K2: Work-Constraining Scheduling of NVMe-Attached Storage

Till Miemietz, Hannes Weisbach
Operating Systems Group
Technische Universität Dresden
Dresden, Germany
{till.miemietz, hannes.weisbach}@mailbox.tu-dresden.de

Michael Roitzsch, Hermann Härtig
Barkhausen Institut
Composable Operating Systems Group
Dresden, Germany
{michael.roitzsch, hermann.haertig}@barkhauseninstitut.org

***Abstract*—For data-driven cyber-physical systems, timely access to storage is an important building block of real-time guarantees. At the same time, storage technology undergoes continued technological advancements. The introduction of NVMe fundamentally changes the interface to the drive by exposing request parallelism available at the flash package level to the storage stack, allowing to extract higher throughput and lower latencies from the drive. The resulting architectural changes within the operating system render many historical designs and results obsolete, requiring a fresh look at the I/O scheduling landscape. In this paper, we conduct a comprehensive survey of the existing NVMe-compatible I/O schedulers in Linux regarding their suitability for real-time applications. We find all schedulers severely lacking in terms of performance isolation and tail latencies. Therefore, we propose K2, a new I/O scheduler specifically designed to reduce latency at the 99.9th percentile, while maintaining the throughput gains promised by NVMe. By limiting the length of NVMe device queues, K2 reduces read latencies up to $10\times$ and write latencies up to $6.8\times$, while penalizing throughput for non-real-time background load by at most $2.7\times$.**

***Index Terms*—I/O scheduler, work-constraining, NVMe, SSD**

I. Introduction

Timely access to persistent data is critical for many real-time applications, ranging from latency-sensitive datacenter workloads to control and telemetry systems. In recent years, storage hardware has undergone far-reaching technological advancements. Non-Volatile Memory Express (NVMe) is now the state-of-the art protocol to access flash-based storage devices directly over PCI Express without the additional hop through a Serial-ATA (SATA) controller. Compared to traditional SATA-based solid-state drives, these NVMe-attached devices exhibit two new fundamental characteristics: They process considerably higher rates of I/O operations per second and they expose device-internal parallelism to the operating system.

Modern flash storage contains multiple flash packages that can operate in parallel [1]. This parallelism is available to software as multiple request queues, which are processed out-of-order by the device and completed asynchronously. These techniques are necessary to exploit the high throughput offered by the flash hardware. The former serial interface was insufficient to saturate the capabilities of modern drives. Also, NVMe interfaces are not limited to desktop- or server-class machines. ROCK Pi[1] for example is a single-board computer for embedded IoT devices and features NVMe-attached storage.

We believe these developments pose new challenges to real-time systems that are worth exploring. There has been existing research on mediating access to solid-state storage, but it either predates these fundamental hardware changes [13, 17, 19] or targets fairness or throughput improvements rather than timeliness [10, 14, 23]. In this paper, we show that state-of-the-art I/O schedulers for NVMe devices are not focussing on request latency and therefore offer insufficient support for real-time workloads. We also present an analysis of NVMe device behavior and and an overview of the scheduling infrastructure in Linux. We point out the lessons learned and the differences presented by NVMe drives over previous storage technologies.

We propose K2, a new I/O scheduler that allows to control request latencies for NVMe-attached storage. Because K2 works with off-the-shelf NVMe drives with no modifications to drive internals, we cannot provide hard guarantees, but are limited to empirical arguments. We do not consider our research directly applicable to critical areas like autonomous driving, because off-the-shelf flash storage does not offer the level of reliability mandated by regulatory bodies. Drive-internal error correction may re-read defective data multiple times, necessitating a timeout for real-time operation. Ultimately, read requests may fail completely, requiring additional measures to ensure availability.

K2 protects the tail latencies (99.9th percentile) of real-time applications from competing lower-priority load. The concept of K2 is inspired by the Dynamic Active Subset scheduler [20] for spinning hard drives: throughput of background load is throttled to constrain the latency of real-time loads. But instead of forming subsets based on request deadlines, K2 implements bandwidth restrictions with a policy we call *work-constraining* scheduling: less work is made visible to the drive in order to bound the number of requests it can execute ahead of a real-time request.

We make the following contributions:

- We point out the key differences of NVMe-attached storage devices over previous technologies (Section II). One surprising difference is that garbage collection is no longer

[1]http://rockpi.org, retrieved May 2019

an issue for real-time behavior. Because writes are cached and sequential reads can be prefetched, random reads are now the most challenging workload. Consequently, K2 must focus on improving random read behavior.
- We provide a comprehensive analysis of existing NVMe-compatible I/O schedulers in Linux (Section III). Among those schedulers are KYBER [22], which claims to respect a configurable latency target, and BFQ [24], which claims to improve responsiveness by implementing request budgets. To our knowledge, these schedulers have not been tested for their real-time behavior on contemporary NVMe devices.
- We design and implement K2 (Section IV), a new I/O scheduler for NVMe-attached storage with predictably low request latencies in the 99.9th percentile. K2 is designed according to the constraints we have learned from analyzing NVMe device behavior and the existing schedulers. One particular constraint is that complicated scheduling policies may spend enough CPU cycles to impact the overall throughput of the drive. K2 implements a global policy to trade latency of real-time loads against throughput of background loads without unduly restricting the performance of the NVMe device.

