A Systematic Configuration Space Exploration of the Linux *Kyber* I/O Scheduler

Zebin Ren
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Krijn Doekemeijer
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Animesh Trivedi
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

ABSTRACT

NVMe SSDs have become the de-facto storage choice for high-performance I/O-intensive workloads. Often, these workloads are run in a shared setting, such as in multi-tenant clouds where they share access to fast NVMe storage. In such a shared setting, ensuring quality of service among competing workloads can be challenging. To offer performance differentiation to I/O requests, various SSD-optimized I/O schedulers have been designed. However, many of them are either not publicly available or are yet to be proven in a production setting. Among the widely-tested I/O schedulers available in the Linux kernel, it has been shown that *Kyber* is one of the best-fit schedulers for SSDs due to its low CPU overheads and high scalability. However, *Kyber* has various configuration options, and there is limited knowledge on how to configure *Kyber* to improve applications' performance. In this paper, we systematically characterize how *Kyber*'s configurations affect the performance of I/O workloads and how this effect differs with different file systems and storage devices. We report 11 observations and make 5 guidelines that indicate that (i) *Kyber* can deliver up to 26.3% lower read latency than the *None* scheduler with interfering write workloads; (ii) with a file system, *Kyber* can be configured to deliver up to 35.9% lower read latency at the cost of 34.5%–50.3% lower write throughput, allowing users to make a trade-off between read latency and write throughput; and (iii) *Kyber* leads to performance losses when *Kyber* is used with multiple throughput-bound workloads and the SSDs is not the bottleneck. Our benchmarking scripts and results are open-sourced and available at: https://github.com/stonet-research/hotcloudperf24-kyber-artifact-public.

CCS CONCEPTS

• **Software and its engineering** → **Secondary storage**; *Operating systems.*

KEYWORDS

Linux storage schedulers, Kyber, Measurements

ACM Reference Format:
Zebin Ren, Krijn Doekemeijer, and Animesh Trivedi. 2024. A Systematic Configuration Space Exploration of the Linux *Kyber* I/O Scheduler. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3629527.3651416

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

ACM ISBN 979-8-4007-0445-1/24/05.
https://doi.org/10.1145/3629527.3651416

1 INTRODUCTION

Modern high-performance solid-state drives (SSDs) are able to deliver millions of I/O operations per second (IOPS) with single-digit microsecond-level latency [5, 11, 12]. These devices are widely used in multi-tenant cloud environments for their improved performance over hard disks [30, 36, 46]. Cloud providers need to provide quality of service (QoS) guarantees for I/O services such as throughput or tail latency service-level objectives across multiple tenants. These guarantees are usually achieved by scheduling I/O requests with an I/O scheduler [29, 34].

However, existing Linux I/O schedulers designed for hard disks do not work well with these high-performance SSDs and induce significant CPU and scalability overheads [38, 42]. To reduce these overheads, there are many *state-of-the-art* I/O schedulers designed for SSDs [21, 24, 26, 27, 31–33, 35, 39, 41, 43]. Despite these studies on I/O schedulers for SSDs, using these past published I/O schedulers is challenging. Many of them do not have their source code public or are written for a specific kernel version, or assume specific hardware support from SSDs [24, 43]. Thus, users need to implement these I/O schedulers in the Linux kernel, which is not trivial, preventing their widespread use.

Compared to these *state-of-the-art* I/O schedulers, the *state-of-the-practice* plug-and-play Linux I/O schedulers [7], *Kyber* [6], *MQ-Deadline* [8], and *BFQ* [2], are the most accessible schedulers. In our past studies, we demonstrate that *Kyber* has a low CPU overhead and high scalability on fast SSDs and recommend using *Kyber* on high-performance SSDs for its low CPU overhead and high scalability [37, 38]. We also identify that *Kyber*'s configuration significantly impacts workload performance in terms of latency and throughput, and this impact also differs between different workloads [37]. *Kyber* provides two configurable parameters, read and write target latency, allowing users to set the target latencies that *Kyber* should try to deliver. The effect of *Kyber*'s configurations and the difference of this effect on different workloads create challenges in using *Kyber* in practice. There is no existing study on configuring *Kyber* for specific software and hardware settings. Specifically, how to find an optimized *Kyber* configuration with a specific setting for different (1) workloads, (2) file systems, and (3) types of SSDs.

