Do we still need IO schedulers for low-latency disks?

Caeden Whitaker, Sidharth Sundar, Bryan Harris, Nihat Altiparmak
Dept. of Computer Science & Engineering, University of Louisville
{caeden.whitaker,sidharth.sundar,bryan.harris.1,nihat.altiparmak}@louisville.edu

ABSTRACT

The performance of recent data storage devices has significantly improved over previous generations, with lower latency, greater throughput, and greater parallelism. Since we now have Ultra-Low Latency (ULL) data storage devices capable of providing data in less than 10 microseconds, in this paper we question the need for IO schedulers for better performance and energy efficiency. Specifically, we measure the latency costs of Linux IO scheduling algorithms and investigate their impact on overall performance and energy efficiency using a ULL storage device, a power meter, and various IO workloads. Our observations indicate that IO schedulers for ULL storage either do not help or significantly increase request latencies while also negatively impacting throughput and energy efficiency. Although we recognize the value of IO schedulers for slower devices or for other metrics such as fairness and QoS, we believe that IO schedulers have become unnecessary for ULL devices to improve performance or energy efficiency.

CCS CONCEPTS

• **Software and its engineering → Secondary storage**; • **Information systems → Storage power management**.

KEYWORDS

IO scheduler, ultra-low latency storage, energy efficiency

ACM Reference Format:
Caeden Whitaker, Sidharth Sundar, Bryan Harris, Nihat Altiparmak. 2023. Do we still need IO schedulers for low-latency disks?. In *15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23), July 9, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3599691.3603400

HotStorage '23, July 9, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603400

1 INTRODUCTION

Disk IO schedulers are traditionally used to improve performance of storage devices. For example, a hard disk drive (HDD) is limited by mechanics—the rotational speed of its magnetic platter and the movement of its actuator arm. It is also a *sequential* device; its read/write head serves only one request at a time and can more easily access long contiguous regions. In order to use an HDD efficiently, the operating system must carefully plan, or *schedule*, the dispatch of IO requests to the device to obtain efficient actuator arm movements and reduced seek time. An IO scheduler may also merge requests of adjacent data to improve sequentiality. Using these techniques, effective system software can save many milliseconds per request through efficient IO scheduling—large gains when compared to the cost of a few microseconds of processing. IO schedulers for HDDs are essential and necessary tools *for performance* in any OS design, backed by decades of research and understanding.

These properties change when the physical mechanics are removed. Solid-state drives (SSDs) are based on newer technologies (such as flash or phase change memory) without the mechanical limitations of HDDs, resulting in greater performance and lower latency, on the order of 10s of microseconds. With this improved latency, an IO scheduler requiring a few microseconds of processing time has become a significant portion of the response time of an application's IO request. In addition, the internal structure or design of an SSD, such as the flash translation layer (FTL), is typically managed exclusively by the SSD controller and abstracted away from the host's system software. The IO scheduler is therefore incapable of making scheduling decisions based on the SSD's internal architecture, and so the host must consider the SSD as a *random access* device; at any time, it expects to access data at any physical location with similar latency. There are some specialized (e.g. open-channel, ZNS) SSDs that expose their internals to the host, but this management further adds to the costs of system software. In addition, SSDs are internally *parallel*, capable of serving multiple requests simultaneously—support for which is carried to the operating system with advances in interfaces and bus architectures such as NVMe over PCIe. These changes in hardware motivate changes in system software; for example, a multi-queue design for the Linux block layer (*blk-mq*) has replaced the previous single dispatch queue designed for HDDs, enabling parallel IO dispatching across multiple

CPU cores. Due to the random access and parallel nature of SSDs, and their abstracted internal structure, the need for scheduling to achieve performance becomes questionable, especially considering the now significant processing cost of scheduling. This relative cost will only increase as devices become ever faster.