We evaluate K2 with microbenchmarks and application workloads in Section V. We demonstrate that K2 reduces read latencies up to $10\times$ and write latencies up to $6.8\times$ in the 99.9th percentile. The source code for K2 is available under an open-source license.[2]

II. BACKGROUND

I/O scheduling has a long history, dating back at least to the 1960's with first works on storage access policies [4]. In this section, we will first revisit key historic advancements and explain, how these findings extend to the specific behavior of today's NVMe storage devices. We will also provide an overview of the NVMe subsystem in Linux as an example for the implementation constraints a modern I/O scheduler faces. Along the way, we derive lessons learned regarding the design of our proposed NVMe scheduler.

A. Evolution of Storage Hardware and I/O Scheduling

Magnetic drum storage and early spinning disk drives were among the first popular storage technologies, for which I/O scheduling strategies were investigated. A key characteristic of these devices was high access latency caused by the mechanical movement involved in accessing data. Because each misplaced request would incur a notable delay, spending CPU cycles to create a beneficial request order is economical. Thus, to improve throughput, these early drives were scheduled entirely in software by strategies such as the Elevator Algorithm and Shortest Access Time First (SATF) [4].

In the late 1990's, drives started abstracting from their internal physical layout and offered logical-block-based addressing. Scheduling individual requests became difficult, because knowledge about physical distances was no longer precise. Therefore, the related technologies Tagged Command Queueing (TCQ) and Native Command Queueing (NCQ) began to move scheduling functionality into the device itself. These in-device schedulers optimized for throughput, so the real-time community investigated augmenting or replacing those schedulers to achieve latency targets [20].

Solid-State Drives (SSDs) gained wide-spread application after 2005. Flash cells as storage medium mandated the introduction of wear leveling by way of a Flash Translation Layer (FTL). The FTL maps logical addresses to arbitrary physical storage locations on the drive. Thus, the internal layout of data on the device as well as the different characteristics of faster single- and slower multi-level cell storage [1] are decoupled from the logical view presented to the operating system. A novel issue particularly critical for real-time applications are timing hazards caused by garbage collection [13].

In recent years, SSDs migrated from SATA controller connections to being directly attached via PCIe, eliminating the additional controller. The resulting NVMe protocol achieves higher request rates and lower completion latencies. Because SSDs are internally implemented by multiple flash packages, NVMe exposes the resulting parallelism to the operating system by offering multiple request queues per device, compared to just a single request stream with SATA. This queue-based parallel interface forces a programming model with multiple in-flight requests, which is necessary to achieve the high throughput promised by NVMe storage devices. At the same time, this paradigm hands the device a large set of jobs to execute in an order decided by its own internal scheduler.

Lesson learned: Any effort at I/O scheduling for off-the-shelf NVMe-attached storage must accept that substantial internal complexity is hidden within the device. This abstraction gap has widened over generations of storage hardware and is unlikely to be closed in the foreseeable future.

An upside of the complexity-hiding is that the drives are in a distinctive position to optimize for throughput, delivering impressive performance gains to applications. In the early 2000's, spinning disks typically offered around 100 I/O operations per second (IOPS), while x86 CPU frequencies around 1 GHz were common, resulting in one I/O operation per 10 million cycles. Early SSDs around 2010 provided 50,000 IOPS, with typical CPU speeds around 3 GHz. This technology shift narrowed the gap between CPU and I/O speeds by two orders of magnitude to one I/O operation per 60,000 cycles. NVMe-attached storage improves the ratio by another $5\times$ to one I/O operation per 10,000 cycles (400,000 IOPS @ 4 GHz). This cycle count is in the order of magnitude of an x86 inter-processor interrupt round trip.

Lesson learned: The throughput difference between CPUs and I/O devices is shrinking. Any scheduling algorithm for NVMe-attached storage that spends too many cycles per request on a complex scheduling policy will harm achieved bandwidth and request latency. We share this realization with Hedayati, Shen,

[2] https://github.com/TUD-OS/k2-scheduler

Scott, and Marty [7]. Upcoming storage technologies such as 3D XPoint memory will continue this trend [6].

B. Specific Behavior of NVMe-Attached Storage

Being a fast SSD on a parallel queue-based interface, NVMe-attached storage inherits and magnifies many of the behavioral quirks of SSDs. Writing data on flash storage works differently than on magnetic media: existing data in flash cells has to be erased before new data can be written, adding an extra erasure step when modifying existing data. While read and write requests are organized in pages with typical sizes of 2, 4, or 8 KiB, erasure is performed on larger block sizes dictated by the internal structure of the SSD. This size mismatch requires additional data copying of pages that should not be erased. Altogether, erasure is a slow operation that can significantly throttle write performance [12, 19].