In our study, we cover these three aspects to show the effect of *Kyber*'s configurations on its performance. Firstly, workloads have different I/O patterns and latency/throughput requirements [23, 24]. Existing studies of *Kyber* focus on its CPU and latency overhead, scalability, and its ability to deliver low latency for foreground workloads [23, 33, 38, 42]. There is a lack of systematic studies on how *Kyber*'s configurations affect the performance of interfering concurrent workloads with diverse demands in terms of expected read/write latencies and throughputs. Moreover, predicting the achieved performance with the latency targets is not trivial

since the user-specified latency targets are not guaranteed by *Kyber*. Thus, there is a gap between the target performance (*Kyber*'s configurable parameters of read/write latencies) and the achieved performance (latency and throughput), which is not obvious from the latency targets configured. Secondly, real-world workloads usually work with file systems instead of directly accessing the storage device. File systems change the I/O patterns of the workloads. Thus, the effect of *Kyber* on workload performance with different file systems is unknown. Thirdly, different types of SSDs have significantly different performance properties such as peak throughput, latency, and read/write interference behavior [23, 35]. For example, flash-based SSDs have unpredictable performance and read/write interference, but non-flash-based ultra-low latency (UUL) SSDs such as Intel Optane SSDs have stable performance and no read/write interference [44, 47].

In conclusion, the lack of understanding of how *Kyber*'s configurations affect the achieved performance with different workloads, file systems, and types of SSDs makes it unclear how to optimize *Kyber* in practice. Specifically, we investigate the following research questions (**RQs**) around how *Kyber*'s configurations affect the workloads' performance with different workloads, file systems and types of SSDs:

(RQ1) How does *Kyber* affect the performance of workloads when workloads run concurrently and interfere with each other? We investigate how *Kyber* affects the performance of different workloads by studying the relation between target latency and the workloads' achieved performance.

(RQ2) How to configure *Kyber*'s parameters for diverse types NVMe SSDs and diverse file systems to meet workloads' requirements? The key motivation is to find out if and how our findings on *Kyber*'s configurations performance effects can be generalized to different file systems and types of SSDs. We also provide guidelines on how to configure *Kyber* to meet the workloads' requirements in practice with diverse software and hardware environments.

To address these questions, we conduct a first-of-its-kind systematic study of Linux' *Kyber* I/O scheduler with various kinds of workloads, file systems, and types of SSDs to establish guidelines on how to configure *Kyber* in practice. Our key contributions in this work include:

- We extensively study how *Kyber* with different configurations affects workload performance using different combinations of latency-sensitive and throughput-bound workloads on 2 types of SSDs, resulting in 11 observations. To the best of our knowledge, we are the first to investigate the effect of *Kyber*'s configurations on workloads.
- Based on our observations, we provide 5 guidelines on how to configure *Kyber* in practice with various workloads, file systems, and types of SSDs.
- We open-source all artifacts, datasets, and scripts for this paper as FAIR data sets at https://github.com/stonet-research/hotcloudperf24-kyber-artifact-public.

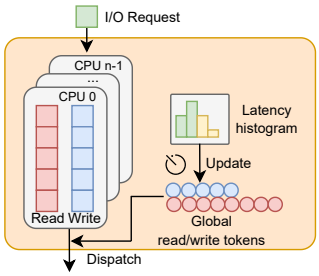


Figure 1: Architecture of the Linux *Kyber* I/O scheduler.

2 BACKGROUND

NVMe SSDs. Non-volatile memory express (NVMe) is an interface for accessing storage devices through PCIe. NVMe is widely used by high-performance SSDs. In this paper, we evaluate two kinds of NVMe SSDs: flash-based SSDs and non-flash-based SSDs with the 3D Xpoint technology [1].

Flash-based SSDs are composed of a controller that is connected to an array of flash chips. Each flash chip is organized in a hierarchy of dies, planes, blocks, and pages. SSDs have high internal parallelism as both dies and planes can operate in parallel. The NVMe protocol [10] exposes this parallelism to workloads with a multi-queue interface that allows SSDs to execute multiple I/O requests in parallel. Nevertheless, to fully utilize this parallelism, workloads need to issue multiple concurrent I/O requests to the SSD. A challenge here is that a plane can not execute different types of commands (read or write) in parallel. If a read is issued to a die where a write is already being executed, the read is blocked until the write finishes, leading to a 10–40× longer read latency. This performance degradation is called read/write interference [17, 45]. Moreover, the physical constraints of flash chips do not allow in-place updates or intra-block random writes. Pages in a block can only be written sequentially, and written pages need to be erased before they can be rewritten. Erasures happen at the unit of blocks, not at the unit of pages. To imitate the block interface provided by hard disks, the Flash Translation Layer (FTL) in SSD controllers maps logical addresses provided in the block interface to physical addresses in the flash chips. On an update, the data of the update is written to a new page, and the old page is marked invalid. The internal operations lead to additional interference with user I/O requests, leading to unpredictable performance. In conclusion, flash-based SSDs have (1) high parallelism, (2) unpredictable performance, and (3) read/write interference.