Ultra-low latency (ULL) storage is defined as having less than 10 µs access latency [15]. Such devices can be based on flash technology, such as Samsung's Z-SSDs [3], or phase-change memory such as Intel's Optane SSDs [4, 11]. With less than 10 µs device latency, the relative time spent by system software becomes ever more significant. Software, rather than hardware, has now become the bottleneck to further reducing access latency, motivating a reexamination of all components in the storage stack. In this paper, we investigate the costs of IO scheduling using an Intel Optane ULL SSD, and evaluate if these costs are still worth their benefits to performance and energy efficiency, while leaving their impact on fairness and QoS as a future work.

2 BACKGROUND

Traditional disk IO schedulers were designed for hard disk drives, and reorder requests for efficient access and movement of the physical components of an HDD, while perhaps also considering application fairness. Since HDDs are sequential devices that can typically service only one request at a time, the operating system uses a single software queue on which the scheduler operates in order to reorder or merge requests based on its scheduling algorithm. Previous versions of Linux that used this single queue design had a choice of three schedulers: *noop*, *deadline*, and *completely fair queueing* (CFQ). The *noop* scheduler is the simplest, which performs no reorganization of requests (first in, first out), but merges sequential requests. The *deadline* scheduler was designed to prevent starvation by imposing a deadline to start requests. It generally favors reads over writes, as APIs often block on reads. CFQ was designed to enforce fairness using per-process queues, which dispatch requests based on managed time slices assigned to each queue. As storage devices became faster, and computer systems grew more parallel with more CPU cores and the introduction of parallel SSDs, this single queue design (with a single lock) became a performance bottleneck. Kernel developers soon considered single queue systems to be incapable of achieving more than one million IOPS, regardless of the number of CPU cores [9], thus motivating a new design.

The Linux multiqueue block design (*blk-mq*) was fully implemented by kernel version 3.16 (in 2014). This parallel design provides separate software staging queues for each CPU core (removing the need for a lock) and multiple hardware dispatch queues that map to the device driver. Together with a parallel driver, such as the NVMe interface that specifies multiple submission/completion pairs on a device controller, this *blk-mq* design enables a fully parallel storage stack from CPU cores to the device controller. The IO scheduler operates only on the software staging queues, and can reorder requests within each queue, but not across queues. The scheduler can be selected or changed at run time through *sysfs* [7]; each request queue simply has a pointer to a struct of kernel functions that implement the scheduler. These functions may be built into the kernel or implemented in dynamically loaded kernel modules. With blk-mq, IO merging is also moved outside of the schedulers and performed by default. The introduction of *blk-mq* required new implementations of IO schedulers. Linux currently includes four schedulers:

none is not a "plug and play" scheduler, but rather the code executed when the pointer to the scheduler's functions is *NULL*. It simply adds new requests to the end of the software queue, providing a simple FIFO behavior with no reorganization. It is currently the default scheduler for many popular distributions such as Ubuntu Linux.

mq-deadline is the revision of the single-queue *deadline* scheduler for *blk-mq* [1]. It uses timestamps to enforce fairness and achieve quality of service by implementing queues sorted by requests' assigned "deadlines," which also aims to prevent the starvation of any single request. *mq-deadline* is currently the default scheduler for AHCI devices in Ubuntu Linux.

kyber was originally designed with "fast devices" and "multiple queues" in mind [2], which uses dispatch tokens to balance IO across a number of domains. It allows the user to tune latency goals, and is capable of dynamically adjusting the importance of fairness in order to reach those goals. Namely, *kyber* can monitor the current average IO latency relative to the target and adjust the supply of dispatch tokens to manage latency. To ensure that synchronous requests are not starved by asynchronous requests, a proportion of the queue depth is reserve for each type of request. The policies that *kyber* uses to manage tokens were inspired by network routing techniques, mainly from block buffered writeback throttling (*blk-wbt*) [2].

bfq is a multi-queue extension of the single queue version of *budget fair queueing*, which drew inspiration from *cfq*. *bfq* aims to achieve low latencies for real time tasks [10] through the use of a large set of heuristics to detect application type. Each application is allocated a scheduling queue [20] and *bfq* maps requests from many applications onto the available software queues according to the application's bandwidth budget. The software complexity of *bfq*'s heuristics and internal queues likely increases its processing cost when compared to other IO schedulers. *bfq* is currently the default scheduler for AHCI devices in Fedora Linux, and for Chromebooks [16].