To mitigate this problem, modern NVMe drives use a portion of the flash cells as a fast *write cache*. In this mode, the cells only store a single bit, i.e. a single voltage level is used (SLC — single level cells), which is faster than typical storage operation with multiple voltage levels. The write cache is pre-erased to absorb write requests as quickly as possible. Depending on the drive model, the cache can be dynamically allocated. Therefore, problems with degrading write performance only occur, when the drive fills up and less storage capacity can be spent on this cache.

In order to prepare flash blocks for writing, the drive firmware needs to erase them. Any data that should be kept needs to copied to a different flash block. This process is known as garbage collection. As an optimization, the operating system can inform the drive, whenever data pages are discardable, for example when their enclosing file has been deleted within the file system. This optimization is called *trimming* and allows the drive to elide the copy step and erase the data in place. Reading from trimmed pages will return unspecified data — either all ones, all zeros, or the previous content [16]. The drive will asynchronously coalesce and erase trimmed pages and can add them to the write cache.

Lesson learned: Caching on modern drives helps to serve writes with high bandwidth and low latency. A system designer only needs to use a file system with support for trimming and ensure sufficient free space on the drive. Contrary to previous SSD technologies, block erasure and garbage collection overheads are largely moved off the critical path.

With writes being absorbed by a fast cache and sequential reads being prefetched, random reads turn out to be the challenging usage scenario for real-time scheduling. We will support this conclusion with experiments in Section III.

C. Block I/O in Operating Systems

As an example of a modern I/O subsystem capable of achieving high throughput on NVMe-attached devices, we summarize the implementation of block I/O in the Linux kernel, particularly version 4.15.0[3]. The Linux subsystem for block I/O

[3]https://www.kernel.org

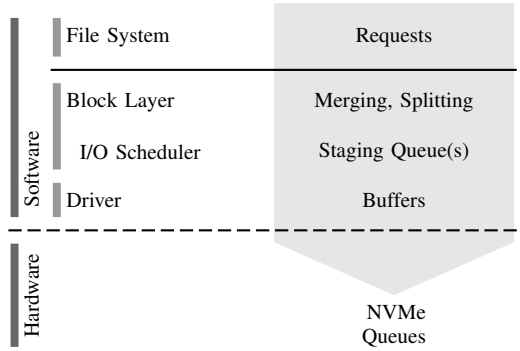

Figure 1. Block I/O in the Linux operating system.

is called the *block layer*. It mediates between the file system and the device driver as a common interfacing component. As such, it also performs shared management tasks like I/O scheduling and accounting. Figure 1 illustrates the Linux block layer.

When processing parallel requests at high rate, the operating system can become a bottleneck in the storage stack. Sequential data structures, global locks, and cache interference can severely impact I/O performance. To avoid such problems, Linux introduced a multi-queue block layer [2], which is based on parallel request processing. An I/O request is submitted in the form of a scatter-gather structure describing the block device regions it affects. Upon entering the block layer, requests to consecutive regions can be merged and are then inserted into a *staging queue*. In the default configuration, the Linux kernel allocates one such queue per CPU core. In contrast to a global queue, such core-local queues reduce cross-core locking operations. Requests in the staging queue are still mutable, so they can be merged with other incoming requests.

The kernel queries the staging queues to select requests for promotion to *driver buffers*, which then feed into the block device driver. Each staging queue has one driver buffer assigned. The driver will forward the buffered requests to the *NVMe queues* provided by the storage device, where the device will autonomously process them and signal completion back to the driver. The kernel aims at establishing a one-to-one mapping between staging queues and NVMe queues, so one NVMe queue is dedicated to each core, again reducing cross-core interactions. I/O requests in the driver buffer and NVMe queues are immutable and must not be reordered, because the driver may forward them to the device at any time and the device independently consumes the NVMe queues at its own pace.

Lesson learned: The Linux block layer architecture is designed to avoid cross-core interactions. However, this design complicates the implementation of a global policy like prioritizing requests according to their timing constraints. Our scheduler needs to find a balance between these conflicting objectives.

The last step of consuming requests from the NVMe queues is performed by the drive itself. The NVMe standard

defines a round-robin policy to arbitrate between requests from different NVMe queues. A weighted round-robin strategy is also specified [9], but only as an optional feature, which we do not consider, because relying on it would restrict the applicability of our results. In which order requests are then fulfilled and retired is not specified and entirely up to the internal decisions of the drive's controller.

D. Linux I/O Schedulers

I/O scheduling is the process of deciding, which I/O requests should reach the device in what order. Schedulers can implement different strategies to maximize throughput, reduce latency, or improve fairness. These goals can be contradictory, so schedulers have to make policy decisions when designing their strategy. The Linux block layer offers powerful extension points to implement a variety of policies.

I/O schedulers in Linux are realized as kernel modules. A scheduler offers implementations for a set of functions that the block layer calls, whenever an event like the arrival of a new request occurs. I/O schedulers can instantiate their own staging queue data structure, which then overrides the default staging queue implementation. Schedulers can also implement custom request promotion strategies like holding back requests in the staging queues, if the NVMe queues are too long.