There are also non-flash-based SSDs such as Intel Optane SSDs [4], made with 3D Xpoint technology [1, 44]. 3D Xpoint has two big differences from flash: (1) it is byte-addressable, thus an I/O request can be broken into smaller pieces and processed in parallel by multiple channels to achieve low latency; and (2) it supports in-place updates and can, thus, provide stable performance without internal translation operations needed such as flash-based SSDs [47].

***Kyber* Internals.** *Kyber* is an I/O scheduler designed for fast and highly parallel storage devices inspired by active queue management techniques from network routing [6, 13]. *Kyber* prioritizes reads over writes based on the heuristic that a process that issues a read request usually waits for the issued read to finish. In contrast, a process that issues a write request usually continues executing

Table 1: Benchmarking environment.

Component	*Configuration*
CPU	Single socket Intel(R) Xeon(R) Silver 4210R CPU 10 cores @ 2.40GHz, Hyper-threading disabled, Turbo disabled.
Memory	256 GiB, DDR4.
Storage	Samsung 980 PRO 1 TiB (Flash-based SSD); Intel SSD 900P (Optane SSD)
Software	Ubuntu 20.04 with Linux kernel `v6.3.8`, `fio` v3.35.

without waiting for the write to finish. Figure 1 shows the architecture of the Linux *Kyber* I/O scheduler. *Kyber* maintains two queues for each CPU core, one for reads and one for writes. *Kyber* inserts I/O requests into the queues on the same core where the application issued the requests. These read/write queues are associated with a global token bucket. These tokens are used to limit the number of concurrent requests issued to the SSD to achieve high responsiveness. An I/O request is dispatched to the NVMe device driver only when there are available tokens. The number of tokens remains the same if both read and write target latencies are satisfied, and is increased if read or write P99 latency exceeds the target latency. Increasing the number of tokens increases the priority of that workload. The number of NVMe 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; and (2) the achieved P99 latency for the other type is higher than the target latency. *Kyber* aims to deliver the user configured target latencies. However, there is no guarantee that *Kyber* achieves the target latencies. Further on, it is not studied how these target latencies affect the achieved workload throughput and latency. The default read and write target latencies are 2 ms and 10 ms, respectively. However, achievable latencies for NVMe SSDs range from 10 to 80 μs and differ between different types of SSDs [4, 11]. Therefore, there is a huge gap between the default target latencies and the best latencies that NVMe SSDs can deliver. It is unknown how this gap and the performance difference of SSDs affect workload performance on NVMe SSDs when using *Kyber*. The aim of this paper is to investigate how *Kyber*'s target latencies and the performance of different SSDs affect the achieved workload throughput and latency.

3 METHODOLOGY

Hardware and Software. Our benchmarking environment is shown in Table 1. We use `fio` [3] as a workload generator with the io_uring interface [14]. All the I/O requests are issued with the `O_DIRECT` flag so the I/O requests bypass the page cache. We use two metrics to evaluate performance: throughput and latency. We measure throughput in I/O operations per second (IOPS) and latency in 99 percentile operation tail latencies (P99 latency). Before running the experiments, we precondition the flash SSD according to [16]—by sequentially writing the entire SSD, then writing 2 TiB of 4 KiB random writes. We run each experiment on the Samsung 980 PRO for 12 minutes (6 minutes warm-up time + 6 minutes run time) with five repetitions. For each experiment, we report the average(s) of these five runs. On the Intel Optane SSD, we run each experiment with one repetition for 2 minutes and 30 seconds (30 seconds warm-up time + 2 minutes run time). We use a shorter run time for the experiments on the Optane SSD because it delivers stable performance.

Synthetic Workloads and Methodology. Workloads in cloud environments have diverse I/O requirements, such as latency-sensitive workloads (e.g., online database query) and throughput-bound workloads (e.g., batch processing systems) [24]. We use two synthetic workloads: latency-sensitive workloads (L-app) and throughput-bound workloads (T-app). Both only issue 4 KiB read or write requests. For the L-apps, we issue a single outstanding request (we use queue depth, or QD to represent 'the number of outstanding requests' for simplicity in later sections). The T-apps issue 256 outstanding requests to saturate the SSDs. In the following sections, we use `R1` and `W1` to represent L-app read and write workloads respectively and `R256` and `W256` to represent T-app read and write workloads respectively. The number after `R` and `W` represents the QD of the workload.

Table 2: Baseline performance of Samsung 980 PRO SSD with the *None* scheduler.

