) QD = 16 (c) QD = 32 (d) QD = 64 (e) QD = 256

Figure 2: Intra-process scalability latency CDFs with increasing queue depth (QD); Note the different x-axis scale for (e).



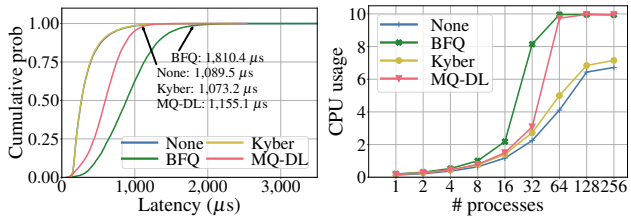
(a) 1 L-app (b) 16 L-apps (c) 32 L-apps (d) 64 L-apps (e) 256 L-apps

Figure 3: Inter-process scalability latency CDFs with increasing number of L-apps; Note the different x-axis scale for (e).



(a) Intra-process (b) Inter-process

Figure 4: CPU usage for intra/inter-process concurrency.



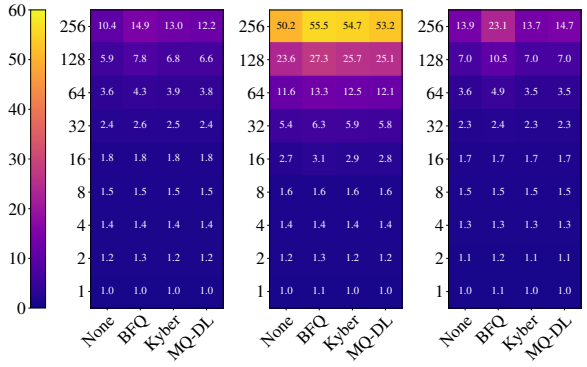
(a) Latency CDF for 256 L-apps (b) CPU usage

Figure 5: L-app inter-process scalability (10 cores, 1 SSD); Note in (b) the y-axis is the CPU usage, which is from [0–10], representing 10 CPU cores.

issues 1, 16, 32, 64, or 256 outstanding requests at a time. For this process, the latency CDF is shown in Figure 2 (annotations are for P99 latency). There are two key observations here. Firstly, for the concurrency of 1 and 16, the latency profile shape of different I/O schedulers looks quite similar. This is due to the fact that as we increase concurrency for a single process, the CPU load increases but still remains under 100%, hence having spare capacity (Figure 4a). We still report that P99 latencies increase up to 73.9% for *all* schedulers (from 78.3 to 136.2 μs). *As long as the CPU is not 100% utilized, all three schedulers have comparable performance* (**O-4**). Secondly, when the CPU load is 100% (at QD=32), the scheduler performance starts to diverge significantly (Figure 2 (c–e)). *Hence, there is a clear number of outstanding requests where BFQ introduces the maximum overheads (43.2% over the None scheduler for QD=256)* (**O-5**). The overall P99 latencies deteriorate from 77.3–79.4 μs (QD=1) to 806.9–1,155.1 μs (QD=256), due to the CPU overheads (as we show previously that a single app can not saturate a single SSD).

In the case of **inter-process concurrency** (multiple concurrent L-apps pinned on a single CPU core with each having QD=1), the latencies deteriorate faster than the intra-process configurations as shown in Figure 3. With the inter-process setup, the Linux kernel has to deal with multiple concurrent processes and associated abstractions and overheads (scheduling, context switching, virtual memory). In a single process case, many of these overheads could be amortized or eliminated. Hence, we also report that the single CPU core where all of these concurrent processes are pinned is saturated with QD=16 via 16 processes as shown in Figure 4b. In comparison, the intra-process saturation point is at QD=32. *Overall as the concurrency increases, the inter-process P99 latencies are approximately 4× higher than their intra-process latencies* (**O-6**). For example, at QD=256 concurrency, the intra-process and inter-process P99 latencies with Kyber are 1,003.5 μs and 4,227.1 μs, respectively.

We further experiment with the **inter-process setup with all 10 cores**, thus no pinning and restricting the performance to a single core. We show the results in Figure 5. *An interesting observation is that with more CPU cycles being available (single core to 10 cores), all the schedulers improve their P99 latencies and bring it closer to their single CPU core, intra-processes performances (Figure 2e), due to the reduction of process scheduling and context switches* (**O-7**). Kyber and None improve much better than BFQ and MQ-Deadline. Their performances are closer to each other with overlapping lines in Figure 5a. BFQ has 1.57× worse P99 latency than its intra-process counterpart. Both BFQ and MQ-Deadline have more slanted shapes, thus having higher P50 and P75 latencies than None and Kyber. This behavior can be explained by their CPU utilization as shown



(a) 1 core, 1 process y-axis: QD **(b) 1 core, QD 1 y-axis: # processes** **(c) 10 cores, QD 1 y-axis: # processes**

Figure 6: Normalized heatmap for P99 tail latency differences (scale from 0–60×) for various concurrency levels (y-axis, 1–256) achieved with (a) QD increase with a single process (intra-process) with 1 CPU core; (b) increasing the number of processes with 1 QD on 1 CPU core (inter-process); and (c) increasing the number of processes with 1 QD on 10 CPU cores (inter-process). The baseline is the None scheduler with 1 concurrency (bottom left box with 1.0 value).