We now briefly describe all I/O schedulers available in the standard Linux kernel that are compatible with the multi-queue block layer. When no specific scheduler is active, the NONE policy is used: Synchronous requests from applications are directly inserted into the driver buffer, bypassing the staging queues. Asynchronous requests are first inserted into the staging queue, but immediately propagate to the driver buffer unless the driver buffer is full. No reordering is performed.

MQ-DEADLINE is a simple scheduler designed for spinning hard drives. It orders requests by their target address in order to minimize seek operations. To prevent starvation of operations that access remote disk regions, each request is given an artificial deadline. Whenever a request violates this deadline, it is served immediately. However, this deadline cannot be used to prioritize amongst multiple applications.

BUDGET FAIR QUEUING (BFQ) [24] is a priority-aware fair queuing scheduler that provides bandwidth guarantees to processes. With BFQ, each process maintains a private request queue for synchronous operations. Asynchronous operations are collected in additional global queues according to their priority. BFQ divides time into epochs. When a new epoch starts, each queue receives a bandwidth budget that it may spend by issuing requests during the current epoch. Request arbitration uses a heuristic based on budget and priority. Whenever all queues have used up their budget or there is no more outstanding work, a new epoch starts and all budgets are replenished.

The KYBER scheduler was exclusively designed for SSDs [22]. Furthermore, it is the only strategy making use of core-local staging queues instead of managing all requests in a global data structure. KYBER categorizes requests into three different classes: reads, synchronous writes, and other operations like trim requests. The scheduler then applies a token-bucket algorithm to limit the amount of requests that each class may have in flight within the driver buffer or the NVMe queues. This arbitration between the different request types allows to control the latency experienced by requests of each class. However, prioritization between requests of the same type, but of different applications is not considered.

Table I
MAXIMUM DEVICE THROUGHPUT AND IOPS AS CLAIMED BY THE MANUFACTURER AND THE PERFORMANCE WE WERE ABLE TO ACHIEVE.

Workload	Queue Depth/ No. of Threads	Performance Manufacturer	Performance Measurement
Seq. Read		3400 MB/s	3070 MB/s
Seq. Write (burst)		1500 MB/s	1500 MB/s
Seq. Write (sust.)		300 MB/s	320 MB/s
Ran. Read	1 / 1	15 kIOPS	12.7 kIOPS
Ran. Read	32 / 4	200 kIOPS	195 kIOPS
Ran. Write	1 / 1	50 kIOPS	80 kIOPS
Ran. Write (burst)	32 / 4	350 kIOPS	344 kIOPS
Ran. Write (sust.)	32 / 4	80 kIOPS	73 kIOPS

Lesson learned: With the exception of MQ-DEADLINE, which targets spinning drives, all Linux I/O schedulers operate some form of traffic shaping to manage bandwidth or latency. Schedulers do not inspect and reorder requests based on their individual characteristics, but broadly arbitrate between streams of events.

One possible reason is that schedulers do not want to spend too many cycles executing their policy. As our measurements in the next section will show, BFQ — which implements the most complex policy — already impacts latency results.

Lesson learned: Given that NVMe-attached devices do not execute individual requests one by one, but can have multiple requests in flight, the work-conserving nature of a scheduler becomes non-binary. Other than a CPU, which either executes or idles, we can pass fewer or more requests to the storage device, depending on the scheduling policy.

III. DISSECTING LINUX' I/O SCHEDULERS

We use this section to assess existing I/O schedulers in Linux. To the best of our knowledge this is the first study examining the latency characteristics of the Linux I/O schedulers for NVMe drives and their suitability for real-time workloads.

Our system is based on a quad-core Intel i5-4590 with SMT disabled and 16 GiB of RAM. We selected a Samsung 970evo with a capacity of 250 GB[4] as our test subject. The 970evo is based on 3 bit MLC V-NAND, features a PCIe Gen 3.0 x4 interface, and implements the NVMe 1.3 protocol. To illustrate the complex performance characteristics of NVMe drives we reproduce the throughput values measured by the manufacturer from the device datasheet [21] in Table I. These figures illustrate the impact of the write cache, queue depth parameter, and exploitation of parallelism on the performance achieved by the drive.

