Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring

Zebin Ren
z.ren@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Animesh Trivedi
a.trivedi@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Abstract

Linux storage stack offers a variety of storage I/O stacks and APIs such as POSIX I/O, asynchronous I/O (libaio), high-performance asynchronous I/O (emerging `io_uring`) or SPDK, the last of which completely bypasses the kernel. Despite their availability, there has not been a systematic study of their performance and overheads. In order to aid our understanding, in this work we systematically characterize performance, scalability and microarchitectural properties of popular Linux I/O APIs on high-performance storage hardware (Intel Optane SSDs). Our characterization reveals that: (1) at low I/O loads, all APIs perform competitively with each other, with polling helping the performance by 1.7×, but consuming 2.3× CPU instructions; (2) at high-loads and scale, `io_uring` is more than an order of magnitude slower than SPDK; (3) at high-loads and scale, the benchmarking tool (fio) itself becomes a bottleneck; (4) state-of-practice Linux block I/O schedulers (BFQ, mq-deadline, and Kyber) introduce significant (up to 50%) overheads, and their use of global locks hinder their scalability. All artifacts from this work are available at https://github.com/atlarge-research/Performance-Characterization-Storage-Stacks.

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

Keywords: Linux storage stack, `io_uring`, SPDK, Efficiency, Measurements

ACM Reference Format:
Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23), May 8, 2023, Rome, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578353.3589545

CHEOPS '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0081-1/23/05.
https://doi.org/10.1145/3578353.3589545

1 Introduction

Modern storage devices such as Intel Optane SSDs can deliver millions of IOPS (I/O operations per second) with single-digit microseconds (*µsecs*) I/O access latencies [7, 17]. Meanwhile, the CPU performance has remained relatively stable as Moore's Law driven performance gains stall [29]. Consequently, the stalled CPU performance with high-performance storage hardware has exposed many previously hidden software overheads in the storage stack implementations, thus leading to a series of efforts to redesign and optimize the storage stack focusing on lock contentions, polling, copy elimination, new interfaces, scheduling, context switches, asynchronous I/O paths, interrupt and system call eliminations [3, 18, 20, 25, 30, 36, 37, 39, 40, 45, 56, 59, 66, 68].

Beyond these optimizations, there have been many efforts to improve the user-kernel and user-storage APIs and abstractions. Linux supports two popular and widely used APIs called (synchronous) POSIX file I/O calls [12, 13] and an asynchronous API called libaio [3]. Both of these APIs interact via system calls (syscalls) with the Linux kernel which can have high overheads [22, 38, 55]. More recently, Linux developers have introduced a new high-performance I/O API called `io_uring` [8]. It takes many established ideas from the high-performance networking domain (shared-memory queues, asynchronous I/O, polling, shared I/O contexts) and applies them to storage in a unified manner [61, 62]. These advancements are now merged in the Linux storage stack (since `v5.1` kernel version), and have shown to deliver high performance and CPU efficiency [22]. All of these APIs (POSIX, libaio, `io_uring`) work within the kernel.

The Linux kernel with its generic code execution, functionalities, and features can also introduce significant overheads [51], thus leading to the design of kernel-bypassing userspace storage stacks [24, 34, 69, 74]. The Storage Performance Development Kit (SPDK) is one of the most popular and widely used user space I/O libraries, which can deliver up to 10 million IOPS using a single CPU core [2]. However, user space I/O libraries lack many kernel-supported features such as fine-grained isolation, access control, file systems, multi tenancy, and QoS support [48, 64].

In summary, over the past decade, the in-kernel and userspace I/O stacks have undergone a significant development phase. Despite sharing a common functional goal

- *access to data from storage devices* - these I/O stack implementations prioritize different goals (performance, stability, security, efficiency). As a result, there is a lack of understanding about the performance, efficiency, and scalability of these stacks, specifically on modern high-performance storage devices such as Intel Optane SSDs. Closest to our work is the recently published work by Didona et al. that studies the performance and scalability of libaio, `io_uring`, and SPDK on flash SSDs [22]. We build on their findings and further report on performance breakdowns (microarchitectural, and instructions profiles), with I/O scheduler-related overheads.

In this paper, we take a step back and systematically study these storage stacks on modern high-performance storage devices. Our test bed consists of Intel Xeon CPUs with seven (7) Intel Optane devices in a single machine with the peak performance of 4.2 million IOPS (specification: 600 kIOPS × 7 devices) (Table 1). We start our investigation by studying the performance (expressed as IOPS) of POSIX I/O, libaio, `io_uring`, and SPDK at a low load with a single outstanding I/O request on a single Optane device and a single CPU core, and quantify in detail their microarchitectural properties and CPU instructions breakdown. We then quantify their performance as we increase the number of outstanding requests (expressed as queue depth), and add CPU cores and NVMe devices (from 1 to 7). Lastly, we measure the impact of three state-of-the-practice block I/O schedulers (BFQ, mq-deadline, Kyber) on the performance of these stacks [10]. The primary contribution of this work is about performance and scalability characterization of state-of-the-practice Linux storage APIs. Our key findings demonstrate that:

1. At low I/O loads, we report 81.1-94.6 kIOPS for non-polling APIs (POSIX, libaio, and `io_uring`). Polling increases the performance to 108-138.9 kIOPS (1.7× increase) with `io_uring` and SPDK, with a proportional increase in the CPU instructions required per I/O operation (up to 2.3×). (§3.1).
2. At high-loads (128 queue depth, 7 devices), `io_uring` is more than an order of magnitude inefficient than SPDK. SPDK can saturate our hardware with 5 cores (using fio) or just a single core when using SPDK's light-weight perf benchmark. In contrast, the best performing `io_uring` configuration needs 13 cores. (§3.2).
3. At high-loads, the benchmarking tool (fio) itself becomes a bottleneck. In our setup, SPDK with fio takes 5 cores to deliver 4.2 Million IOPS. SPDK with its own perf benchmark [15] takes only one, thus showing 5× overheads with fio. (§3.2).
4. State-of-practice Linux I/O schedulers (BFQ, mq-deadline, and Kyber) introduce significant (up to 50%) overheads, and their use of global locks hinder their scalability. (§3.3).