3 METHODOLOGY

3.1 Experimental Setup

Our experiments were run on a Dell PowerEdge R230 (Intel Xeon E3-1230 quad core, eight threads, 3.4 GHz, 64 GB RAM) computer with an Intel Optane P4801X Series NVMe SSD as our storage device. We installed Ubuntu 22.04 and upgraded the Linux kernel to 5.18. We connected the sole power supply of our test system to an Onset HOBO UX120-018 Data Logger [17] to measure the energy consumption of the system. The system clock of our test system and the power meter were synchronized using a time server.

3.2 Workloads

First, we use microbenchmark workloads to investigate the limits of the storage system, using single- and multi-tenant experiments. Second, we use a macrobenchmark to produce IO from a real database application.

3.2.1 Microbenchmarks. We used the Flexible IO Tester (*fio*) version 3.31 to generate our microbenchmark workloads. We used io_uring as the IO interface due to its popularity, high-performance settings, and asynchronous IO capability [8]. Each workload is repeated for each of the four schedulers *none*, *mq-deadline*, *kyber*, and *bfq*. We present the average of five test runs.

To investigate the effects of schedulers, it is useful to have block layer (*blk-mq*) queues of significant and known length on which the schedulers operate, to allow for reordering the requests. The total number of outstanding requests in the system is the overall *IO depth*, which we distribute in two ways with two workload types: single- and multi-tenant. For both types, we measure workloads with a total IO depth from 1 to 128 (using powers of 2). Our single-tenant workloads use a single application process running alone, continuously and asynchronously submitting requests to maintain the desired IO depth on a single block layer queue. In contrast, our multi-tenant workloads use multiple processes submitting single requests, thus maintaining the total IO depth across multiple queues. Both single- and multi-tenant workloads submit three types of requests: random reads, random writes, and a mixture of 50% reads and 50% writes. In addition, as a variation on the multi-tenant workloads, we measured an additional scenario, where half the processes submit 4 KB reads and the other half submit 8 KB reads.

3.2.2 Macrobenchmarks. In order to draw practical conclusions of schedulers, it is critical to analyze realistic workloads. *fio* is able to test the limits of IO in a system because it incurs minimal CPU overhead in its workloads. However, to demonstrate a workload with a more realistic mix between CPU usage and IO, we tested the commonly used RocksDB [6, 18] key-value store database. In particular, RocksDB is known for optimizing towards lower latency key-value lookups. This makes it an ideal application to examine the impact of IO scheduling on ULL devices, because it is more likely than non-latency-sensitive applications to benefit from the use of ULL devices.

The *db_bench* tool that accompanies RocksDB was used to generate the macrobenchmarks, which demonstrate the performance and energy efficiency characteristics of IO schedulers. For each benchmark, we had a setup and experimental phase. In the setup phase, we generated a RocksDB database on our Intel Optane storage device, which otherwise contained only an *xfs* filesystem. This database consists of 23M key-value pairs. Each key was 16 bytes and each value was 4 KB, for a combined 4112 bytes. Since we used no compression, this database consumes approximately 87 GB of space out of the 93 GB available (due to the filesystem). It was important to nearly saturate the capacity of our device to ensure that every memory chip contained data and the maximum parallel performance could be obtained. In the experimental phase, we then performed a *readrandom* workload in *db_bench* to simulate the process of answering a large number of database fetch requests. We collected information from *db_bench* to understand the intensity of the workload on the database. However, we choose to measure the number of IO requests and bandwidth through Linux's */proc/diskstats* interface in order to accurately measure the performance from the device's perspective. To understand the energy characteristics of the system when using this workload, we again measured system power draw throughout the experiment with the Onset HOBO UX120-018 Data Logger [17]. In order to control for any variation in the system, storage device, or database performance, we performed five test runs and present the sample mean of the resulting data points.