[4]Model Code MZ-V7E250BW

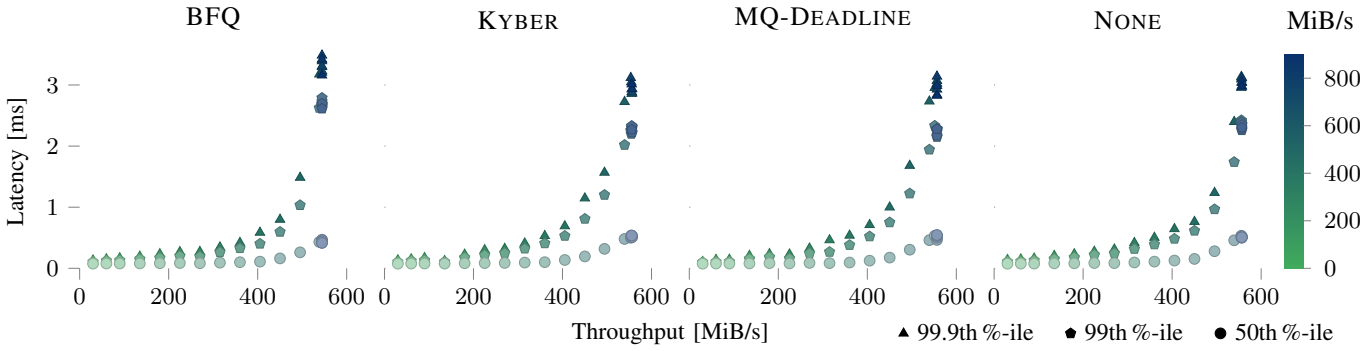

Figure 2. Bandwidth-latency plots of standard Linux I/O Schedulers for random read workload with 4 KiB block size.

A. Benchmark Setup

To investigate the latency characteristics of the Linux I/O schedulers, we measure the makespan of disk requests from entering the I/O scheduler until completion of the request by the device. We perform these measurements on Linux 4.15 using kernel tracepoints and the LTTng infrastructure. We added new tracepoints or modified existing tracepoints to capture request submission, request propagation between queues, and request completion. LTTng uses high-resolution timers with nanosecond granularity.

We use the Flexible I/O Tester (FIO) version 3.1 to conduct our microbenchmarks: three processes simulate background workload while a fourth process acts as a real-time application issuing a stream of requests with an interval of 2 ms between consecutive requests. FIO configures I/O priorities to distinguish background and real-time tasks. The background load uses asynchronous I/O to achieve higher throughput, while the real-time task performs synchronous I/O. In synchronous mode, FIO issues an I/O request and then waits for the specified time of 2 ms before issuing the next request. FIO therefore does not precisely implement a strictly periodic job submission, but because it is a standard load generator, we refrained from modifying it. Due to this operation mode, real-time request queues never saturate and requests are never dropped. We think the results produced are easier to interpret.

Keeping the configuration of the real-time application constant, we ramp up the bandwidth requirement of the background workload. Each plotted point constitutes one such configuration, where we measure the makespan of real-time requests as well as the bandwidth actually achieved by the background workload.

We tuned the I/O schedulers for low latency as follows: Using KYBER's configuration options, we reduced the target latency for read requests to 2 µs and for write requests to 10 µs. For BFQ, we disabled the anticipation mechanism for HDDs. Our attempts at tuning MQ-DEADLINE resulted in increased I/O latency, so we used the default configuration.

We loaded the drive with random and sequential read and write accesses, as well as mix of random read and write accesses, each for block sizes of 4 KiB to maximize IOPS and 64 KiB to maximize throughput. We configured FIO to use direct I/O to forgo the Linux buffer cache and ensure our results reflect drive access latencies.

The amount of trimmed pages influences performance, because it impacts the ability of the drive's firmware to perform garbage collection. For example, reading and writing a fully trimmed disk is very fast but also uncommon. Reading trimmed pages does not require the drive to access its flash and is therefore very quick[5]. Writing to a fully trimmed disk is also very fast, because empty blocks are readily available. The other extreme is writing to a disk with no trimmed pages. Because the drive has to reorganize and garbage collect very often, access latency increase and throughput decreases. We consider both cases as hypothetical, as they should occur rarely, if ever, in practice. Nonetheless we performed our benchmark on those drive configurations out of academic curiosity and we were able to reproduce the expected results. In the remainder of the paper — unless otherwise noted — we configured the drive such that half of its blocks are occupied and the other half are trimmed.

B. Benchmark Results

We present the results of our analysis as bandwidth-latency plots. The bandwidth available to background load is plotted on the x-axis and the median, 99th, and 99.9th percentile of the real-time request latency is on the y-axis. The color of the plot marks indicates the bandwidth target we configured for the background workload. Due to the limited space available we do not reproduce all measurements as graphs but concentrate on interesting and insightful results.

We start our examination in Figure 2 by comparing achieved bandwidth and latency when the drive is exercised with a random read workload with 4 KiB block size. There is neither much qualitative nor quantitative difference between the schedulers: with low background load each scheduler is able achieve very low latencies of less than 200 µs; maximum bandwidth is around 550 MiB/s and latencies increase to 3 ms. A random read workload with 64 KiB block size (not shown) looks qualitatively similar, but when the drive is saturated

[5]As per the NVM Express specification 1.3, §6.7.1.1 the drive is free to return a value of all zeros, all ones, or the previous data for such a request.