To facilitate reproduction, our benchmarking code is available at https://github.com/atlarge-research/Performance-Characterization-Storage-Stacks.

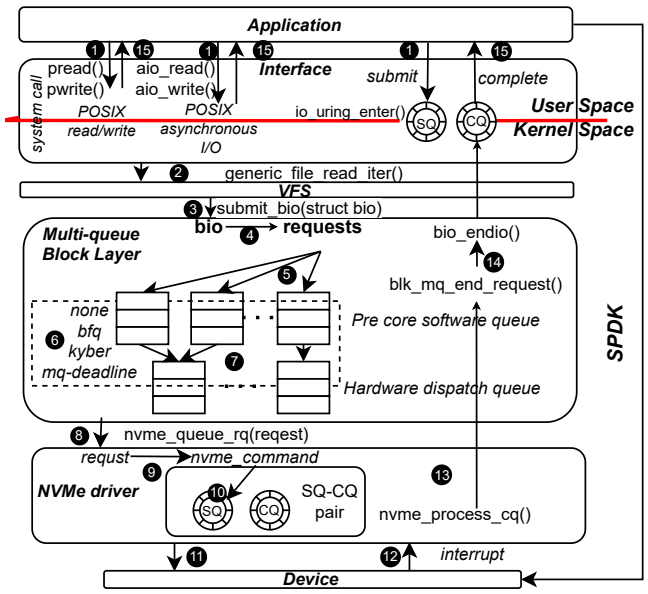


Figure 1. A high-level overview of the Linux storage stack.

2 Background

Before we start with benchmarking, we briefly recap the key concepts and details from the Linux storage stack (Figure 1).

2.1 Linux I/O Stack

Applications submit I/O requests to the I/O interface (step 1), such as POSIX read/write [12, 13] (referred to as psync), libaio [3], `io_uring` [8]. The I/O interface constructs `struct kiocb` and submits it to a file system via the VFS (step 2). The file system then resolves the relevant block address containing data, constructs a `struct bio` request and submits it to the block layer through `submit_bio()` (step 3). The Linux block layer converts `struct bio` to `struct request` (step 4) and puts it in a per-core software queue (step 5). These requests are then processed by a block I/O scheduler and put in a hardware dispatch queue (step 6-7). Requests in the hardware dispatch queue are processed by the NVMe driver by `nvme_queue_rq()` (step 8). The NVMe driver constructs an NVMe command according to the requests (step 9), and writes them to the submission queue (SQ) (step 10) to be processed by the device (step 11). After the command is processed by the device, the device writes the request to the completion queue (CQ) and generates an interrupt (step 12). The completion is processed by `nvme_process_cq` and then `blk_mq_end_requeset()` and `bio_end_io()` are called to finalize the request in `bio` layer (step 14). At last, the application is notified of the completion results (step 15).

2.2 Emerging High-Performance io_uring API

io_uring maintains shared-memory, lock-free submission (SQ) and completion queues (CQ) between the kernel and

Table 1. Specification of the server machine.

CPU	Dual socket (2×) Intel(R) Xeon(R) Silver 4210R CPU 10 cores @ 2.40GHz, Hyper-threading disabled, Turbo disabled
Memory	256GB, DDR4
Storage	7× Intel Corporation Optane SSD 900P Series (512 bytes LBA, preconditioned 10×, spec: 550 KIOPS with 4KiB random read), all 7 NVMe SSDs connected to the CPU NUMA domain 1
Software	Ubuntu 22 with Linux kernel `v5.15.79`, built with default config, fio-3.32 (commit db7fc8d), SPDK 22.09 (commit aed4ece93)

application. The application writes to the SQ to submit I/O request and gets completion notficiation via the CQ. By default (identified as *iou*), when an application submits new requests to the SQ, it notifies the kernel by `io_uring_enter` syscall. The same syscall is used to retrieve completion events from the CQ. `io_uring` can use polling to eliminate syscalls and interrupts. The application can poll on the CQ, thus actively reaping completions without interrupts. If there are completions available on the CQ, these can be processed without a syscall, otherwise a syscall is issued to poll in the kernel for I/O completion. This mode is referred to as *completion polling* or *iou-c* (using `hipri` flag). To remove a syscall during SQ submission, the application can start a kernel thread to poll for I/O requests on the SQ on behalf of the application (using `sqthread_poll`), thus eliminating a need for a syscall. This mode is referred to as *submission polling* (*iou-s*, with the `hipri` flag).

2.3 Kernel-bypassing Userspace SPDK API

Storage Performance Development Kit (SPDK) is a user space I/O library that provides zero-copy, high-performance, and efficient direct access to NVMe SSDs from the userspace leveraging a poll-based NVMe driver [69]. SPDK initializes and accesses NVMe I/O queues directly in the userspace, thus completely skipping the kernel during I/O operations. Currently, SPDK is considered the state-of-the-art I/O stack that can deliver the best performance to workloads and is used extensively [31, 38, 41, 67, 71].

3 Experiments

We now start with our study of the performance and efficiency of the Linux storage stack with its I/O APIs: POSIX I/O (psync), libaio (aio), `io_uring` (with 3 configurations: default as iou, completion polling iou-c, and submission with completion polling as iou-s), and SPDK. The purpose of our study is to answer:

(Q1) *What is the performance gap among different I/O APIs and their configurations?*

(Q2) *Why is there a performance gap, and how do these APIs use the CPU time, cycle, and instructions?*

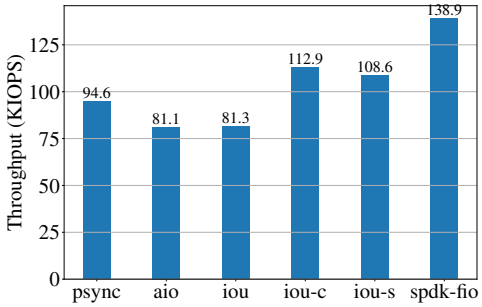


Figure 2. Random read 4KiB throughput (IOPS).