4 EXPERIMENTAL RESULTS

First, we look at the latency cost of IO scheduling, then investigate the performance and energy efficiency effects of it using microbenchmarks. Finally, we compare these results to macrobenchmarks using a real database application.

4.1 Latency cost of scheduling

As storage device latencies become faster, the overhead cost of the system software to manage requests becomes a greater portion of the total request latency. Here we look at the time cost of scheduling by comparing access latencies using the four IO schedulers.

Table 1 lists the median read and write latencies of one million 4 KB requests (single process, QD=1) using the four IO schedulers. The *io_uring* API has optional features that can reduce latency and improve single-core performance, such as polling for completion (opposed to interrupts [23])

Table 1: Median latency (µs) for 4 KB requests

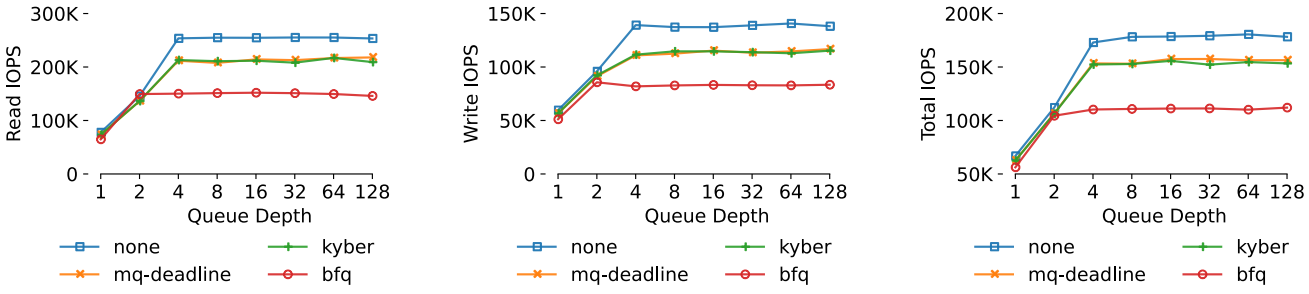
Scheduler	Read	*% diff.*	Write	*% diff.*
io_uring with defaults				
none	**12.52**	—	**16.55**	—
mq-deadline	13.39	*6.9%*	17.65	*6.6%*
kyber	13.44	*7.3%*	17.24	*4.2%*
bfq	15.01	*19.9%*	19.43	*17.4%*
io_uring with performance				
none	**7.81**	—	**12.19**	—
mq-deadline	8.32	*6.5%*	12.83	*5.3%*
kyber	8.33	*6.7%*	12.90	*5.8%*
bfq	9.41	*20.5%*	14.18	*16.3%*

and kernel-side submission queue polling (which reduces system calls [8]). The top half of Table 1 lists latencies submitted using *io_uring*'s default settings, while the bottom half were submitted using both these performance features, giving us the lowest latencies we observed overall. We tested both these default and performance settings for all our microbenchmarks, and found the trends across schedulers to be similar. Since these performance features are rather specialized optimizations, we present our microbenchmark results using the default settings as we believe this to be more commonly used and more broadly applicable.