#	Workload(s)	R TP (in KIOPS)	W TP (in KIOPS)	R P99 Lat (in μs)	W P99 Lat (in μs)
1	`R1`	17.0	-	77.5	-
2	`R256`	364.3	-	793.8	-
3	`W1`	-	62.3	-	23.1
4	`W256`	-	70.0	-	15,794.2
5	`R1-W1`	4.0	65.0	1,879.2	26.8
6	`R1-W256`	0.3	68.9	15,217.5	15,558.2
7	`R256-W1`	302.6	61.5	3,044.1	32.1
8	`R256-W256`	83.2	93.1	15,283.0	15,938.4

4 BASELINE PERFORMANCE WITH THE *NONE* SCHEDULER

As we explained in §2, flash-based SSDs have read/write interference, which means that a write blocks concurrent reads to the same die. In this section, we establish the baseline performance of the evaluated flash SSDs with and without interference. We report the read/write throughput and latency with different workload combinations (i.e., L-app, T-app). We use the *None* scheduler, a no-op scheduler, which passes the I/O requests to the NVMe device driver in a first-in-first-out manner. Each workload is pinned to a dedicated CPU core to avoid interference by the process scheduler. Table 2 shows the throughput (in KIOPS) and P99 tail latency (in μs) of different workload combinations. We show the (combination) of workloads in the second column and we show the throughput and latency of the read and write workloads in the third to sixth columns. We have three observations:

Asymmetric read/write performance. The flash SSDs have asymmetric read/write performance **(Observation 1, O-1)**. With a single CPU core and no interfering workloads, the flash SSD delivers up to 364.3 KIOPS random read throughput at QD=256 and 77.5 μs P99 random read latency at QD=1 (row 1). When `fio` issues random writes, the flash SSD delivers up to 70.0 KIOPS throughput at QD=256 (row 4) and 23.1 μs P99 latency at QD=1 (row 3). In short, the flash SSD has different throughput and latency for reads and writes without interference. Next, we show how this performance changes with the interference of a second workload.

Writes have a huge impact on read performance. A concurrent write workload significantly degrades the performance of a co-running read workload. When a latency-sensitive read workload `R1` is mixed with a latency-sensitive write workload `W1` (row 5), the read throughput drops 76.5% (from 17.0 to 4.0 KIOPS) and the latency increases 24.2× (from 77.5 to 1,879.2 μs) compared to `R1` without interference (row 1). The read performance degradation is more significant with a throughput-bound write workload `W256` (row 6), showing 98.2% lower throughput (from 17.0 to

Table 3: Evaluated workloads combinations with *Kyber*.

	L-app with write (W1)	T-app with write (W256)
L-app with read (R1)	R1–W1	R1–W256
T-app with read (R256)	R256–W1	R256-W256

0.3 KIOPS) and 200.8× higher latency (from 77.5 to 15,217.5 μs) than R1 without interference. When a read throughput-bound workload and a write throughput-bound workload compete for throughput (row 8), the read workload has 77.2% lower throughput (from 364.3 to 83.2 KIOPS) than without the interference of concurrent writes. To conclude, the read performance is highly sensitive to the write workload, changing the write workload leads to a significant effect on read throughput and latency **(O-2)**.

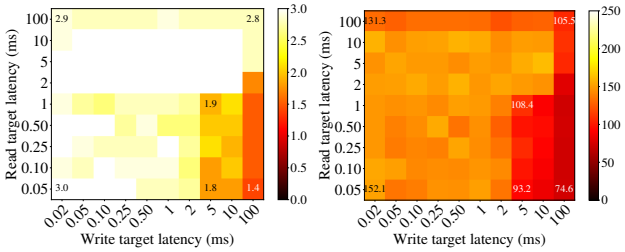
Reads have a less significant impact on write performance. A co-running read workload has less impact on the write performance than the impact write workloads have on read workloads. When W1 runs with R1 in the background (row 5), the write workload has comparable throughput (from 62.3 to 65.0 KIOPS), and the latency only increases by 16.0% (23.1 to 26.8 μs) compared to running W1 in isolation. With a throughput-bound workload R256 on the background (row 7), the write latency increases 39.0% (from 23.1 to 32.1 μs), much lower than the latency increase of reads in this setting (200.8×). Thus, the write performance is less sensitive to the read workload than the interference of writes on reads **(O-3)**.

The key finding here is that the *None* scheduler can not mitigate read/write interference. When a latency-sensitive read workload runs concurrently with a write workload, the read workload has a significantly higher P99 tail latency than the read workload running in isolation. When there are two throughput-bound read and write workloads competing for throughput, *None* does not provide any functionality to tune the throughput share between the read and write workloads. *Kyber* offers configuration options that let the users prioritize reads or writes over each other. Thus, in the following sections, we investigate if and how *Kyber* affects read/write interference and how it affects the throughput share between throughput-bound read and write workloads under different configurations.