(Q3) *How does the performance gap scale with the number of CPU cores and storage devices?*

(Q4) *What is the impact of I/O schedulers on the performance of these I/O APIs?*

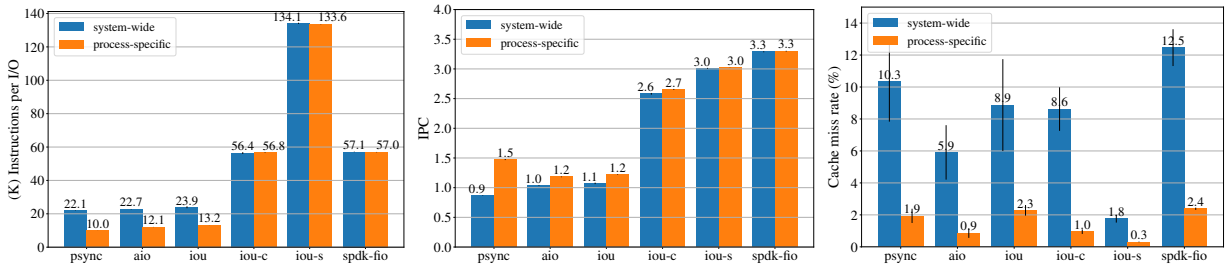
We cover **Q1-3** in §3.1 and §3.2, whereas §3.3 answers **Q4**.

To prepare for the benchmarking, we follow the best practices from [65] for Optane SSDs. All devices are formatted with the LBA size of 512 bytes, and are written completely 10 times. For our experiments we use the request size of 4KiB which shows the same performance as 512B requests. We choose fio as the workload generator [6]. All benchmarking processes are pinned to the CPU NUMA node 1 except when we use more than 10 cores (uses CPU/NUMA node 0). We run fio workloads for 140 seconds (2 minutes + 20 seconds warm up time). We use `perf record` to record microarchitectural event metrics from the CPU counters, and report an average of 10 times over 5 seconds runs while the fio workloads run. Table 1 presents the benchmarking environment.

3.1 The Efficiency of I/O Stacks

We start our benchmarking with a single NVMe Optane device with a single CPU core, except for iou-s that uses 2 cores. The reason for using 2 cores is the performance collapse due to the contention of fio and kernel threads polling on a single physical CPU core [22]. We collectively refer to iou-c, iou-s, and SPDK as *polling* libraries, and psync, libaio, and iou to as *non-polling* libraries.

3.1.1 Performance. We report on the maximum IOPS completed with 4KiB random reads with a single outstanding request (i.e., queue depth QD=1). Figure 2 shows our results with IOPS (y-axis, higher is better) and libraries on the x-axis. There are a few observations here. First, iou and aio have the worst performance compared with other libraries, 17% to 72% lower IOPS than psync and SPDK, respectively. psync, the oldest among the six configurations, has 17% higher IOPS compared with aio and iou. All non-polling libraries perform relatively close by 81.3-94.6 KIOPS. The use of polling increases the performance significantly. iou-s and iou-c (both polling based) have 34% and 39% higher IOPS than the non-polling iou configuration. SPDK has the highest IOPS, 71.3%



(a) Instructions per I/O operation. **(b)** Instructions per cycle (IPC). **(c)** Cache miss rates.

Figure 3. Comparison on micro-architecture efficiency of 6 different I/O libraries using a single thread and queue depth = 1.

higher than iou. In general, I/O APIs with polling improve the performance up to a margin of 1.7× (81.3 vs 138.9 KIOPS).

3.1.2 Microarchitectural characterization.

As a next step, we characterize their CPU profiles focusing specifically on: (i) instructions required to complete a 4KiB random read operation (capturing the software overhead), lower is better; (ii) code execution efficiency by calculating instruction retired per CPU cycle (IPC), higher is better; and (iii) data fetching efficiency of the code path by calculating the cache miss rates, lower is better. We calculate these quantities for two configurations, system-wide (`-a` flag with perf) and process-specific. The process-specific configuration only reports events that are attributed to the fio process, whereas the system-wide configuration captures all events happening in the system. Since fio is the only workload running on the idle server, all events are directly (process-specific) or indirectly (system-wide) attributed to fio. Nonetheless, we do report them separately and explain the gap between them when they diverge.

We show our results in figure 3. There are three primary observations here. First, non-polling libraries (psync, aio, and iou) all have equal or better instructions required per I/O operation than the polling libraries (iou-c, iou-s, and SPDK) as shown in figure 3a. This result is expected as polling libraries are constantly utilizing the CPU core to 100%, thus increasing the number of instructions executed. Consequently, for similar performance (IOPS), they show higher (worse) instructions/IOPS. iou-s takes 2.3× more instructions to process each I/O request than iou-s and spdk-fio because it runs on the two cores. Note that there is 2× difference between system-wide and process-specific instruction/IOPS for non-polling libraries. It is because when the CPU utilization is not 100%, the `swapper` process takes up half the instructions in a system-wide profile. The `swapper` process (PID 0) is a special process in the Linux kernel that is run by a CPU when there is no other processes to run (i.e., the CPU is idle). Hence, at low-load (QD=1), polling-based libraries need 5-12× more instructions to complete an I/O request than their non-polling counterparts.

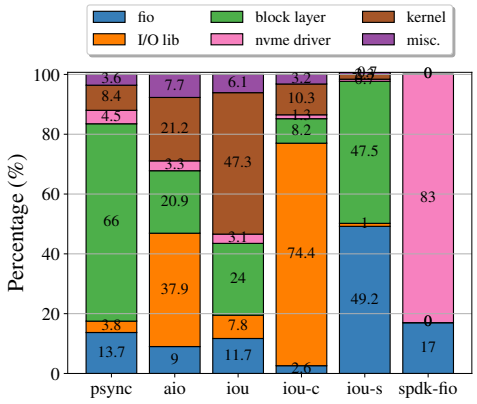
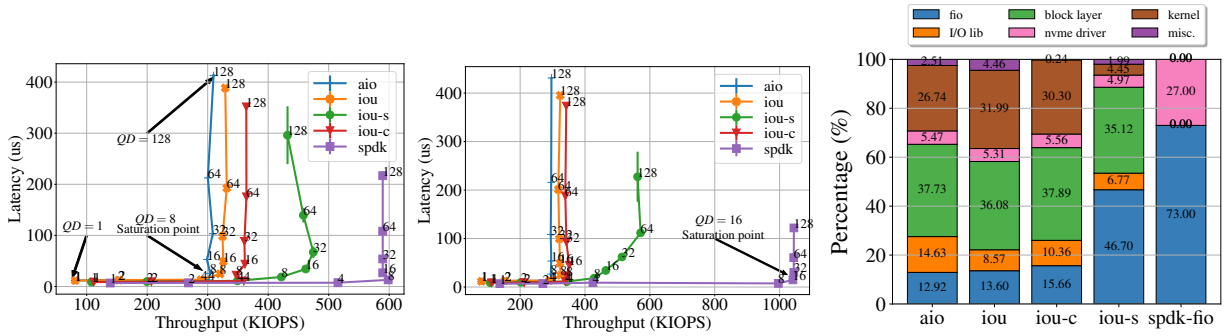