The percentage differences in Table 1 are the additional latency costs added by choosing a particular scheduler compared to *none*. By choosing *mq-deadline* or *kyber* over the *none* scheduler, both reads and writes take roughly 6% longer for both configurations (an additional 0.5–0.9 µs). Choosing *bfq* can add as much as 2.5 µs, or roughly 20% to the median latency of requests. The use and choice of IO scheduler therefore adds a significant time cost to IO requests. Notice that the performance configuration reduces the average latency by roughly 5 µs in all cases, which is from removing latencies caused by other system mechanisms (system calls and interrupt handling). Even with the reduced latency of this performance configuration, the IO scheduler adds a significant time cost. This cost is expected to increase with larger IO depths, as there are more opportunities for the schedulers to perform reorganization. Next, we investigate if this cost yields any benefits in performance and energy efficiency.

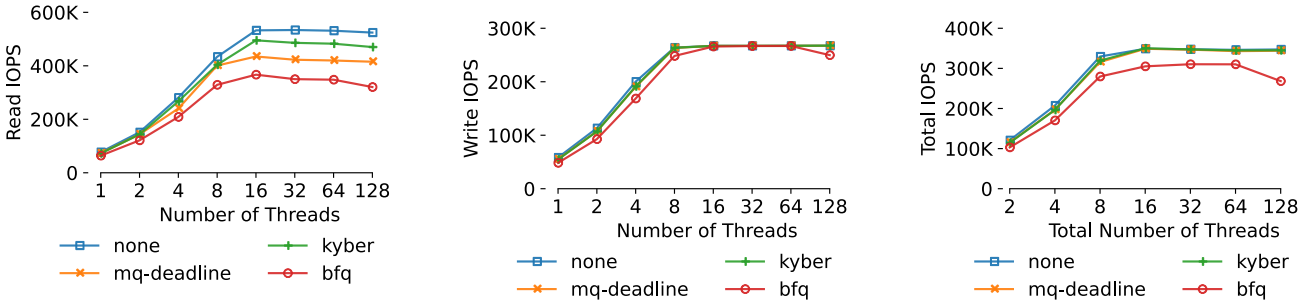
4.2 Microbenchmark Results

Figures 1 and 2 present the IO performance (IOPS) results for read, write, and mixed (50% read, 50% write) workloads in single- and multi-tenant scenarios, respectively. In the single-tenant experiments, a single-threaded application issues asynchronous IO requests from a single core at increasing IO queue depths, represented by the x-axis. In the multi-tenant experiments, the x-axis represents the number of



(a) Read **(b) Write** **(c) Mixed (50-50)**

Figure 1: Performance (IOPS), Single-Tenant



(a) Read **(b) Write** **(c) Mixed (50-50)**

Figure 2: Performance (IOPS), Multi-Tenant

tenants, where each tenant (process) issues one request at a time and can execute on separate cores.

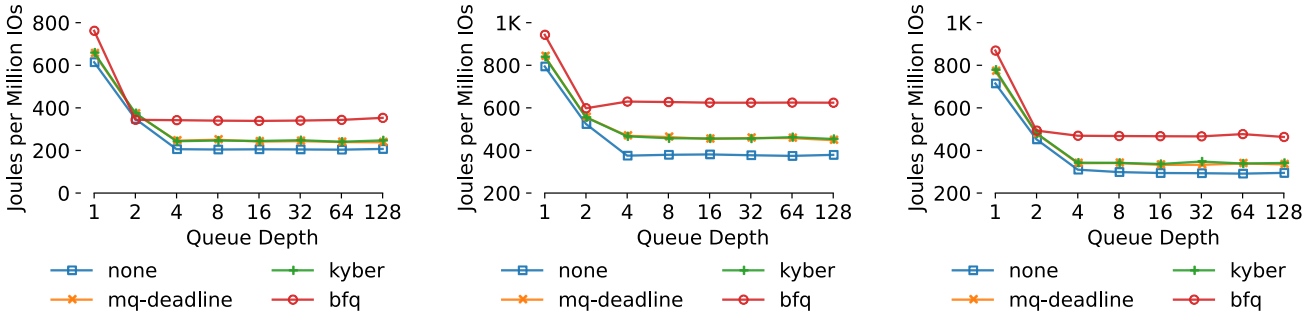
The microbenchmark results clearly indicate that **IO scheduling hurts performance more than helps.** For all cases shared in Figures 1 and 2, *none* yields either the best IO performance or ties with other schedulers. The results are similar for other IO performance metrics that we measured, including bandwidth, median latency, and tail latency, which we could not share in the paper due to limited space. For the 50% read and 50% write mixed workloads, we also measured the individual performance of read and write flows, and observed similar trends.