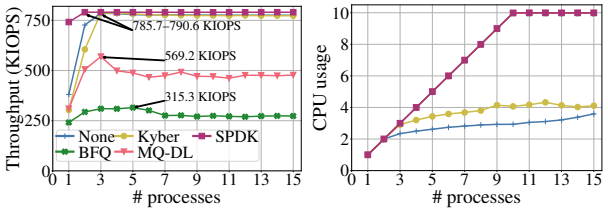
in Figure 5b where BFQ and MQ-Deadline have significantly higher CPU utilization than Kyber and None. With 64 processes, both BFQ and MQ-Deadline are CPU-bounded and thus experience quick degradation of latencies. For 256 processes (for which Figure 5a shows the latencies), None and Kyber use 7 out of 10 CPU cores whereas BFQ and MQ-Deadline use all 10. *These observations summarize that among the three schedulers under study, Kyber is the most light-weight, while BFQ is the most CPU intensive and complex* (**O-8**).

In order to visualize the difference between intra- and inter-process latencies overheads we also plot a normalized heatmap in Figure 6. Here we illustrate P99 latencies degradation (normalized to the 1 concurrency with None scheduler) as we increase the level of concurrency in the system (the y-axis, from 1–256) in three configurations: intra-process (1 CPU core), inter-process (1 CPU core), inter-process (10 CPU cores). The heatmap shows that up to a concurrency of 8, the P99 latencies increase slowly over the None scheduler, and intra-process counterparts (the baseline, showing the normalized value of 1.0). *Between 8–16 concurrency, the CPU becomes the bottleneck, and at this point, the deterioration starts very quickly reaching higher than 50× for inter-process overheads on a single CPU for all schedulers* (**O-9**).

4.2 Scheduler Overheads on Throughput

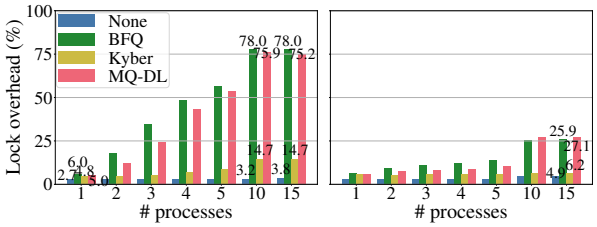
We now bring our attention to the throughput-bounded T-apps where we measure peak throughput (IOPS) and show what scalability and overheads are observed by such workloads with I/O schedulers. In this section, we study four specific questions regarding the scalability properties of the CPU cores and SSDs:

Firstly, what are the scheduler overheads for the throughput-bounded applications, and how do they scale with the number of CPU cores? We start with reporting the scheduler performance



(a) Total throughput **(b) Total CPU usage**

Figure 7: T-app inter-process scalability (10 cores, 1 SSD) with an increasing number of T-4KiB-apps.

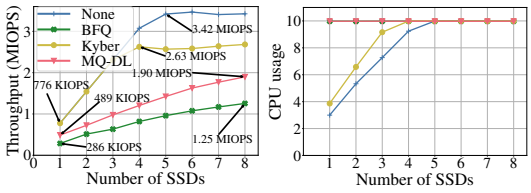


(a) 1 SSD **(b) 8 SSDs**

Figure 8: Combined lock overhead of T-4KiB-apps with an increasing number of concurrent processes.

(in IOPS) as we increase the number of T-4KiB apps. Recall from §3 that a single T-4KiB-app can not saturate a single SSD, and at least 3–4 T-4KiB-apps are needed to saturate a single SSD with the None scheduler. Figure 7 shows scalability results in throughput (IOPS, y-axis, higher is better) for multiple T-4KiB-apps (x-axis) on a single SSD. There are two key observations here. Firstly, the Kyber scheduler performs very closely to None, where both reach the peak single SSD performance of 785.7 KIOPS with 3 processes. For reference, we also have a line for SPDK that can deliver 790.6 KIOPS with a single CPU core, thus demonstrating a wide gap between the in-kernel and kernel-bypass (SPDK) storage stacks. Secondly, both BFQ and MQ-Deadline can not reach the peak device throughput with any number of concurrent T-4KiB-apps. BFQ and MQ-Deadline deliver a peak performance of 315.3 KIOPS (0.40× of the peak 785.7 KIOPS) and 569.2 KIOPS (0.72× of the peak 785.7 KIOPS), respectively. *This loss represents a significant performance degradation, and we conclude that BFQ and MQ-Deadline are unsuitable to be used with modern NVMe SSDs* (**O-10**). The reason for this performance degradation is related to each scheduler's complexity, CPU utilization, and scalability bottlenecks, specifically lock contention.

Figure 7b shows the CPU utilization where we report that despite having CPU available (not all 10 CPU cores are 100% utilized until 10 processes), both BFQ and MQ-Deadline suffer from significant lock contention, thus limiting their performance scalability. To further analyze their CPU utilization behavior, we break down the CPU utilization of these I/O schedulers. We count the total number of CPU cycles (`cpu-cycles` counter with the Linux `perf` framework) and classify lock-related CPU cycles by attributing them to specific lock-related functions: `native_queued_spin_lock_slowpath`, `_raw_spin_lock`, `_raw_spin_lock_irq`, `_raw_spin_lock_irqsave`, and `mutex_lock`. We then plot the fractions of CPU cycles spent in such lock-related functions in Figure 8.



(a) Total throughput **(b) Total CPU usage**

Figure 9: T-app inter-process scalability (10 cores, 10 concurrent T-4KiB-apps) with an increasing number of SSDs.

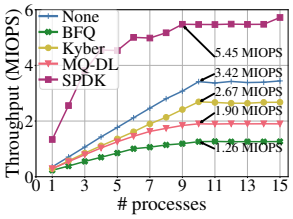


Figure 10: T-app inter-process scalability (10 cores, 8 SSDs) with an increasing number of T-4KiB-apps (x-axis).