Figure 4. Instruction breakdown by %.

Second, with a streamlined execution of instructions for polling (an optimized code path), the polling libraries show better IPC than the non-polling libraries (figure 3b). A higher IPC implies the instructions are better structured so that they are executed with each CPU clock cycle without being stalled, hence, showing an efficient code path. The three non-polling libraries have lower IPC, about a one-third ($1/3^{rd}$) compared with the polling-based libraries on average. However, the higher IPC of the polling-based libraries does not mean they are more efficient as the CPU spends many instructions on polling. A polling code is typically well structured and optimized, leading to a higher IPC for the polling-based libraries. IPC becomes comparable when the workload increases (§3.2).

Lastly, figure 3c presents the cache miss rates (lower is better) that represent the data fetching efficiency of the code path. Even though psync has 1.6-2.5× higher cache miss rates, it still delivers higher IOPS than its non-polling counterparts. SPDK has the highest cache miss rate, 14.1× higher than the lowest one, iou-s and 1.5× higher compared with the second highest one, psync. With SPDK, all DMA operations are done directly to user buffers, hence, all CPU initiated requests result in a mandatory cache miss. We do not observe a direct impact of such cache miss rates on the performance. We hypothesize that this is due to the small memory footprint of



(a) One Optane device. **(b)** 7 Optane devices. **(c)** Instruction breakdown by %.

Figure 5. Throughput and latency with queue depth ranging 1-128, and the instruction breakdown at the saturation point.

the data request sizes involved. However, in our experiments, we notice that the process scheduler has a huge impact on the cache miss rate. When the threads are not pinned in a single node and the user and kernel threads of iou-s are mapped to different nodes, the cache miss rate is higher than 95% (not shown).

3.1.3 Instruction breakdown. As the last step in our analysis, we break down the different components where the CPU has spent time by executing instructions. Figure 4 shows our results in five distinct categories: (i) fio, the workload; (ii) I/O library specific code in the user or kernel space; (iii) block layer specific routines; (iv) NVMe device driver; and lastly (v) kernel-specific symbols (scheduling, memory allocation, etc.). The rest is marked misc. In an ideal setting, all CPU instructions (100%) are to be consumed by the fio, thus showing a zero-cost ideal I/O API, however, such a system is unrealistic. psync spends a significant amount of instructions (66%) in the block layer, whereas its own psync-specific code only requires 3.8%. In contrast, libaio spends nearly 40% of its instruction budget on its own implementation [21]. The default iou is somewhere in between, however, as polling is introduced, iou-c spends the majority of its instructions with completion polling (triggered via `io_do_iopoll` in the kernel, attributed to I/O lib). iou-s evenly splits between fio-based polling for completion (`fio_ioring_getevents`) and block-layer kernel thread polling as expected. Lastly, SPDK spends 83% of its instructions on polling as attributed to the NVMe driver (a part of the SPDK) with 17% instructions left for the fio benchmark.

3.1.4 Summary. At low I/O loads, non-polling and polling APIs perform competitively with each other, respectively. We report 81.1-94.6 KIOPS for non-polling APIs. Polling increases the performance to 108-138.9 KIOPS (1.7× increase) with `io_uring` and SPDK. However, the polling-based APIs take more CPU instructions per IOPS (up to 2.3×). The CPU instruction breakdown shows non-polling APIs spend a significant fraction of their instruction budget on non-fio code.

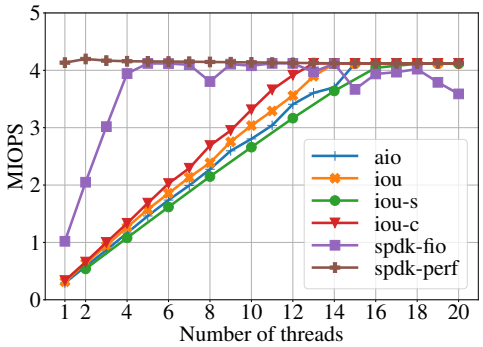
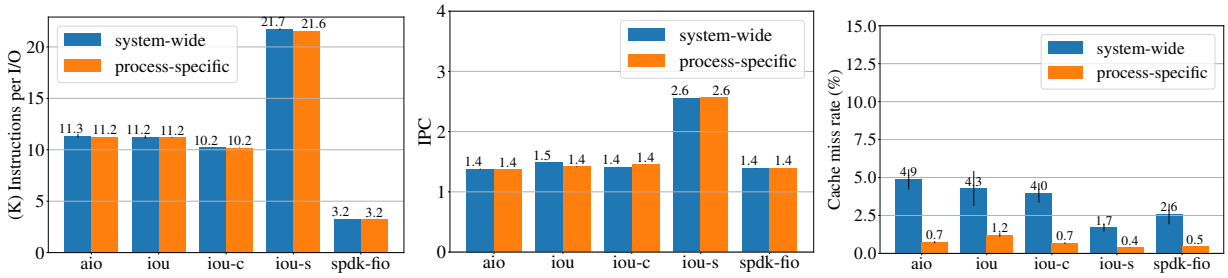


Figure 6. Multi-core scalability of different I/O libraries.