We repeat the same benchmark on an Intel Optane P900 SSD (the results are not plotted in the paper). We have two observations. Firstly, the Optane SSD have symmetric read/write performance. Unlike the flash SSD, the Optane SSD deliver comparable throughput and latency for both reads and writes. Secondly, the Optane SSD have less read/write interference. R1 has up to 65.6% higher latency (15.4 vs. 44.8 μs) with a concurrently running W256 compared to running R1 in isolation, which is significantly lower than the Samsung 980 PRO (24.2× lower). We show how *Kyber* and it's configurations affect the performance of these workloads in §5.

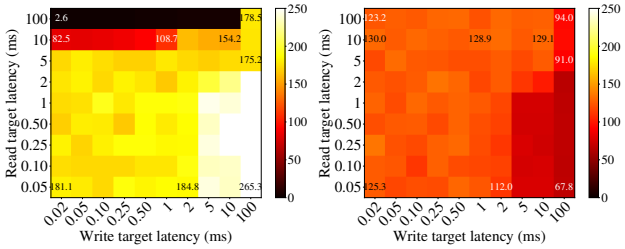
5 PERFORMANCE EFFECT OF *KYBER*'S CONFIGURATIONS WITHOUT A FILE SYSTEM

We start our analysis with a performance characterization of the impact of different *Kyber* configurations on fio-based micro-benchmarks. We run these micro-benchmarks without any file system. Specifically, we investigate how different *Kyber* configurations affect the P99 latency of L-apps and throughput of T-apps when concurrent



(a) R1 P99 latency (in ms) **(b) W256 throughput (in KIOPS)**

Figure 2: Performance of L-app (read-only) and T-app (write-only) workload combinations with different *Kyber* configurations.



(a) R256 throughput (in KIOPS) **(b) W256 throughput (in KIOPS)**

Figure 3: Performance of T-app read and write workload combinations with different *Kyber* configurations.

read and write workloads interfere with each other, see Table 3 for all combinations. Such a setup is common in the multi-tenant cloud. For each benchmark, we start two concurrent fio processes. One fio process issues reads and one process issues writes. Each fio process is pinned to a separate and dedicated CPU core to prevent them from competing for CPU resources. We do a grid search to investigate how *Kyber*'s configurations affect the achieved performance of fio workloads in the search space. We set the lowest read and write target latency to 50 μs and 20 μs respectively, based on the minimum P99 latency of the flash SSD (§4), and we gradually increase the target latency to 100 ms. We report our performance results in Figure 2 and Figure 3 as heatmaps where the x-axis represents the write target latency and the y-axis represents the read target latency. The temperature in the heatmaps is the measured performance with read and write target latencies set to corresponding values in y- and x-axis.

How does *Kyber* affect the performance of different combinations of workloads? We report that *Kyber*'s configurations do not have a significant effect on the performance of workload combinations R1–W1 (thus they are not plotted in the paper) **(O-4)**. We do not observe a relation between the P99 read/write latencies and *Kyber* configurations. The P99 read latency varies between 1.3 and 1.6 ms and the P99 write latency varies between 23.4 and 27.3 μs. The reason that *Kyber* does not have a significant effect on R1–W1 is that *Kyber*'s mechanism is only effective with multiple outstanding I/O requests. Yet, with R1–W1, there is only 1 outstanding read and write request. Since *Kyber* allows at least one read and one write to be sent to the SSDs, thus *Kyber* does not throttle the request with R1–W1. We report that *Kyber* is effective when there is at least

more than one outstanding read or more than one outstanding write **(Guideline 1, G-1)**.

Can *Kyber* provide bounded P99 latency for the L-app when an L-app interferes with a T-app? Figure 2 shows how *Kyber*'s configuration affects `fio`-workload performance when a read L-app (`R1`) and a write T-app (`W256`) run concurrently. The temperature in Figure 2a shows the read P99 latency (in ms, darker is better) of the read L-app and the temperature in Figure 2b shows the throughput (in KIOPS, lighter is better) of the write T-app. In our experiments, *Kyber* mitigates read/write interference at the cost of write throughput **(O-5)**. When the read target latency is set to 50 μs and the write target latency is set to 100 ms, the achieved read P99 latency is 1.4 ms, 26.3% lower (1.8 ms) than the read latency of `R1–W1` with the *None* scheduler (row 5, Table 2). Thus, *Kyber* delivers low read latency with background throughput-bound write workloads by setting the read target latency to the lowest read P99 latency that the SSD can achieve (50 μs in our case) and the write target latency to a value higher than the achieved write latency with the *None* scheduler (15.6 ms, row 6, Table 3). However, the cost of achieving low read latency is lower write throughput (from peak 152.1 to 74.6 KIOPS, 50.9% lower throughput). We suggest using *Kyber* in multi-tenant situations when low read latency is considered more important than high throughput **(G-2)**.