From these graphs, we observe that even for one SSD, BFQ and MQ-Deadline spend a significant fraction of their CPU cycles on locking/synchronization functions (as high as 78.0% for 10–15 processes). Increasing the number of available SSDs from 1 to 8 (Figure 8b) reduces the overall CPU cycles spent in lock-related functions for BFQ and MQ-Deadline significantly, by up to 67.1%, suggesting device-specific nature of these locking overheads. Yet even in this case, almost a quarter of CPU cycles (up to 27.1%) are spent on the locking-related functions. In comparison, Kyber only spends 14.7% (single SSD) to 6.2% (8 SSDs) of CPU cycles on lock-related functions. *Based on this analysis, we conclude that locking is the cause of bottleneck for scalability of BFQ and MQ-Deadline and must be urgently tackled* (**O-11**). As of January 2024, Linux kernel developers have also identified this locking issue and are improving the scalability of the BFQ and MQ-Deadline schedulers [35, 36].

Secondly, how does the performance scale in the presence of lock-related overheads with the number of NVMe SSDs? Figure 9 shows our results as we study the performance scalability properties of the schedulers (as IOPS, on the y-axis, higher is better) with 10 T-4KiB-apps (fixed) when we increase the number of SSDs from 1 to 8 (x-axis). In this experiment, the relative overheads and ranking of I/O schedulers remain the same, where Kyber performs the best, followed by MQ-Deadline and lastly BFQ. The performance of BFQ and MQ-Deadline improves sub-linearly from 286 KIOPS and 489 KIOPS with a single SSD to 1.25 MIOPS and 1.90 MIOPS for 8 SSDs, respectively. An interesting observation here is that even with a single SSD, the CPU load for BFQ and MQ-Deadline is 100% for all 10 CPU cores, yet the performance improves as the number of SSDs increases. We speculate (not verified) that this is due to the presence of device-specific locking that exists for a single device, but the presence of multiple devices offers more opportunities for parallelism without being restricted by a single device lock. In the case of Kyber and None, they reach the CPU saturation points with 4 and 5 SSDs, respectively, thus showing IOPS scaling up to those points, delivering a peak performance of 2.63 MIOPS (Kyber) and 3.42 MIOPS (None).

Thirdly, what is the peak performance that various I/O schedulers can provide in Linux? We run the full hardware configuration with 10 CPU cores, 8 SSDs, and vary the number of processes from 1 to 15, and measure the peak IOPS performance of different I/O schedulers. Figure 10 shows our results. There are three specific observations from this experiment. Firstly, there is a clear order in terms of performance among the schedulers (in ascending order): BFQ (1.26 MIOPS), MQ-Deadline (1.90 MIOPS), Kyber (2.67 MIOPS) and None (3.42 MIOPS). Secondly, all of these schedulers reach their peak throughput with 10 processes where all CPU cores become 100% utilized. Lastly, for comparison, we also show the performance of SPDK that can deliver 5.45 MIOPS with 9 CPU cores, demonstrating an almost 2× performance gap to the next best I/O scheduler, Kyber. *From this experiment, we conclude that the CPU and associated heavy Linux software stack and I/O schedulers have become the performance and scalability bottleneck with modern NVMe devices* (**O-12**). Though this final takeaway message is not a surprising find, our key contribution is quantifying the gap between schedulers via controlled experiments where we characterize the performance of the I/O schedulers under various conditions (CPU or SSD becoming the bottleneck). For example, between BFQ and Kyber, there is a performance difference of 2.1×.

Lastly, How do I/O schedulers scale with throughput-bound applications on a single SSD with large request sizes (64KiB)? Unlike the prior evaluation, where the CPU becomes the bottleneck with the T-4KiB-apps, we repeat the same experiments for T-64KiB-apps and report that in this scenario all configurations are SSDs bounded. In these experiments, the performance of all I/O schedulers is almost identical (not shown in any graph). *All the I/O schedulers are able to saturate the SSD with only 2 processes for bandwidth-driven workloads* (**O-13**). The None scheduler reaches a peak throughput of 232 KIOPS with only a single T-64KiB-app process. As the SSD is the bottleneck for the T-64KiB-apps, the CPU utilization for all I/O schedulers remains low, (less than 2 CPU cores). BFQ has a slightly higher CPU utilization of 1.3 CPU cores, 8.3% higher than None (1.2 CPU cores).

Answering RQ1 and RQ2 with KF-2: Based on the analysis in this section, we summarize when the CPU is not the bottleneck, all schedulers perform similarly. As the CPU progressively becomes a bottleneck (i.e., 100% utilization), the Linux I/O schedulers can introduce up to 63.4% performance overheads for the throughput, and more than 50× degradation for P99 latencies over the None scheduler. We also report that the presence of multiple SSDs improves the performance scalability of I/O schedulers due to eschewing the device-specific locking overheads. Based on these findings we recommend using Kyber scheduler (lowest CPU overheads) for NVMe SSDs with modern multicore CPU machines.

5 I/O INTERFERENCE WITH CONCURRENT WORKLOADS

In this section we investigate RQ3: "*how can I/O schedulers help to control interference in the presence of competing workloads, specifically latency-vs-bandwidth, and read-vs-write*". Quality-of-service (QoS) is an essential component of I/O schedulers in multi-tenant