3.2 Scaling: I/O Depths and NVMe Devices

In the previous section, none of the configurations reached the single Optane hardware limit because the queue depth was the bottleneck (QD=1). In this subsection, we study the scalability properties (**Q3**) of these I/O APIs as we scale the load on the system (QD > 1) and increase the number of devices from 1 to 7. Due to its synchronous nature, we drop psync from the consideration here.

3.2.1 Performance. Figures 5a and 5b show our results with a *single CPU core*. The x-axis shows IOPS, and the y-axis the corresponding latency as we increase the queue depth by issuing 1-to-128 requests. A flat horizontal line is preferred. We observe that Optane devices can deliver more performance (600 KIOPS) than the peak throughput specification from the Intel spec sheet (550 KIOPS). With 7 devices, a single CPU core with SPDK can deliver close to 1.1 million IOPS (@7.4μsecs). All lines in the graph form a J shape line. At the start, throughput increases with a negligible increase in latency, then it stops increasing, and the latency increases dramatically as the CPU becomes saturated and it cannot keep up with the load. We call this point the *saturation point* (the heel of the J curve). aio, iou, iou-s, iou-c, and spdk-fio reach the saturation point when QD = 8, 8, 32, 8, 8 with a



(a) Instructions per I/O operation. **(b)** Instructions per cycle (IPC). **(c)** Cache miss rates.

Figure 7. Comparison on micro-architecture efficiency of 5 different I/O libraries at saturation point

single device, respectively. The same queue depth is needed with 7 devices except for spdk-fio, which needs QD=16. In comparison, the other I/O libraries only deliver 0.79× (474K vs. 600K IOPS for one device) to 0.53× (549K vs. 1.1 million IOPS, for 7 devices), a fraction of the SPDK's performance.

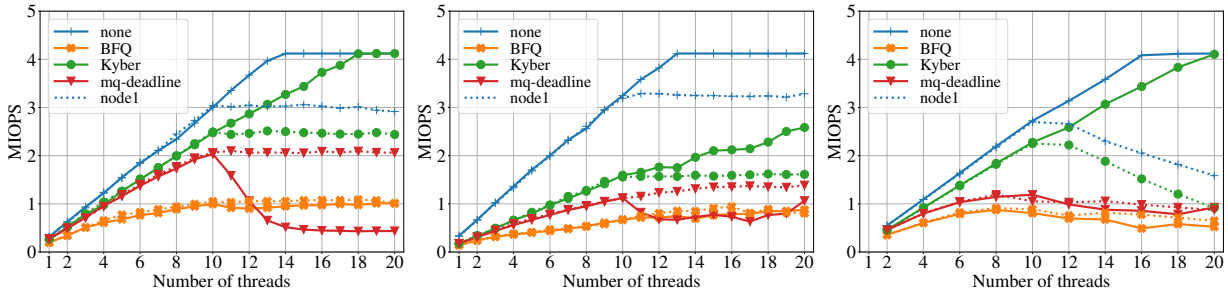
To showcase the scalability of performance, we now add more CPU cores at the peak performance with QD=128 with 7 Optane devices. Figure 6 shows our results. The y-axis is the throughput in million of IOPS, and the x-axis shows the number of cores. The figure shows that all I/O stacks can reach the peak hardware performance, however, with different CPU cores. The most efficient one among them is SPDK which only takes 5 CPU cores to deliver the peak 4.2 million IOPS. In comparison, the best `io_uring` variant (iou-c) takes 13 cores (2.6× inefficient).

At this configuration, we notice that the fio benchmarking code itself becomes a bottleneck. To verify this hypothesis, we use SPDK's own benchmarking framework called SPDK-perf [5, 16]. With this, SPDK is able to deliver the peak performance of 4.2 million IOPS even with just a single core! In this configuration, SPDK-perf is 13× more efficient than iou-c with fio. From our initial analysis, SPDK-perf is able to outperform fio because of its simple setup as it runs a set of independent threads that only send NVMe requests and poll them for completion. Notably, each thread resides on an independent core and does not coordinate with the other threads for job accounting and scheduling, runs minimal checks, and allocates DMA buffers directly with a simple memory pool without touching the data. SPDK-perf represents an unrealistic workload. However, it is useful in establishing the peak performance bounds.

3.2.2 Microarchitectural characterization. We now study the microarchitectural properties of the benchmark at the saturation point with 1 CPU core and 7 Optane devices (at the knee of the J curve in figure 5b). Figure 7a shows the instructions needed per IOPS. Notice the significant decrease in the number of instructions needed for 4KiB I/O from figure 3a (they have different y-axes). SPDK's overheads have decreased by 17.9× (needing only 3.2 thousand instructions/4KiB), whereas for non-polling libraries it only decreases by 2×. Hence, at the saturation point, the polling-based libraries become more efficient than the non-polling ones. All libraries illustrate a similar IPC performance (figure 7b), thus hinting that the code complexity is similar across all libraries, but their code lengths differ, hence larger numbers of instructions/IOPS. Both non-polling and polling-based libraries have the same IPC at the saturation point, effectively amortizing the cost of polling. iou-s stands out as an anomaly as it uses 2 CPU cores. In comparison to figure 3c, the cache miss profile (figure 7c) shows improvements across the spectrum under load.

3.2.3 Instruction breakdown. Lastly, we now discuss the CPU instruction profile with single core and 7 devices. In comparison to figure 4 as the load on the system increases, SPDK spends fewer instructions on I/O processing (amortization, mostly on NVMe processing) and more on the benchmark fio (getting close to the ideal I/O stack). As we have demonstrated previously, eventually fio itself becomes a bottleneck. We further notice that all in-kernel stacks spend approx. 35% of their instruction budget on the block layer, which does not decrease or amortize beyond a point. In comparison with iou and iou-c, the kernel-related overheads are decreased in iou-s.

3.2.4 Summary. All I/O stacks scale to the peak hardware performance, but with significantly different CPU costs (**Q3**). `io_uring` is more than an order of magnitude inefficient than SPDK. SPDK-fio can saturate our hardware with 5 cores or just a single core when using SPDK-perf. In contrast, the best-performing `io_uring` configuration needs 13 cores. The microarchitectural profile indicates a similar level of code complexity (IPC) and data coverage (cache behavior), but with significantly longer code paths (instructions/IOPS) among the stacks. Furthermore, at high-loads, the benchmarking tool (fio) becomes a bottleneck as shown by SPDK-fio delivering 1.1 million IOPS and SPDK-perf delivering 4.2 million IOPS with a single core (4×).