Next, we issue one million IO operations to the storage device, and measure the total energy consumption of the system for different schedulers. Figures 3 and 4 present the system's total energy consumption in joules per million IOs for increasing IO depths in single- and multi-tenant scenarios, respectively. Similar to the performance results, *none* also yields either the best energy efficiency, or ties with other schedulers. In other words, **IO scheduling hurts energy efficiency**. In various cases, *none* saves around 200 joules compared to *bfq* for every million IO operations completed. The main reason behind this energy saving is not because *none* causes significantly lower power consumption during the execution of the IO workload, but because *none* finishes the same workload much faster than other schedulers in many scenarios. In other words, *none* allows more IO operations to be completed per joule consumed. Finishing the same IO workload faster can eventually allow the system to stay in the idle state longer, or even transition into lower power states earlier as we discuss in Section 6.

We also ran microbenchmarks with mixed request sizes (4 KB and 8 KB) as well as different IO interfaces and read/write mixes, but again did not observe any benefits of IO scheduling in any of our performance or energy metrics. Similar to other cases, *none* yielded either the best result or tied with other schedulers.

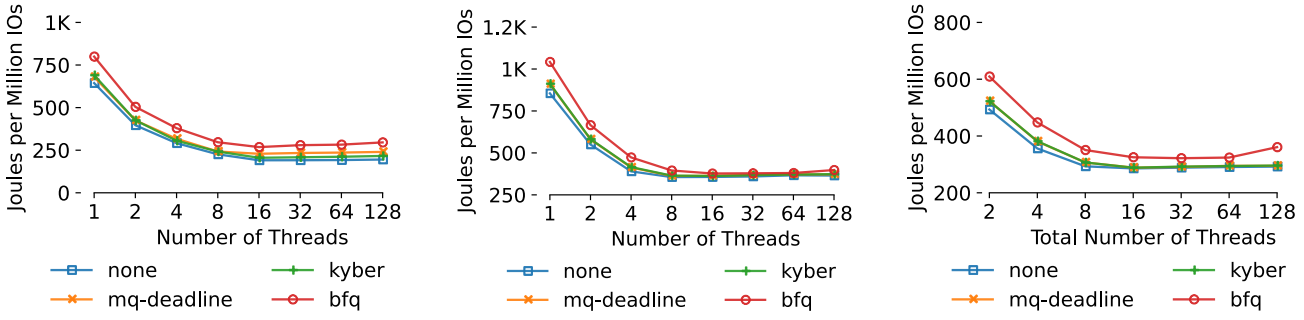
4.3 Macrobenchmark Results

Figure 5 illustrates the performance (5a) and energy consumption (5b) impact of IO schedulers using the RocksDB key-value store, where *db_bench*'s *randomread* key-value lookup workload is used as the macrobenchmark. Similar to microbenchmark results, these macrobenchmark results also indicate that IO scheduling does not provide any performance or energy benefit. With various IO intensities represented by IO depths on the *x*-axis, *none* yields the greatest performance (IOPS) and lowest energy consumption (joules per million IOs), or ties with other schedulers.