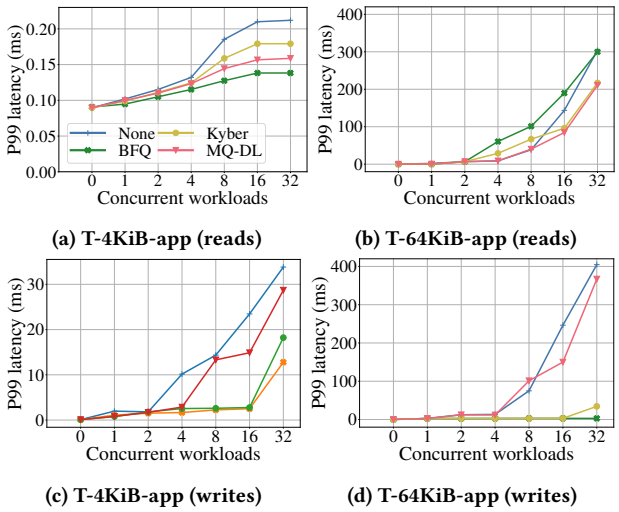


Figure 11: L-app tail latency with an increasing number of interfering background applications; *Note: scales differ on the y-axis and they are in Milliseconds!*

environments (e.g., defined as SLAs), especially for latency-critical workloads. However, there is limited literature available on the QoS of *state-of-the-practice* I/O schedulers available in Linux with modern high-speed NVMe SSDs. In this section, we evaluate the QoS of applications in multi-tenant environments. We define the QoS as the P99 tail latency for latency-sensitive applications and the IOPS for throughput-bound applications. The unique aspect of our study is that we study the mixed settings (mixing latency with throughput) as well as read-write interference, the latter of which is unique to flash-based SSDs [20, 26, 33, 40, 46, 52, 56, 57].

5.1 Latency Interference from Concurrent Read-Write T-4KiB-apps

For latency-sensitive applications (L-apps), it is essential that tail P99 latency is low and bounded even in the presence of concurrent workloads. Below, we determine if Linux I/O schedulers are able to meet such demands by controlling the interference among workloads. We devise an experiment in which one foreground L-app issues I/O requests concurrently with multiple interfering throughput-heavy T-4KiB-apps and T-64KiB-apps workloads in the background. Considering that SSD writes and reads typically have a different performance within flash that can lead to read-write interference, our background workloads issue both (random) read and write I/O requests to the SSD.

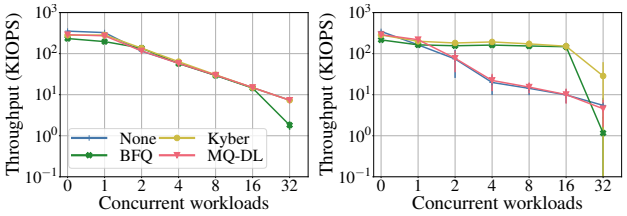
What kind of bounded P99 latencies do the I/O schedulers offer when a foreground L-app runs with background read and write T-apps? For this experiment, we measure the P99 latency of a single L-app while we increase the number of background read and write applications. Figure 11 shows our results of P99 latencies (y-axis, lower is better) with four different configurations with an increasing number of background workloads (on the x-axis): (a) multiple random read T-4KiB-apps (CPU bounded); (b) multiple random read T-64KiB-apps (SSD bounded); (c) multiple random write T-4KiB-apps (CPU bounded); and (d) multiple random write T-64KiB-apps (SSD bounded). From Figure 11 we observe that BFQ has the lowest P99 read latencies under the majority of the evaluated scenarios (except for random read T-64KiB-apps), with up to 34.8% lower latency than None (212.0 μs vs. 138.2 μs). *The lower latency for BFQ holds for small (4KiB) reads and small (4KiB) and large (64KiB) write requests* (**O-14**). We have two assumptions about why BFQ achieves low latency for the L-app: (1) the BFQ-specific option `low_latency` [1] automatically detects and provides low latency for latency-sensitive applications, which is enabled by default; and (2) BFQ gives exclusive access of the SSD to a workload (no concurrent request dispatching from multiple workloads), which reduces the interference from other workloads. From Figure 11 (c) and (d) we also report that Kyber has low latencies with concurrent small (4KiB) and large (64KiB) write requests (similar performance to BFQ for writes). However, the latencies are only slightly better than using the None scheduler for concurrent reads (Figure 11b). The reason for Kyber's low P99 latency in the presence of concurrent writes (Figure 11 (c-d)) is that Kyber prioritizes read requests over writes [8] under all circumstances. Considering that the L-app solely issues reads, this means that the foreground L-app is prioritized among the concurrently writing background workloads. *Hence the L-app with reads is prioritized among the concurrent writers, but is treated similarly and reaps no benefit with concurrent readers* (**O-15**).

We further identify that the None scheduler is not capable of bounding or guaranteeing any latency performance, and performs the worst in multiple scenarios. Specifically with read-write interferences, as shown in Figure 11 (c-d), the P99 latency deteriorates very quickly. With 1 background workload, the performance of the None scheduler is the same as other schedulers. However, as the number of background workloads increases the gap widens to as large as 139× (404.8 ms for None vs. 2.9 ms for BFQ in Figure 11d).

In Figure 11c and Figure 11d, the read latency of MQ-Deadline increases significantly (up to 32.4× with T-4KiB-apps and 4,142.2× with T-64KiB-apps) as the number of concurrent write applications increases. The performance trend is very similar to the performance trend when None scheduler is used. The reason is that MQ-Deadline does not have different priorities between reads and writes, and does not prioritize one application over the other. Therefore, the effects of read/write interference are not controlled, as requests are dispatched summarily to the SSD for both operations as they arrive in the scheduler. As the SSD becomes the bottleneck with 3-4 concurrent workloads, the I/O latencies reflect the read-write latencies of the SSD performance with interference. To summarize (**KF-3**), (i) BFQ offers better control over interference (lower P99 latencies) for latency-sensitive workloads (while trading performance, §4) than other schedulers; (ii) Kyber and BFQ excel in managing read-write interferences among concurrent workloads. However, Kyber has poor performance with read-read interference.

5.2 Throughput Interference from Read-Write Workloads

For throughput-bound applications (T-apps), throughput should be as high as possible and shared evenly among concurrent applications. Therefore, we evaluate if a foreground T-app can maintain a high throughput in the presence of concurrently running background workloads. We run a single foreground T-4KiB-app with an



(a) T-64KiB-apps (reads) (b) T-64KiB-apps (writes)

Figure 12: Read throughput (IOPS) of a T-4KiB-app workload with an increasing number of interfering background T-64KiB-app workload. Note: The y-axis is log-scale.

increasing number of concurrent T-64KiB-apps in the background; hence, introducing workload competition for throughput resources. Similar to §5.1, we evaluate the interference from both read and write background workloads. We plot the throughput of the foreground T-4KiB-app in IOPS (y-axis, higher is better) in Figure 12, with the number of concurrent background T-64KiB-apps increases.