(a) default `io_uring` (iou). **(b)** with completion polling (iou-c). **(c)** with submission polling (iou-s).

Figure 8. Comparison on the effect of Linux kernel I/O scheduler with increasing number of threads at queue depth = 128.

Table 2. Overheads from different block I/O schedulers.

	none	BFQ	Kyber	mq-deadline
iou	320.0 (1.0)	195.8 (0.61×)	264.6 (0.83×)	280.5 (0.86×)
iou-s	565.7 (1.0)	357.8 (0.63×)	470.1 (0.83×)	474.5 (0.84×)
iou-c	342.0 (1.0)	142.3 (0.43×)	175.3 (0.51×)	181.6 (0.53×)

3.3 Impact of I/O Schedulers

In this section, we investigate the overheads from three state-of-the-practice Linux block I/O schedulers [10], BFQ [4], Kyber [9], mq-deadline [11] **(Q4)**. In this evaluation, we quantify overheads of I/O schedulers by comparing their IOPS performance with a none scheduler, which is a no-operation (no-op) scheduler. All schedulers are configured with their default parameters.

Originally designed for disks, BFQ (Budget Fair Queueing) is a proportional-share I/O scheduler that guarantees each application (using heuristics) a desired fraction of the storage throughput by assigning a budget to weighted queues (measured in sectors) [63]. Kyber is a self-regulating I/O scheduler that separates reads (synchronous) and writes (asynchronous) operations in separate queues, and only dispatches I/O requests to the hardware dispatch queue when it can maintain a certain target latency (configurable) [32]. Kyber prioritizes reads over writes. Lastly, just like Kyber, the mq-deadline scheduler also separates writes and reads into separate queues, but it also sorts them in an increasing logical block order. The sorting allows it to extract I/O merging opportunities. The I/O is issued in batches from the queues prioritizing reads. The past performance of read and write queues determines how priority to read over write is carried forward to the next round of batch I/O requests [1]. A design of a fair and proportionally-shared I/O scheduler for high-speed NVMe devices is a field of active research [27, 33, 35, 49, 53, 60, 64].

Table 2 presents the throughput of these schedulers with `io_uring` configurations with a single core and 7 devices at the saturation point. We show the raw numbers and show the relative slowdown against the none scheduler in the parentheses. We report that all the I/O schedulers hurt the throughput. BFQ has the worst performance among the three schedulers, it delivers 0.43-0.63× of the none throughput. Kyber and mq-deadline are better than the BFQ, but these two schedulers still reduce the throughput by 14% to 47%. iou-c is more sensitive to I/O schedulers, losing 20-30% more throughput compared with iou and iou-s.

Figure 8 presents the throughput scaling of four schedulers with an increasing number of threads with QD=128. Keep in mind, iou-s takes 2× the number of cores, hence, it is only run till 1-10 fio threads, which effectively takes additional 1-10 kernel polling threads to get to 20 threads (solid lines). We report that all three schedulers hurt the throughput with Kyber having the least amount of overheads, thus the highest throughput. mq-deadline has a better throughput than BFQ before running on the remote NUMA node (more than 10 threads). In this setting, the throughput of the mq-deadline drops as it contends on a lock for a global variable in `blk_mq_hw_ctx`. This lock contributes to 79.6%, 28.7%, and 48.83% of the total CPU cycles for iou, ios-s, and iou-c with 16, 16, and 12 threads, respectively. Kyber does not use a global lock, which leads to a linear scalability of the throughput.

In order to limit the NUMA impact, we also report on a configuration (shown as the dotted line) where we restrict fio threads (1-20 threads) to the NUMA node 1. In this case, the mq-deadline does not deteriorate, while none and Kyber do not reach the hardware limits (figure 8a). For iou-s (with the kernel threads polling, whose CPU affinity we do not control), the NUMA restriction significantly hurts the performance scalability of none and Kyber schedulers.

Summary: All three state-of-the-practice block I/O schedulers have non-negligible performance overheads (14-57%). Apart from that, they all suffer from scalability issues on modern multi-socket NUMA machines. We plan to further investigate the reason for this.

4 Related Work

Linux storage stacks have undergone significant changes over the past decade. These changes include using multi-queue to avoid lock contention [20], optimizing the bottom-half interrupt handler [54], introducing asynchronous to the I/O path to hide latency [40], polling [18, 39, 70], eliminating copying [36, 56], designing light-weight storage layer [40], reducing interrupt and syscall costs [45, 59]. Yang et al. [68] compare the performance between polling and interrupts for low-latency SSDs. Xu et al. [66] analyze the performance of NVMe devices and their impact on data-intensive workloads like databases. Koh et al. [37] analyze the system challenge with ultra-low latency SSD. I/O schedulers have been studied in the past where their designs focus on fair sharing or QoS (quality of service) in a multi-tenant environment [28, 42, 48, 64]. However, even state-of-the-art I/O schedulers like D2FQ still harm performance with big access size (8KiB) and parallel access (6 threads) [64]. Userspace I/O stacks like SPDK [14, 69] are also used extensively as they allow to bypass kernel-related overheads (including schedulers) [24, 34]. Apart from optimizing the storage stack, there have been efforts to specialize it by using application-specialized codes such as using eBPF codes for caching or bypassing expensive I/O stacks for data processing [19, 43, 72, 73]. This specialization allows for efficiency by shortening the code path. Accelerators have also been used to execute these specialized codes for data processing [57, 58].

Closest to our work is the recently published work by Didona et al. that studies the performance and scalability of libaio, `io_uring`, and SPDK. We build on their findings and further report on: (i) analysis with Intel Optane SSD devices that are known to show different performance characteristics than NAND flash SSDs [65]; (ii) detailed performance breakdowns with microarchitectural, and instructions profiles; and (iii) I/O request scheduling related overheads with the three Linux block I/O schedulers.