How do *Kyber*'s configurations affect the throughput share of two throughput-bound `fio`-workloads? Figure 3 shows how *Kyber*'s configurations affect the interference between a read T-app and a write T-app. The temperature in Figure 3a shows the read throughput and the temperature in Figure 3b shows the write throughput (in KIOPS, lighter is better). Firstly, decreasing the read target latency leads to higher read throughput. With a fixed write target latency (fixed x value), as the read target latency decreases, the read throughput increases **(O-6)**. For example, with the write target latency is set to 20 μs (first column in Figure 3a), as the read target latency decreases from 100 ms to 50 μs, the read throughput increases from 2.6 KIOPS to 181.1 KIOPS, a 69.7× increase. Secondly, when the read target latency is lower than 10 ms and the write target latency is lower than 5 ms, changing *Kyber* configurations does not lead to a statistical difference in read and write throughput. The reason is that the achieved read and write latencies are 5 ms and 18 ms (not visualized). Tuning the target latencies in a configuration space where all the candidate values are much lower than the lowest achievable latency, all the configurations in this configuration space lead to comparable performance (we call this space the *dead configuration space*). In conclusion, by tuning *Kyber*'s configuration, the throughput between reads and writes can be distributed. We suggest that the users (1) run this grid search micro-benchmark in Figure 2 and Figure 3 to find out how the target latencies affect the performance of a specific SSD and (2) avoid tuning *Kyber* in the dead configuration space **(G-3)**.

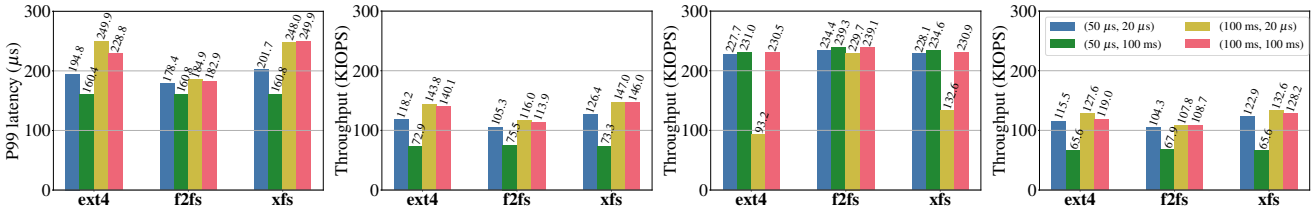
How does this effect change with different SSDs? We repeat our experiments on the Intel Optane SSD to investigate how this effect varies across different SSDs. Firstly, similar to the Samsung SSD, we observe that with `R1–W256`, prioritizing reads by setting low read target latency and high write target latency leads to lower P99 read latency at the cost of write throughput. When *Kyber* is configured to prioritize reads, it delivers 62.5% lower latency (from 44.8 to 16.8 μs) than it does with the *None* scheduler at the cost of 67.1% lower write throughput (from 228.2 to 75.0 KIOPS) **(O-7)**. Secondly, when two throughput-bound workloads interfere with each other (`R256–W256`), we report that prioritizing reads leads to lower write throughput (from 202.5 to 68.9 KIOPS) and comparable read throughput when prioritizing writes. However, when neither reads nor writes are prioritized, setting the read and write latency to the same value leads to high read and write throughputs (215.4 and 203.1 KIOPS, respectively) at the same time. The explanation for this phenomenon is that the Optane SSD has low read/write interference [44]. When the SSD is not saturated, adding a concurrent write workload with a read workload does not have a significant effect on the performance of the read workload. In this setting (`R256–W256`), the SSD is not saturated. In short, limiting read (or write) throughput does not increase the write (or read) throughput on the Optane SSD. With the Optane SSD, a misconfiguration when the SSD is not saturated leads to a throughput drop for reads or writes without any throughput increase for writes or reads **(O-8)**. Thus, we suggest using *Kyber* with Optane SSDs only when the SSDs are the bottleneck.

6 PERFORMANCE EFFECT OF *KYBER*'S CONFIGURATIONS WITH FILE SYSTEMS

In the previous section, we investigate how *Kyber* affects the performance of `fio`-workloads without using any file system. However, real-world workloads usually access SSDs via file systems. In this section, we characterize how *Kyber*'s configuration affects the I/O performance with three different file systems: ext4 [9], f2fs [28], and xfs [22]. The goal is to investigate if the observations of our microbenchmarks (§5) generalize to file systems. We evaluate two workload combinations: `R1–W256` and `R256–W256` in Table 3 with four *Kyber* configurations where the target read and write latency is set to (50 μs R, 20 μs W), (50 μs R, 100 ms W), (100 ms R, 20 μs W) and (100 ms R, 100 ms W), the four extreme configurations in the configuration search space in the previous section. The performance of the `fio` workloads is reported in Figure 4.