Can I/O schedulers evenly share the IOPS performance among concurrent throughput bounded (4KiB and 64KiB) workloads? With the even sharing the expectation is that an IOPS-driven workload (T-4KiB-app) should observe linearly bandwidth deterioration as throughput-bounded workloads are run in the background. Figure 12a demonstrates that none of the schedulers are capable of offering a bounded and linear IOPS performance sharing with read-read interference, and the performance of the foreground workload drops very quickly (the y-axis is logarithmic). *An investigation of the total throughput (sum of the T-app and background operations) shows that the IOPS is divided equally among the applications* (**O-16**). However, a point to be noted here is that even though 4KiB and 64KiB applications both receive equal IOPS, the 64KiB applications get 16× more bandwidth than the 4KiB application due to their larger request size. Hence, all schedulers fail to provide equal bandwidth sharing to ensure a *proportional IOPS sharing* among the workloads.

IOPS sharing in the presence of read-write interference is managed better. *In Figure 12b we report that Kyber and BFQ offer the highest throughput for read operations for the foreground workload* (**O-17**). We have two key findings. Firstly, None and MQ-Deadline both lead to read IOPS degradation for the foreground workload in the presence of read-write interference from the background writer workloads. A reason for this is that background writes are faster than the reads (lower latencies), hence, they can occupy the device's internal bandwidth the majority of the time. None and MQ-Deadline do not have any mechanisms to throttle writes to help reads. In contrast, Kyber differentiates the two, and prioritizes the reads, hence, maintaining a flat-line performance for the foreground T-4KiB-app. The reasons that BFQ leads to a higher throughput are twofold: (1) the fair sharing where BFQ equally divides the available bandwidth between applications; (2) BFQ prioritizes interactive applications (i.e. the foreground app could have been identified as interactive). *Beyond 16 concurrent applications in Figure 12b, a bottleneck can be identified for BFQ* (**O-18**). The throughput of BFQ decreases by 99.2% between 16 and 32 concurrent applications and averages at 1.2 KIOPS, down from 147.7 KIOPS with 16 background apps. This leads to a significantly lower throughput than the throughput of the other schedulers (from 73.9% up to 95.8%). This is a consequence of its locking overheads as lock contention increases significantly after 16 processes.

Hence, our findings recommend to (**KF-4**) (i) use Kyber or BFQ to control read/write interference; (ii) be aware of concurrency limits to the schedulers as the CPU becomes 100% occupied, a configuration that can lead to significant performance losses.

6 KYBER CONFIGURATION EXPLORATION

In the previous sections, we observed that both BFQ and MQ-Deadline are unable to saturate a single fast SSD because of high CPU overhead and scalability issues, which makes them unsuitable for such SSDs. Kyber, on the other hand, is specifically designed for fast multi-queue devices and has better scalability than BFQ and MQ-Deadline. Henceforth, we focus the rest of our studies on Kyber. In particular, we look at configuring Kyber since Linux provides various configuration parameters for this scheduler. We evaluate how Kyber's parameter configuration affects its performance characteristics, and we give guidelines on configuring Kyber in practice, thus answering *how do a scheduler configuration parameters affect the schedulers' behavior on competing workloads?* (**RQ4**).

Kyber has two configurable parameters [8], `read_lat_nsec` (default = 2 ms) and `write_lat_nsec` (default = 10 ms) — we will refer to them as *R lat* and *W lat*. These parameters control the desired or target request latencies. In order to prioritize a read or a write type of I/O request, Kyber uses tokens for read and write requests. The number of tokens helps Kyber to control the maximum number of I/O requests of a particular type in flight, bounding the size of the request queue to SSDs, and as a result, the maximum latency for the I/O request. The number of tokens can not be configured with Kyber's parameters directly. Kyber controls the number of tokens by closely monitoring the current read and write completion latencies and comparing them against the target latencies, i.e., *R lat* and *W lat*. The number of tokens remains the same if both achieved read and write latencies are lower than the target latencies. If the completion latency exceeds the target latency, Kyber increases the number of tokens. Hence, setting a lower target latency for reads or writes implicitly prioritizes it (by not letting the requests build up in the queue, thus dispatching it immediately).

The number of tokens for a particular type of request (read or write) is reduced when (1) the achieved *P90 latency* for that request type is lower than the target latency (i.e., this type of request is well served); and (2) the achieved *P99 latency* for the other type is higher than the target latency (i.e., the other request type is badly served). In this case, Kyber can re-prioritize the badly served request by reducing the number of tokens for the other type. The minimum number of tokens for both read and write is 1, and the maximum number of tokens is 256 for read and 128 for write. We define the number of read tokens as *R Tokens* and write tokens as *W Tokens*. Below, we investigate how setting Kyber's target latencies affects Kyber's throughput and tail latency.

Table 2: Kyber configuration impact on the throughput of a concurrent read and write application (R and W KIOPS). Highlighted entries are discussed in §6.1.