5 Conclusion and Future Work

In this work, we systematically study the performance and scalability of the Linux I/O stack with four different I/O APIs: POSIX I/O, asynchronous libaio, `io_uring`, and userspace SPDK stacks. Our findings reveal that SPDK is still the defacto winner in terms of raw performance and I/O efficiency (instructions/operation). Given enough CPU cores, the performance of `io_uring` does scale and matches the performance of SPDK, but consequently loses in terms of I/O efficiency. The Linux I/O scheduler can introduce significant overheads. We also report on the detailed instruction breakdown how the CPU instructions are used across different I/O stack components. Our results summarize that the Linux storage stack still has significant performance overheads, and scalability challenges in the presence of high-performance, low-latency NVMe devices. As a next step, we are expanding our study to include file systems (ext4, F2FS, XFS), complex workloads (databases [26], key-value stores [23], machine learning training [47], bioinformatics [44], graph processing [50, 52]), mixed read-write workloads, and mixed priority tenants with/without a disaggregated setting [46].

Acknowledgments

This work is funded by The Dutch Research Council (NWO) grant number OCENW.KLEIN.561. The authors would like to thank Jesse Donkervliet, Sacheendra Talluri, Krijn Doekemeijer, and Nick Tehrany for their help with the paper.

References

[1] 2023. Chapter 12. Setting the disk scheduler. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/setting-the-disk-scheduler_monitoring-and-managing-system-status-and-performance. Accessed: 31-03-2023.

[2] Accessed: 2023-01-26. *10.39M Storage I/O Per Second From One Thread.* https://spdk.io/news/2019/05/06/nvme/

[3] Accessed: 2023-01-26. *aio.* https://man7.org/linux/man-pages/man7/aio.7.html

[4] Accessed: 2023-01-26. BFQ Budget Fair Queueing Document. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html

[5] Accessed: 2023-01-26. Evaluate Performance for Storage Performance Development Kit (SPDK)-based NVMe* SSDs. https://www.intel.com/content/www/us/en/developer/articles/technical/evaluate-performance-for-storage-performance-development-kit-spdk-based-nvme-ssd.html

[6] Accessed: 2023-01-26. *fio.* https://github.com/axboe/fio

[7] Accessed: 2023-01-26. 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: 2023-01-26. *io_uring.* https://man.archlinux.org/man/io_uring.7.en

[9] Accessed: 2023-01-26. Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/

[10] Accessed: 2023-01-26. Linux I/O schedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers

[11] Accessed: 2023-01-26. Multi-Queue Block IO Queueing Mechanism (blk-mq). https://www.kernel.org/doc/html/latest/block/blk-mq.html

[12] Accessed: 2023-01-26. *pread.* https://man7.org/linux/man-pages/man2/pread.2.html

[13] Accessed: 2023-01-26. *pwrite.* https://man7.org/linux/man-pages/man3/pwrite.3p.html

[14] Accessed: 2023-01-26. *SPDK.* https://spdk.io/

[15] Accessed: 2023-01-26. *SPDK NVMe perf tool.* https://github.com/spdk/spdk/blob/master/examples/nvme/perf/perf.c

[16] Accessed: 2023-01-26. SPDK perf code. https://github.com/spdk/spdk/tree/master/examples/nvme/perf

[17] Accessed: 2023-01-26. *Toshiba Memory Introduces XL-FLASH Storage Class Memory Solution.* https://americas.kioxia.com/en-us/business/news/2019/memory-20190805-1.html

[18] Jens Axboe. Accessed: 2023-01-26. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf

[19] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 121–134. https://www.usenix.org/conference/atc19/presentation/bijlani

[20] Matias Bjørling, Jens Axboe, David W. Nellans, and Philippe Bonnet. 2013. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013*, Ronen I. Kat, Mary Baker, and Sivan Toledo (Eds.). ACM, 22:1–22:10. https://doi.org/10.1145/2485732.2485740

[21] Jonathan Corbet. Accessed: 2023-01-26. *Fixing asynchronous I/O, again.* https://lwn.net/Articles/671649/

[22] 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, Haifa, Israel, June 13 - 15, 2022*, Michal Malka, Hillel Kolodner, Frank Bellosa, and Moshe Gabel (Eds.). ACM, 120–127. https://doi.org/10.1145/3534056.3534945

[23] Krijn Doekemeijer and Animesh Trivedi. 2022. Key-Value Stores on Flash Storage Devices: A Survey. https://doi.org/10.48550/ARXIV.2205.07975

[24] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. 2021. BDUS: implementing block devices in user space. In *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz (Eds.). ACM, 8:1–8:11. https://doi.org/10.1145/3456727.3463768

[25] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. Analyzing, Modeling, and Provisioning QoS for NVMe SSDs. In *11th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2018, Zurich, Switzerland, December 17-20, 2018*, Alan Sill and Josef Spillner (Eds.). IEEE Computer Society, 247–256. https://doi.org/10.1109/UCC.2018.00033

[26] Sergej Hardock. 2020. *Flash-aware Database Management Systems.* Ph. D. Dissertation. Technical University of Darmstadt, Germany. http://tuprints.ulb.tu-darmstadt.de/14476/

[27] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queueing. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 301–314.

[28] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 301–314. https://www.usenix.org/conference/atc19/presentation/hedayati-queue

[29] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60. https://doi.org/10.1145/3282307

[30] 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, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang

[31] Intel®. Accessed:2022-04-26. SPDK In The News. https://spdk.io/news/.

[32] Jonathan Corbet. 2017. Two new block I/O schedulers for 4.12. https://lwn.net/Articles/720675/. Accessed: 31-03-2023.

[33] 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 *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 289–300.