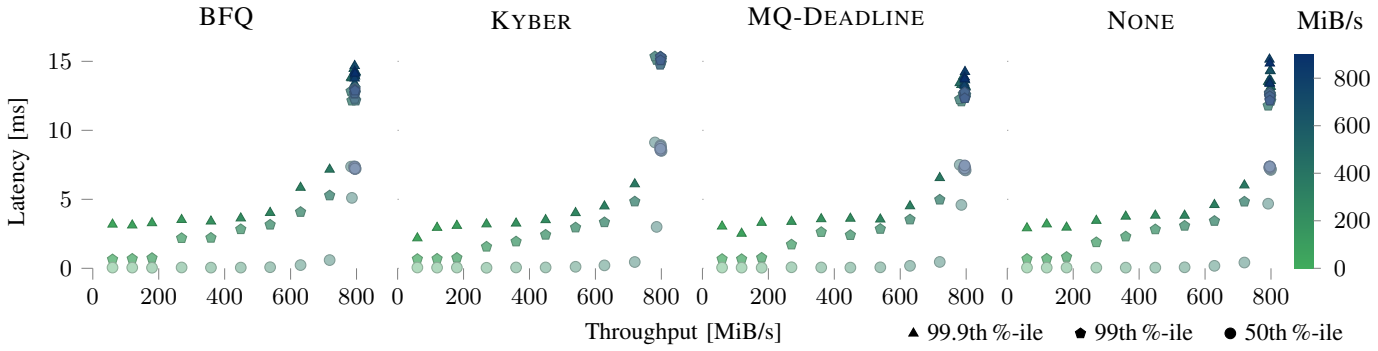

Figure 3. Bandwidth-latency plots of standard Linux I/O Schedulers for random read/write workload with 64 KiB block size.

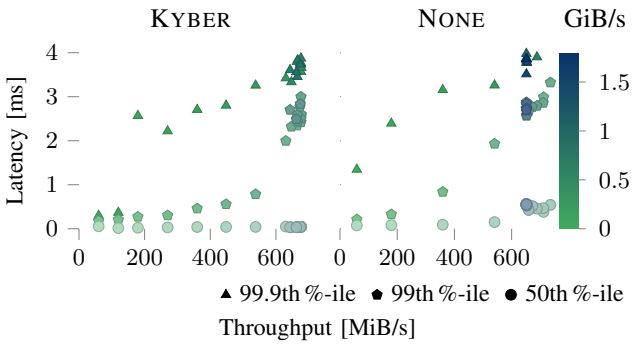

Figure 4. Bandwidth-latency plots for KYBER with random read/write workload (left) and NONE with random read/write workload for background tasks and random read for foreground tasks, both with 4 KiB block size.

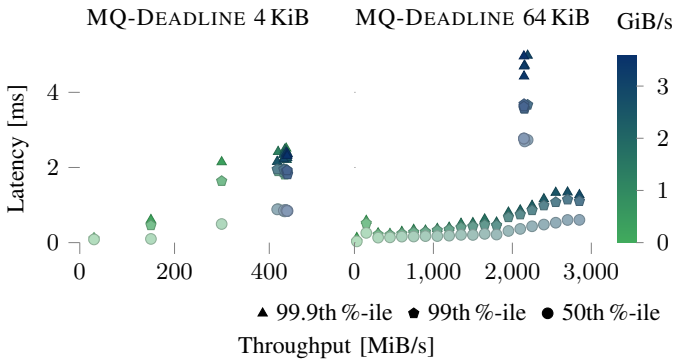

Figure 5. Bandwidth-latency plots for MQ-DEADLINE with sequential read workload with 4 KiB (left) and 64 KiB (right) block size.

at little over 550 MiB/s, latencies increase significantly up to 14 ms.

Next, we turn to a 50/50 mixed random read/write workload with a block size of 64 KiB in Figure 3. Here again the different I/O schedulers behave very similarly. For low to medium throughputs of up to 600 MiB/s median latency continues to be around 100 µs to 200 µs, while 99.9th-percentile latencies are in the order of magnitude of 5 ms, significantly higher compared to read-only workloads. When the background load is increased to the maximum throughput of 800 MiB/s, 99.9th-percentile latencies rise even further, up to 17 ms in case of KYBER.

With the data presented so far, we hope to convince the reader that there is little qualitative difference between schedulers. This remains true for the following experiments, so to save space we only show one representative for the following workloads.

We selected KYBER as representative for a 50/50 random read/write workload with a block size of 4 KiB shown in Figure 4. While the 99th percentile is close to the median for a wide range of throughput values, once the drive is saturated the 99th percentile quickly closes the gap to the 99.9th percentile which itself increases only slightly. The median latency is not affected by drive saturation.

So far foreground and background load always performed the same work type. For the next experiment we mixed a random read foreground workload with an 80/20 random read/write background workload. We selected NONE as the proxy for this benchmark group plotted in Figure 4 (right). The result is not substantially different from a homogeneous random read/write workload. While the 99th percentile is close to the median in a pure random read/write workload, we measured a 2 ms higher latency for the lowest throughput values when the foreground application performs a read-only workload, contending against a random read/write background load.