ID	*R lat*	*W lat*	*R, in KIOPS*	*W, in KIOPS*	*R Tokens*	*W Tokens*
0	2 ms	10 ms	156.7±5.3	103.4±5.1	256±0	93±7
1	0 s	10 ms	189.4±11.5	79.4±2.2	256±0	54±15
2	2 ms	0 s	137.8±0.6	114.8±4.2	253±2	128±0
3	0 s	0 s	136.1±1.0	118.5±5.0	256±0	128±0
4	1 s	1 s	137.2±1.8	113.6±5.7	256±0	128±0
5	0 s	1 s	219.9±11.0	70.0±0.3	256±0	1±0
6	1 s	0 s	2.1±0.0	118.8±5.4	1±0	128±0

Table 3: Kyber configuration impact on read P99 latency in milliseconds and W tokens (presented in "()") of an L-app running with an interfering T-4KiB-app (random write). The highlighted entries are discussed in §6.2.

		W lat				
		0 s	100 μs	10 ms	20 ms	100 ms
R lat	0 s	2.8 (128)	2.7 (119)	1.4 (10)	1.4 (1)	1.6 (1)
	50 μs	2.8 (128)	2.7 (119)	1.5 (17)	1.7 (1)	1.6 (1)
	100 μs	2.8 (128)	2.8 (116)	1.7 (39)	1.6 (1)	1.5 (1)
	500 μs	2.8 (128)	2.7 (118)	1.9 (50)	1.5 (1)	1.5 (1)
	10 ms	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)
	100 ms	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)	2.8 (128)

6.1 Mixing throughput-bound mixed read-write workloads

Kyber limits the maximum number of concurrent requests on the device side through the number of tokens. Limiting the number of requests decreases the throughput, which prevents throughput saturation and leads to SSD idling. Thus, the hypothesis is that changing the target latency directly affects the throughput. To test this hypothesis, we run a foreground T-4KiB-app with random reads and a background T-4KiB-app with random writes concurrently. We issue I/O requests to one SSD and pin each process to a different CPU core to avoid interference with the process scheduler. Table 2 shows the read and write throughput (and accompanying number of tokens) under various Kyber configurations (identified with unique *ID*s). The default configuration is *ID* 0 (*R lat*: 2 ms, *W lat*: 10 ms).

If we change the default *R lat* to the minimal value (0), the read throughput increases significantly (20.9%) at the cost of write throughput (a 23.2% decrease), a change that is reflected as a significant decrease in the number of write tokens (93 to 54). Similarly, when *W lat* is changed to the minimal value (0), the write throughput increases significantly (11.0%) at the cost of read throughput (a 12.1% decrease), a change that is reflected as a slight decrease in the number of read tokens (256 to 253). *In short, setting the target latency to the minimum for a particular type of request (read or write) leads to a significant increase in the throughput (up to 20.9%) at the cost of the other type's throughput degradation (up to 23.2%)* (**O-19**).

In *ID* 3–4, we set *R lat* and *W lat* both to the same unrealistic extreme values, the minimum (0 s, highest priority) and an arbitrarily high value (1 s, least priority). This leads to the maximum number of tokens for both reads and writes (256 for read and 128 for write). Lastly, we set *R lat* or *W lat* to 0 s while the other is set to 1 s (*ID* 5, 6). With this configuration, we try to get the lowest latency possible for either reads or writes. *This configuration leads to a significant throughput increase for the prioritized target request type (40.3% for reads, 14.9% for writes), but leads to a significant throughput decrease for the other type also (98.7% for reads, 32.3% for writes)* (**O-20**).

6.2 Latency-sensitive Read Workload with a Write-driven Throughput Workload

To investigate how configuring Kyber affects the latency of L-apps with background write T-4KiB-apps, we run an L-app (read) with a background T-4KiB-app (write) on a single SSD, with each application pinned to a separate CPU core. In Table 3 we show the P99 tail latency in milliseconds of L-apps with the number of *W Tokens* shown in parentheses. The chosen *W lat* is represented with the columns and the *R lat* with the rows.

We first find a general trend between the achieved read latency and the number of W Tokens: the higher the number of W Tokens, the worse the achieved read latency (**O-21**). When the *R lat* is set to an extremely high value that is unlikely to be reached by the hardware (in our case higher than 10 ms) or the *W lat* is set to the minimum value (0), the number of *W Tokens* is set to the largest value, i.e., 128, meaning that reads are not prioritized, leading to high read latency (2.8 ms). These entries are shown with gray backgrounds. When *R lat* is set to a low value (less than or equal to 500 μs), and the gap between *R lat* and *W lat* increases, the number of *W Tokens* decreases (from 128 to 1), meaning that reads are more prioritized, leading to decreasing read latency (2.8 ms to 1.4 ms, 50% lower). These entries are shown with green backgrounds. *The larger the gap between R lat and W lat (R lat is lower than W lat), the more reads are prioritized, thus improving the achieved read latency (up to 50% lower)* (**O-22**).

6.3 Write-driven Throughput Workload with a Read Workload

To determine the effect of Kyber's parameters on throughput-bound applications in mixed workloads, we use the same experiment as used for evaluating the effect on latency-sensitive applications, but this time reporting the write latency for the T-4KiB-app. In Table 4, we show the result with throughput in KIOPS (higher is better) of the foreground T-4KiB-app writing workload. The number of write tokens is already shown in Table 3. Since reads only have QD=1 (L-app), the number of read tokens has no influence on the read performance, and we do not show the number of *R Tokens*. The configured *W lat* is represented by the columns and the configured *R lat* is depicted in the table rows. We observe that the lower the number of *W Tokens*, the lower the write throughput. *Configuring W lat to a high value (higher than 10 ms for our SSD), while setting R lat to a low value (less than 500 μs) has a negative impact on the write throughput (up to 43.5% lower write throughput)* (**O-23**). These entries are shown with gray backgrounds. If we combine the result with the previous table, we observe that if Kyber provides lower read latency, it comes at the cost of write throughput. Thus, when configuring Kyber for a workload, either read latency or write throughput can be prioritized, but not both (**KF-5**).

Table 4: Kyber configuration impact on write throughput in KIOPS of a T-4KiB-app running with an interfering L-app (random read). The highlighted entries are discussed in §6.3.