[34] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*, Nitin Agrawal and Sam H. Noh (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[35] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2009. Disk Schedulers for Solid State Drivers. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (Grenoble, France) *(EMSOFT '09)*. Association for Computing Machinery, New York, NY, USA, 295–304. https://doi.org/10.1145/1629335.1629375

[36] Sunghwan Kim, Gyusun Lee, Jiwon Woo, and Jinkyu Jeong. 2021. Zero-Copying I/O Stack for Low-Latency SSDs. *IEEE Comput. Archit. Lett.* 20, 1 (2021), 50–53. https://doi.org/10.1109/LCA.2021.3064876

[37] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 2019. Faster than Flash: An In-Depth Study of System Challenges for Emerging Ultra-Low Latency SSDs. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*. IEEE, 216–227. https://doi.org/10.1109/IISWC47752.2019.9042009

[38] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, Arif Merchant and Hakim Weatherspoon (Eds.). USENIX Association, 1–15. https://www.usenix.org/conference/fast19/presentation/kourtis

[39] Gyusun Lee, Seokha Shin, and Jinkyu Jeong. 2022. Efficient hybrid polling for ultra-low latency storage devices. *J. Syst. Archit.* 122 (2022), 102338. https://doi.org/10.1016/j.sysarc.2021.102338

[40] 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 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 603–616. https://www.usenix.org/conference/atc19/presentation/lee-gyusun

[41] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *28th Symposium on Operating Systems Principles (SOSP 21)*. ACM, 819–835.

[42] 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, Virtual Event, June 28 - 30, 2022*, Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos (Eds.). ACM, 13:1–13:12. https://doi.org/10.1145/3524059.3532378

[43] Corne Lukken, Giulia Frascaria, and Animesh Trivedi. 2021. ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs. *CoRR* abs/2112.00142 (2021). arXiv:2112.00142 https://arxiv.org/abs/2112.00142

[44] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. 2022. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 635–654. https://doi.org/10.1145/3503222.3507702

[45] Till Miemietz, Maksym Planeta, and Viktor Laurin Reusch. 2021. New Mechanism for Fast System Calls. *CoRR* abs/2112.10106 (2021). arXiv:2112.10106 https://arxiv.org/abs/2112.10106

[46] 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, New York, NY, USA, 106–122. https://doi.org/10.1145/3452296.3472940

[47] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proc. VLDB Endow.* 14, 5 (mar 2021), 771–784. https://doi.org/10.14778/3446095.3446100

[48] 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, San Jose, CA, USA, February 14-17, 2012*, William J. Bolosky and Jason Flinn (Eds.). USENIX Association, 13. https://www.usenix.org/conference/fast12/fios-fair-efficient-flash-io-scheduler

[49] 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* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 13.

[50] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-Enabled Billion-Scale Graph Neural Network Training on a Single Machine via Provably Optimal in-Memory Caching. *Proc. VLDB Endow.* 15, 11 (sep 2022), 2626–2639. https://doi.org/10.14778/3551793.3551819

[51] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 554–569. https://doi.org/10.1145/3341301.3359640

[52] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The Future is Big Graphs: A Community View on Graph Processing Systems. *Commun. ACM* 64, 9 (aug 2021), 62–71. https://doi.org/10.1145/3434642

[53] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 67–78.

[54] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. OS I/O Path Optimizations for Flash Solid-state Drives. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 483–488. https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin

[55] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls

[56] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 431–445. https://www.usenix.org/conference/osdi22/presentation/stamler

[57] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2018. FastPath: Towards Wire-Speed NVMe SSDs. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 170–177. https://doi.org/10.1109/FPL.2018.00036

[58] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2020. FastPath_MP: Low Overhead & Energy-efficient FPGA-based Storage Multi-paths. *ACM Trans. Archit. Code Optim.* 17, 4 (2020), 37:1–37:23. https://doi.org/10.1145/3423134

[59] 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 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 129–145. https://www.usenix.org/conference/osdi21/presentation/tai

[60] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 397–410. https://doi.org/10.1109/ISCA.2018.00041

[61] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Kornilios Kourtis, Ioannis Koltsidas, and Thomas R. Gross. 2018. FlashNet: Flash/Network Stack Co-Design. *ACM Trans. Storage* 14, 4, Article 30 (dec 2018), 29 pages. https://doi.org/10.1145/3239562

[62] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, and Thomas R. Gross. 2013. Unified High-Performance I/O: One Stack to Rule Them All. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (Santa Ana Pueblo, New Mexcio) *(HotOS'13)*. USENIX Association, USA, 4.

[63] Paolo Valente and Mauro Andreolini. 2012. Improving Application Responsiveness with the BFQ Disk I/O Scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '12)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/2367589.2367590

[64] 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, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 403–415. https://www.usenix.org/conference/fast21/presentation/woo

[65] 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, Renton, WA, USA, July 8-9, 2019*, Daniel Peek and Gala Yadgar (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[66] 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, Haifa, Israel, May 26-28, 2015*, Dalit Naor, Gernot Heiser, and Idit Keidar (Eds.). ACM, 6:1–6:11. https://doi.org/10.1145/2757667.2757684

[67] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. 2020. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In *USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 97–110.

[68] 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, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, William J. Bolosky and Jason Flinn (Eds.). USENIX Association, 3. https://www.usenix.org/conference/fast12/when-poll-better-interrupt

[69] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*. IEEE Computer Society, 154–161. https://doi.org/10.1109/CloudCom.2017.14

[70] Youngjin Yu, Dongin Shin, Woong Shin, Nae Young Song, Jae-Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. 2014. Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Trans. Comput. Syst.* 32, 2 (2014), 6:1–6:48. https://doi.org/10.1145/2619092

[71] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*. ACM, 483–495.

[72] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong

[73] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. 2021. BPF for storage: an exokernel-inspired approach. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, Sebastian Angel, Baris Kasikci, and Eddie Kohler (Eds.). ACM, 128–135. https://doi.org/10.1145/3458336.3465290

[74] Jinbin Zhu, Limin Xiao, Liang Wang, Guangjun Qin, Rui Zhang, Yuting Liu, and Zhonglin Liu. 2021. UPM-DMA: An Efficient Userspace DMA-Pinned Memory Management Strategy for NVMe SSDs. In *Algorithms and Architectures for Parallel Processing - 21st International Conference, ICA3PP 2021, Virtual Event, December 3-5, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13155)*, Yongxuan Lai, Tian Wang, Min Jiang, Guangquan Xu, Wei Liang, and Aniello Castiglione (Eds.). Springer, 257–270. https://doi.org/10.1007/978-3-030-95384-3_17