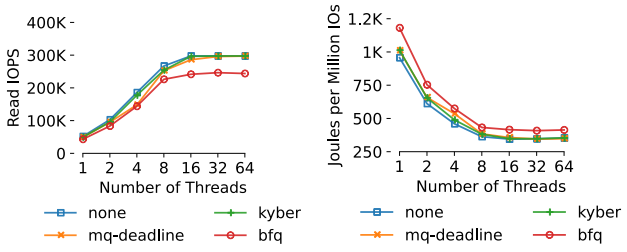
(a) Read (b) Write (c) Mixed (50-50)

Figure 3: Energy consumption, Single-Tenant



(a) Read (b) Write (c) Mixed (50-50)

Figure 4: Energy consumption, Multi-Tenant



(a) Performance **(b) Energy consumption**

Figure 5: RocksDB — *randomread*

5 RELATED WORK

Various IO schedulers have been proposed in the literature with different goals and characteristics [13, 19, 21, 22, 24]. Among them, two recent ones are Device-Direct Fair Queueing [21] and Multi-Queue Fair Queueing [13]. Device-Direct Fair Queueing (D2FQ) is specifically designed to reduce the software cost of IO scheduling in the block layer by making use of NVMe weighted round robin (WRR) arbitration, a method by which the device controller determines from which hardware submission queue to pull the next requests. Although WRR arbitration is standardized in the NVMe specification [5], it is an optional controller feature, and only round robin (RR) arbitration (without weighted priority queues) is required by the standard. This requirement of an additional hardware feature, which is not available on all NVMe storage devices, limits the usability of D2FQ. Multi-Queue Fair Queueing (MQFQ) [13], on the other hand, aims to provide fairness while removing inter-queue communication that causes synchronization slow-down.

In this paper, we focus on the performance and energy efficiency impact of the four IO schedulers included in the official Linux kernel: *none*, *mq-deadline*, *kyber*, and *bfq*. A previous work [16] examined using Optane SSDs as swap space for primary memory in mobile computing applications. They found that some real-time applications have potential promise for IO schedulers such as *bfq*, but found only limited evidence that IO scheduling could reduce latency in applications that relied on a ULL SSD for swapping. Although they looked at energy consumption, it was limited to measuring the energy consumption of RAM. The use of IO schedulers with more traditional SSDs has been investigated in the Linux user community as well [14], with no attention to energy efficiency or ultra low latency SSDs.

6 DISCUSSION AND FUTURE WORK

In energy efficiency analysis, it is insufficient to consider only the power consumption rate of the system when analyzing a particular software configuration. The performance should also be tied into the energy efficiency analysis, as finishing a task faster may allow the system to be idle earlier. For instance, during our experiments we observed 52 W power difference between full IO load and immediately after the workload is completed. Furthermore, while idle, machines can also switch to lower CPU power states depending on the idleness period, or can even hibernate. In addition, NVMe devices can support Autonomous Power State Transitions (APST), which similarly can reduce power consumption of the storage device when not under load. Finally, in cloud computing, energy efficiency is commonly achieved with energy-aware VM packing algorithms that minimize the number of active machines needed so that more machines can be transitioned into lower power states [12].

Our analysis in this work focuses on performance and energy efficiency of ULL storage device characteristics, specifically using Intel Optane. Although we recognize that IO schedulers still have value for slower devices such as HDDs, we believe that using IO schedulers for ULL devices is no longer needed to achieve performance and energy benefits. However, one important role of the operating system is to provide applications with fair and managed access to hardware. Further research is needed to analyze the role of schedulers for the purpose of fairness with ULL storage, and with ULL SSDs based on technologies other than Intel Optane.

7 CONCLUSION

As storage hardware grows faster, performance bottlenecks shift more and more towards system software, motivating operating system designers to reexamine traditional design decisions made for older, slower hardware. In this paper, we examined the performance and energy efficiency costs and benefits of IO schedulers for ultra-low latency (ULL) storage, specifically an Intel Optane SSD. The use of an IO scheduler with a ULL SSD adds significant latency to individual requests (Sec. 4.1) and it impairs throughput performance in all our microbenchmark workloads (Sec. 4.2) and macrobenchmark using RocksDB (Sec. 4.3). In addition, we measured the effect of scheduling on energy efficiency of the system, and found that scheduling impairs IO performance per energy consumed; therefore not using a scheduler (*none*) is more energy efficient. Please note that further investigation is needed for situations in which fairness or QoS is a priority, or while using ULL devices with significantly different characteristics than Intel Optane, such as flash-based ULL devices.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Danny Harnik, for their valuable feedback. This research was supported by the U.S. National Science Foundation (NSF) under grants OIA-1849213 and CNS-2050925.