R lat \ W lat	0 s	100 μs	10 ms	20 ms	100 ms
0 s	129.4	103.4	77.3	73.6	73.1
50 μs	116.5	111.6	80.3	71.2	73.3
100 μs	128.8	128.0	82.4	73.2	73.3
500 μs	122.2	115.6	96.0	73.8	73.5
10 ms	119.6	130.9	127.5	125.4	131.8
100 ms	135.0	123.7	119.7	129.7	118.0

7 RELATED WORK

On *State-of-the-Art* I/O Schedulers. In this work, we investigate Linux's *state-of-the-practice* I/O schedulers. However, there is also a vast literature on *state-of-the-art* schedulers, which are not included with the default Linux kernel and require domain expertise. Therefore, we consider such work orthogonal. Modern SSDs use multiple channels internally to deliver high throughput and have complex internal mechanisms such as FTL, buffers, and garbage collection (GC). These idiosyncrasies require more novel designs than the *state-of-the-practice* Linux I/O schedulers. Here, we discuss *state-of-the-art* schedulers for fair-sharing, low-latency applications, and schedulers that use flash-specific functionalities.

There are many fair-sharing I/O schedulers [17, 21, 26, 46, 52, 53, 56, 57] that can be used as alternatives for the fair-sharing BFQ scheduler. Modern SSDs support multiple hardware dispatch queues and it is necessary to use multiple queues to fully saturate the SSD's performance and to provide fair sharing [56]. Additionally, NVMe features like weight round robin (WRR) allow assigning weights to applications, which can also be used to guarantee fairness across applications. This leads to designs that use multiple queues such as MQFQ [26] and multiple queues with WRR such as D2FQ [56]. We do not measure fair-sharing in this work since most Linux schedulers do not support it.

There are also a number of *state-of-the-art* schedulers optimized for latency-sensitive applications like the L-app used in this paper [27, 39, 42]. Such schedulers can be used instead of the Linux' Kyber and BFQ schedulers, which both have options to prioritize low-latency applications. This includes solutions such as blk-switch [27], which provide low latency while preserving high total throughput, and FastResponse [39] that co-designs the I/O stack and scheduler to reduce cross-layer I/O interference.

Flash-based SSDs have many idiosyncrasies that can be exploited by schedulers [22, 23, 29, 31, 52, 54]. For example, increasing SSD longevity by reducing wear-levelling [54] or reducing GC overhead [22, 23, 29]. Generally, flash-aware schedulers also treat writes and reads differently because of read/write interference [31, 46].

On Performance Characterizations. Parallel to this work, the performance of the Linux storage stack is characterized in many papers. Whitaker et al. [55] characterize the performance of Linux' I/O schedulers for ULL non-flash-based SSDs. Their findings confirm that schedulers inevitably lead to higher latency and lower throughput. Additionally, this work looks at energy efficiency, where they find that schedulers have high energy overhead, especially for BFQ. We extend this work by looking at QoS and looking at the more common flash storage (ULL SSDs are not widely deployed yet). The Linux user community has investigated I/O schedulers for SSDs as well [34], and their work showcases that using no scheduler, followed by Kyber, leads to the highest throughput and low latency for applications like MySQL and RocksDB. There are also various works that do performance characterization of emerging storage APIs such as `io_uring` [19, 24, 51]. Our work differentiates itself in terms of scale, comprehensiveness, and its sole focus on the performance characterization of Linux storage I/O schedulers.

8 CONCLUSION

In this paper, we investigate if the Linux I/O schedulers fit modern NVMe SSDs. Our results show that BFQ and MQ-Deadline have significantly high CPU overhead and scalability issues caused by locking. Thus, we suggest that BFQ and MQ-Deadline should not be used with these SSDs. Kyber has lower CPU overhead than BFQ and MQ-Deadline with near-linear scalability and thus is the best fit of these SSDs. However, the parameters of Kyber need to be tuned carefully or Kyber harms the performance. Our analysis focuses on the Linux I/O schedulers. This work can be expanded in (1) evaluating how the start-of-the-art I/O schedulers perform on the flash-based NVMe SSDs, (2) comparing how different I/O scheduling algorithms and techniques work on these SSDs and (3) optimizing the current Linux I/O schedulers to make them SSD-friendly.

Acknowledgments We thank the ICPE'24 reviewers for their invaluable and constructive feedback. This work is funded by The Dutch Research Council (NWO) grant numbers OCENW.KLEIN.561 and OCENW.KLEIN.209. The authors would like to thank Jesse Donkervliet, Sacheendra Talluri, Matthijs Jansen, and the AtLarge group at VU Amsterdam for their help with the paper. Krijn Doekemeijer is funded by the VU PhD innovation program.

REFERENCES

[1] Accessed: 2024-01-29. BFQ Budget Fair Queueing Document. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html
[2] Accessed: 2024-01-29. Deadline I/O Scheduler Tunables. https://docs.kernel.org/block/deadline-iosched.html#:~:text=The%20goal%20of%20the%20deadline,value%20in%20units%20of%20milliseconds.
[3] Accessed: 2024-01-29. *Disaggregated or Hyperconverged, What Storage will Win the Enterprise?* https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/
[4] Accessed: 2024-01-29. *fio.* https://github.com/axboe/fio
[5] Accessed: 2024-01-29. *Free Your Flash and Disaggregate.* https://www.lightbitslabs.com/blog/free-your-flash-and-disaggregate/
[6] Accessed: 2024-01-29. getrusage(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/getrusage.2.html
[7] Accessed: 2024-01-29. Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html
[8] Accessed: 2024-01-29. Kyber Multiqueue I/O Scheduler. https://lwn.net/Articles/720071/
[9] Accessed: 2024-01-29. MQ-Deadline Implementation. https://elixir.bootlin.com/linux/latest/source/block/mq-deadline.c
[10] Accessed: 2024-01-29. *SPDK.* https://spdk.io/
[11] Accessed: 2024-01-29. *Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution.* https://americas.kioxia.com/en-us/business/news/2019/memory-20190805-1.html
[12] Accessed: 2024-01-29. Two New Block I/O Schedulers for 4.12. https://lwn.net/Articles/720675/
[13] Accessed: 2024-01-29. *What is Composable Disaggregated Infrastructure.* https://blog.westerndigital.com/what-is-composable-disaggregated-infrastructure/
[14] Jens Axboe. Accessed: 2023-01-26. Efficient I/O with io_uring. https://kernel.dk/io_uring.pdf