In Figure 5, we compare the bandwidth-latency characteristics under a sequential read workload with 4 KiB (left) and 64 KiB (right) block sizes. In this case we selected MQ-DEADLINE as our proxy. The sequential read throughput with a block size of 4 KiB is — independently of the scheduler — lower than for random reads. We speculate that random reads are faster, because random accesses potentially exploit more drive-internal parallelism. Sequential data seems to be laid out in a less favorable way. Nevertheless, maximum latencies decrease from around 3.5 ms for random reads to approximately 2.5 ms for sequential reads. In case of a block size of 64 KiB all schedulers achieve a throughput of 2850 MiB/s, including MQ-Deadline depicted in Figure 5. Contrary to previous results, increasing the requested throughput further results in a sharp drop in achieved throughput down to 2100 MiB/s and an equally sharp increase in latency up to 5 ms and 7 ms for BFQ.

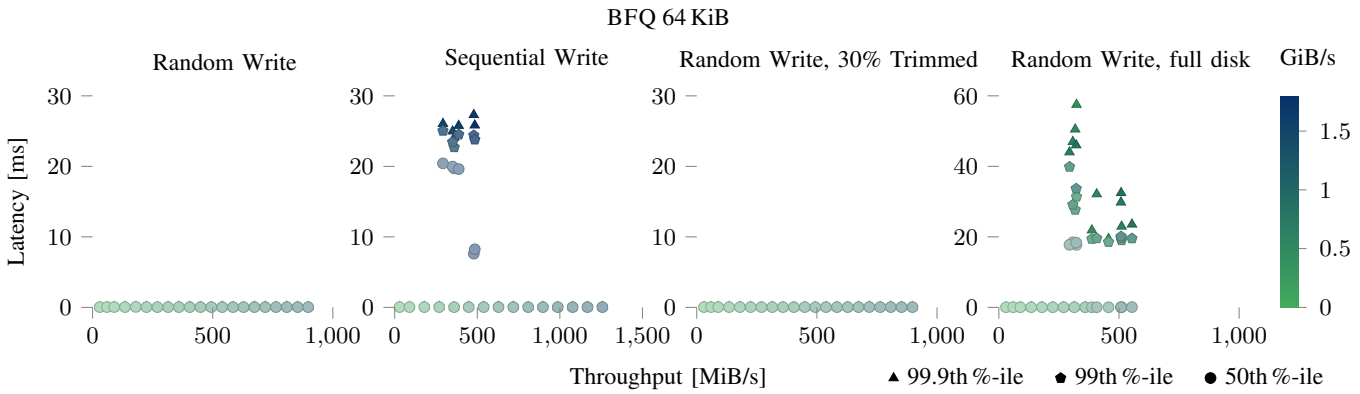

Figure 6. Bandwidth-latency plots of BFQ with 64 KiB block size for random (left) and sequential writes (middle left), and for random writes on a 30% trimmed (middle right) and on a full drive (right).

In Figure 6, we juxtapose the bandwidth-latency plot of a random write workload (left) and a sequential write workload (middle left), both with a block size of 64 KiB. Random write workloads result in a latency of less than 100 µs even in the 99.9th percentile for all schedulers, for block sizes 64 KiB and 4 KiB (not shown), and up to the saturation throughput of roughly 900 MiB/s. With a block size of 4 KiB BFQ is able to sustain a throughput for the background workload of 721 MiB/s, MQ-DEADLINE and KYBER achieve a background throughput of 853 MiB/s, and the NONE scheduler reaches 898 MiB/s.

Similar to random writes, sequential writes also show latencies below 100 µs for all schedulers and both block sizes. With 4 KiB blocks, BFQ is able to achieve a maximum throughput of 759 MiB/s, KYBER of 901 MiB/s, MQ-DEADLINE of 909 MiB/s, and NONE of slightly over 1 GiB/s. In the configuration with 64 KiB block size all schedulers are able to exploit the full performance of the drive with background throughputs of 1201 MiB/s to 1257 MiB/s and consistently low latencies below 100 µs. If the bandwidth requirement of the background work is further increased, the write cache of drive is exceeded and the throughput drops suddenly to around 500 MiB/s and from there on decreases further to the minimum TLC write speed of 300 MiB/s. At the same time latencies increase to 25 ms to 33 ms. Only at this point would write buffering at the operating system level be beneficial.

So far, our measurements were all performed with half of the blocks on the drive trimmed and therefore available for write caching. In Figure 6 (middle right), we show results for random writes of 64 KiB blocks with BFQ, when 30% of the blocks are trimmed. Even for this higher fill level, the write cache enables latencies of less than 100 µs.

To emphasize the importance of trimmed blocks for write latencies, we repeat the previous experiment — random writes of 64 KiB blocks with BFQ — on a full drive, i.e. with no trimmed blocks. The results are shown in Figure 6 (right). In this configuration the drive is able to sustain only half of the throughput and latencies increase from 100 µs in the 99.9th percentile to over 50 ms. We posit that a completely full drive constitutes an unrealistic operation point. In such a situation performance characteristics will be dictated by the drive, regardless of any I/O scheduling policy.