REFERENCES

[1] 2016. mq-deadline multiqueue I/O scheduler. https://github.com/torvalds/linux/blob/master/block/mq-deadline.c.

[2] 2017. Kyber multiqueue I/O scheduler. https://patchwork.kernel.org/patch/9672023/.

[3] 2017. Samsung SZ985 Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[4] 2018. Product Brief: Intel Optane SSD DC P4800X Series. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-brief.pdf.

[5] 2019. NVM Express Base Specification, rev. 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[6] 2022. RocksDB Performance Benchmarking with db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[7] 2023. Multi-Queue Block IO Queueing Mechanism (blk-mq). https://docs.kernel.org/block/blk-mq.html.

[8] Jens Axboe. 2019. Efficient IO through io_uring. https://kernel.dk/io_uring.pdf.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13)*. ACM, New York, NY, USA, Article 22, 10 pages. https://doi.org/10.1145/2485732.2485740

[10] Budget Fair Queueing 2022. Budget Fair Queueing (BFQ) Storage-I/O Scheduler. http://algo.ing.unimo.it/people/paolo/disk_sched/.

[11] F. T. Hady, A. Foong, B. Veal, and D. Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (Sept 2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

[12] Logan Hall, Bryan Harris, Erica Tomes, and Nihat Altiparmak. 2017. Big Data Aware Virtual Machine Placement in Cloud Data Centers. In *4th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT 2017)* (Austin, Texas, USA) *(BDCAT '17)*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/3148055.3148057

[13] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 301–314. http://www.usenix.org/conference/atc19/presentation/hedayati-queue

[14] Michael Larabel. 2020. Linux 5.6 I/O Scheduler Benchmarks: None, Kyber, BFQ, MQ-Deadline. https://www.phoronix.com/review/linux-56-nvme.

[15] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 603–616. https://www.usenix.org/conference/atc19/presentation/lee-gyusun

[16] Geraldo F. Oliveira, Saugata Ghose, Juan Gómez-Luna, Amirali Boroumand, Alexis Savery, Sonny Rao, Salman Qazi, Gwendal Grignou, Rahul Thakur, Eric Shiu, and Onur Mutlu. 2021. Extending Memory Capacity in Consumer Devices with Emerging Non-Volatile Memory: An Experimental Study. *ArXiv* abs/2111.02325 (2021).

[17] Onset Computer Corporation 2017. *HOBO® Plug Load Logger (UX120-018) Manual*. Onset Computer Corporation. https://www.onsetcomp.com/sites/default/files/resources-documents/17838-E%20MAN-UX120-018.pdf

[18] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2021. Optimizing Storage Performance with Calibrated Interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 129–145. https://www.usenix.org/conference/osdi21/presentation/tai

[19] Toby J. Teorey and Tad B. Pinkerton. 1972. A Comparative Analysis of Disk Scheduling Policies. In *Communications of the ACM*. 177–184.

[20] Paolo Valente and Arianna Avanzini. 2015. Evolution of the BFQ Storage-I/O scheduler. In *2015 Mobile Systems Technologies Workshop (MST)*. IEEE, 15–20.

[21] 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 '21)*. USENIX Association, 403–415. https://www.usenix.org/conference/fast21/presentation/woo

[22] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. 1994. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS*. 241–252.

[23] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll is Better than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST '12)*. USENIX Association, USA, 3.

[24] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2015. Split-level I/O scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 474–489.