[15] Jon C. R. Bennett and Hui Zhang. 1997. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Trans. Netw.* 5, 5 (1997), 675–689.

[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[17] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols, SIGCOMM 1989*. ACM, 1–12.

[18] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proc. VLDB Endow.* 14, 3 (2020), 364–377.

[19] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring. In *SYSTOR '22: The 15th ACM International Systems and Storage Conference, 2022*. ACM, 120–127.

[20] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. 2023. Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS). In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 118–131.

[21] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. 1996. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the ACM SIGCOMM 1996 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 1996*. ACM, 157–168.

[22] Jiayang Guo, Yimin Hu, and Bo Mao. 2015. Enhancing I/O Scheduler Performance by Exploiting Internal Parallelism of SSDs. In *Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015. Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 9531)*. Springer, 118–130.

[23] Jiayang Guo, Yiming Hu, Bo Mao, and Suzhen Wu. 2017. Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*. IEEE Computer Society, 1184–1193.

[24] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[25] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017*. ACM, 127–144.

[26] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019*. USENIX Association, 301–314.

[27] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021*. USENIX Association, 113–128.

[28] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek R. Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*. USENIX Association, 519–532.

[29] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. 2014. HIOS: A Host Interface I/O Scheduler for Solid State Disks. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014*. IEEE Computer Society, 289–300.

[30] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. 2016. Improving I/O Performance of NVMe SSD on Virtual Machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016*. ACM, 1852–1857.

[31] Jieun Kim, Dohyun Kim, and Youjip Won. 2022. Fair I/O Scheduler for Alleviating Read/Write Interference by Forced Unit Access in Flash Memory. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, 2022*. ACM, 86–92.

[32] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016*. ACM, 29:1–29:15.

[33] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash = Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, 345–359.

[34] Michael Larabel. Accessed: 2023-11-16. Linux 5.6 I/O Scheduler Benchmarks: None, Kyber, BFQ, MQ-Deadline. https://www.phoronix.com/review/linux-56-nvme

[35] Michael Larabel. Accessed: 2024-02-28. BFQ I/O Scheduler For Linux Sees Big Scalability Improvement, published: 21 January 2024. https://www.phoronix.com/news/BFQ-IO-Better-Scalability

[36] Michael Larabel. Accessed: 2024-02-28. MQ-Deadline Scheduler Optimized For Much Better Scalability, published: 19 January 2024. https://www.phoronix.com/news/MQ-Deadline-Scalability

[37] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, 2020*. ACM, 591–605.

[38] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. 2020. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 1241–1255.

[39] Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, and Liting Hu. 2022. Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs. In *ICS '22: 2022 International Conference on Supercomputing, 2022*. ACM, 13:1–13:12.

[40] Hui Lu, Brendan Saltaformaggio, Ramana Rao Kompella, and Dongyan Xu. 2015. vFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015*. ACM, 125–138.

[41] Liuying Ma, Zhenqing Liu, Jin Xiong, and Dejun Jiang. 2022. QWin: Core Allocation for Enforcing Differentiated Tail Latency SLOs at Shared Storage Backend. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022*. IEEE, 1098–1109.

[42] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. 2019. K2: Work-Constraining Scheduling of NVMe-Attached Storage. In *IEEE Real-Time Systems Symposium, RTSS 2019*. IEEE, 56–68.

[43] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, 106–122.

[44] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*. USENIX Association, 361–378.

[45] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014*. ACM, 471–484.

[46] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012*. USENIX Association, 13.

[47] Bo Peng, Cheng Guo, Jianguo Yao, and Haibing Guan. 2023. LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSDs in Clouds. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023*. USENIX Association, 785–800.

[48] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*. USENIX Association, 665–676.

[49] Purestorage. Accessed: 2024-01-29. *Linux Recommended Settings*. https://support.purestorage.com/Solutions/Linux/Linux_Reference/Linux_Recommended_Settings

[50] RedHat. Accessed: 2024-01-29. *Chapter 20. Setting the Disk Scheduler*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/setting-the-disk-scheduler_managing-storage-devices

[51] Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, CHEOPS 2023*. ACM, 35–45.

[52] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference, 2013*. USENIX Association, 67–78.

[53] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. 2007. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies, FAST 2007*. USENIX, 61–76.

[54] Mingyang Wang and Yiming Hu. 2014. An I/O Scheduler Based on Fine-Grained Access Patterns to Improve SSD Performance and Lifespan. In *Symposium on Applied Computing, SAC 2014*. ACM, 1511–1516.

[55] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. 2023. Do We Still Need I/O Schedulers for Low-Latency Disks?. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 44–50.

[56] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. D2FQ: Device-Direct Fair Queueing for NVMe SSDs. In *19th USENIX Conference on File and Storage Technologies, FAST 2021*. USENIX Association, 403–415.

[57] Minhoon Yi, Minho Lee, and Young Ik Eom. 2017. CFFQ: I/O Scheduler for Providing Fairness and High Performance in SSD Devices. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, IMCOM 2017*. ACM, 87.