How does *Kyber* affect the I/O performance with the use of a file system? We first investigate how *Kyber*'s configurations affect the performance of `R1–W256` with different file systems. Figure 4a and Figure 4b show the P99 read latency (in μs, the lower the better) and write throughput (in KIOPS, the higher the better) respectively with workload `R1–W256`. *Kyber* delivers the lowest read P99 latency with configuration (50 μs R, 100 ms W), 13.0%–35.9% lower latency than the worst configuration (160.4–160.8 μs vs. 184.9–249.9 μs) . This lower P99 read latency comes at the cot of lower write throughput (72.9–75.5 KIOPS or 34.9%–50.1% lower) compared to the highest write throughput (116.0–147.0 KIOPS) **(O-9)**. If the workloads access the SSD via a file system, *Kyber* can be configured to deliver up to 35.9% lower read P99 latency than the P99 read latency delivered in other configurations with concurrent background writes **(G-4)**.

Next, we investigate how *Kyber*'s configurations affect the throughput when a read throughput-bound workload and a write throughput-bound workload run concurrently. Figure 4c and Figure 4d show the read and write throughput with workload setting `R256–W256`. We have two observations. Firstly, the workloads with ext4 and xfs have similar performance. Configuring *Kyber* to prioritize read (e.g.,



(a) R latency in R1–W256 (b) W throughput in R1–W256 (c) R throughput in R256–W256 (d) W throughput in R256–W256

Figure 4: Performance of R1–W256 and R256–W256 combinations with different *Kyber* configurations and ext2, f2fs and xfs.

50 μs read latency and 100 ms write latency, the second bar in each group) does not lead to significantly higher read throughput compared to the other three settings (from 227.7 and 228.1 KIOPS to 231.1 and 234.6 KIOPS, 1.3% and 2.8% higher read throughput). However, the write throughput is significantly decreased (from 127.6 and 132.6 KIOPS to 65.6 KIOPS, 48.6% and 50.5% lower throughput). The same occurs when we configure *Kyber* to prioritize writes over reads (e.g., 100 ms write latencies and 20 μs read latencies, the third bar in each group) **(O-10)**. Secondly, *Kyber*'s configurations do not have a significant effect on the read throughput of f2fs (the read throughput is 229.7–239.3 KIOPS). However, prioritizing reads causes the write throughput to decrease from 108.7 KIOPS to 67.9 KIOPS, a 37.5% lower write throughput **(O-11)**. Thus, we recommend that users should configure *Kyber* with the same read and write target latencies. In our setup, the read and write target latencies are set to (50 μs R, 20 μs W) and (100 ms R, 100 ms W) to achieve both read and write peak throughput **(G-5)**.

In conclusion, when a latency-sensitive workload runs concurrently with a throughput-bound write workload via a file system, *Kyber* can be configured to deliver low read P99 latency by setting low read target latency and high write target latency to prioritize reads. When there are read and write throughput workloads running concurrently, we suggest setting the read and write target latencies to similar values to achieve high read and write throughput.

7 RELATED WORK

I/O schedulers for flash SSDs. Our study focuses on the *state-of-the-practice* Linux I/O scheduler *Kyber*. However, there are many *start-of-the-art* I/O schedulers for SSDs for Linux.

Designing fair-sharing I/O schedulers has been extensively studied with SSDs [15, 18, 21, 35, 39, 40, 43, 48]. MQFQ [21] utilizes the multi-queue interface to increase its scalability. D2FQ [43] further increases the performance of fair-sharing I/O schedulers by eliminating the "stage" step and offloading the scheduling to SSDs using the weighted round-robin feature [25].

There are also I/O schedulers that are optimized to deliver low latency for latency-sensitive workloads in shared environments [24, 31, 33]. K2 [33] strictly prioritizes high-priority requests and trades throughput for latency. blk-switch [24] provides low latency for high-priority workloads and preserves high total throughput at the same time. FastResponse [31] co-designs the I/O scheduler with the storage stack to reduce the I/O interference.

Flash-based SSDs have many idiosyncrasies because of their complex internal architectures. Various I/O schedulers are built to utilize these idiosyncrasies to increase SSDs' write performance and lifespan by using fine-grained access [41], reducing SSD GC overhead [19, 20, 26] and reducing read/write interference [27, 35].

Performance characterization of Linux I/O schedulers. Many studies characterize the performance of Linux I/O schedulers with NVMe SSDs [37, 38, 42]. Whitaker et al. [42] characterize the performance of the Linux I/O schedulers on ULL SSDs based on 3D XPoint technology. Their findings include that Linux I/O schedulers lead to higher latency, lower throughputs, and higher energy overhead than without the I/O schedulers. Ren et al. [37] extended this work by characterizing the performance overhead, scalability, QoS with more common flash-based SSDs. Additionally, they characterize how *Kyber*'s configurations affect the interference between foreground read workloads and background write workloads. We extend this work on *Kyber* by characterizing the performance of *Kyber* with different combinations of workloads and how these effects can be generalized to different file systems. We presented an in-depth, systematic study to give guidelines on how to configure *Kyber* with specified SSDs and workloads.

8 CONCLUSION AND FUTURE WORK

In this paper, we investigate how *Kyber*'s configurations affect the performance of different workloads with various file systems and storage devices. Our results show that *Kyber* can be configured to deliver low read latency when there is a concurrently running write workload. *Kyber* can also be used to balance the throughput share between read and write throughput-bound workloads when the applications directly run on the top of block devices.

This work can be expended in (1) evaluating how *Kyber*'s configuration affects the performance of applications with mixed read and write workloads, (2) designing an automatic tool that can find the best *Kyber* configuration automatically, and (3) designing algorithms that dynamically configures *Kyber* when the workload changes.

Acknowledgments This work is partially supported by Netherlands-funded projects from the Dutch Research Council (NWO) grants (OCENW.KLEIN.561 and OCENW.KLEIN.209) and GFP 6G FNS, and EU-funded projects MCSA-RISE Cloudstars and Horizon Graph-Massivizer. Krijn Doekemeijer is funded by the VU PhD innovation program. We thank the anonymous HotCloudPerf'24 reviewers for their invaluable and constructive feedback. We would also like to thank Jesse Donkervliet, Sacheendra Talluri, Matthijs Jansen, and the AtLarge group at VU Amsterdam for their help with the paper.

REFERENCES

[1] Accessed: 2024-03-13. 3D XPoint. https://insidehpc.com/2015/07/intel-and-micron-announce-3d-xpoint-non-volatile-memory/

[2] Accessed: 2024-03-13. BFQ Budget Fair Queueing Document. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html

[3] Accessed: 2024-03-13. fio. https://github.com/axboe/fio

[4] Accessed: 2024-03-13. Intel Optane 900P Techinical Specification. https://www.intel.com/content/www/us/en/products/sku/123623/intel-optane-ssd-900p-series-280gb-2-5in-pcie-x4-20nm-3d-xpoint/specifications.html

[5] Accessed: 2024-03-13. 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

[6] Accessed: 2024-03-13. Kyber Multiqueue I/O Scheduler. https://lwn.net/Articles/720071/

[7] Accessed: 2024-03-13. Linux I/O Schedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers

[8] Accessed: 2024-03-13. MQ-Deadline Implementation. https://elixir.bootlin.com/linux/latest/source/block/mq-deadline.c

[9] Accessed: 2024-03-13. The New ext4 Filesystem: Current Status and Future Plans. https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf

[10] Accessed: 2024-03-13. NVM Express. https://nvmexpress.org

[11] Accessed: 2024-03-13. Samsung 980 PRO PCIe 4.0 SSD. https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/

[12] Accessed: 2024-03-13. Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution. https://americas.kioxia.com/en-us/business/news/2019/memory-20190805-1.html

[13] Accessed: 2024-03-13. Two New Block I/O Schedulers for 4.12. https://lwn.net/Articles/720675/

[14] Jens Axboe. Accessed: 2024-03-13. Efficient I/O with io_uring. https://kernel.dk/io_uring.pdf

[15] 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.

[16] 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.

[17] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. 2017. Exploiting Intra-Request Slack to Improve SSD Performance. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017*. ACM, 375–388.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Christoph Hellwig. 2009. XFS: The Big Storage File System for Linux. *login Usenix Mag.* 34, 5 (2009).

[23] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. 2022. IOCost: Block IO Control for Containers in Datacenters. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating System 2022*. ACM, 595–608.

[24] 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.

[25] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. 2017. Enabling NVMe WRR Support in Linux Block Layer. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association.

[26] 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.

[27] 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.

[28] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015*. USENIX Association, 273–286.

[29] 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.

[30] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2022. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Trans. Storage* 18, 1 (2022), 5:1–5:21.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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. *To Appear in the Proceedings of the 2024 ACM/SPEC International Conference on Performance Engineering, ICPE 2024*. (2024).

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] Kan Wu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019*. USENIX Association.

[45] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. 2019. HotR: Alleviating Read/Write Interference with Hot Read Data Replication for Flash Storage. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019*. IEEE, 1367–1372.

[46] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015*. ACM, 6:1–6:11.

[47] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Trans. Model. Perform. Evaluation Comput. Syst.* 5, 1 (2020), 4:1–4:28.

[48] 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.