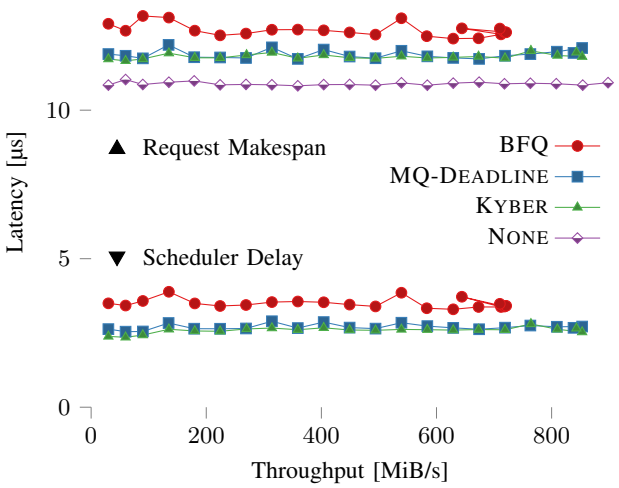

Figure 7. Median request latency and scheduler delay for Linux I/O schedulers for random write workload with 4 KiB block size.

These results show both the efficacy and importance of the write cache. There is no difference in latency between random and sequential writes on this NVMe flash device, but throughput is about 30% higher for sequential writes. Only with sequential writes in combination with large block sizes were we able to exceed the write buffer of the drive and observe the sustained write speed.

Another question we asked ourselves is how the different complexities of the Linux I/O schedulers may influence the achievable throughput and latency. We theorize that there is a tipping point where the additional delay of complex analysis and request reordering does not translate anymore into higher bandwidth and lower latency but instead decreases bandwidth and increases latency. We believe BFQ to be an example of such a complex scheduler. For a random write workload with a

block size of 4 KiB BFQ achieves a throughput of 721 MiB/s, compared to 853 MiB/s for KYBER and MQ-DEADLINE, and 898 MiB/s for NONE, at comparable latency. This corresponds to a throughput penalty of 85% and 80%, respectively. To get an idea of how the processing time of a scheduler affects throughput and latency we compare makespan and processing time.

In Figure 7 we compare the median latency for the random write benchmark with 4 KiB block size with the time the scheduler took to process that request, which we call *scheduler delay*. Since NONE passes synchronous requests directly to the driver buffer, it experiences no such delay. MQ-DEADLINE and KYBER have roughly the same median scheduler delay and also achieve roughly the same latency. BFQ has about 1 µs higher scheduler delay which directly causes the makespan to also increase by 1 µs, or 8% for this workload.

C. Summary

We observed in our benchmarks that more often than not the bandwidth-latency characteristics of a workload are dominated by the behavior and particularities of the NVMe drive and not the I/O scheduler itself. We still believe we can gain some insights from our measurements, not only about the existing I/O schedulers in Linux, but also insights specific to NVMe storages devices:

Writes are fast. The indirection introduced by the FTL causes random and sequential writes to have no difference in performance, neither in terms of latency nor bandwidth. Unless the write cache is full, writes are blazingly fast with write latencies below 100 µs, often less than 20 µs at throughputs of up to 900 MiB/s, only BFQ is significantly slower at 759 MiB/s for sequential writes with 4 KiB blocks.

Reads are slow(er). Reads have considerably higher latencies than writes, and sequential reads with small block sizes have lower throughput than random reads with the same block size, because the drive cannot take advantage of its internal parallelism. Large block sizes alleviate this problem by amortizing over more data. This is were BFQ shines and outperforms all other schedulers in terms of throughput.

No performance isolation. From a real-time perspective we can conclude that no scheduler can maintain low request latencies for the real-time workload under high throughput background workload. This result is only surprising in the case of BFQ, which explicitly distinguishes between I/O priority classes which none of the other scheduler do.

Two-tier scheduling. Except for corner cases, changing the I/O scheduling policy on the operating systems side has no impact on the measured throughput and latency. Modern SSDs offer parallel access to their flash packages, which is managed by a drive-internal scheduler. Combined with long queue depths, the behavior of the drive-internal scheduler dominates the observable behavior of the drive.

Less is more. Less complexity means less delay. BFQ — an example of a complex scheduler — spends about 50% more time on processing requests, which can have a measurable impact on request latency. BFQ also incurs 15% more CPU overhead than other I/O schedulers. This finding calls for a simple solution to reduce scheduling overhead, otherwise the scheduler will constrain the performance of the device [7].

We believe sub-20 µs latencies for write requests are hard to improve upon. However, in case of random and sequential reads our results show that latencies observed by our real-time workload correlated with increasing throughput of the background workload. In the following section we describe how we built an I/O scheduler that is able to trade throughput of background applications for predictable latencies of real-time applications.

IV. THE K2 SCHEDULER

Neither of the I/O schedulers we evaluated was able to isolate real-time application performance from background loads. Latencies suffer under the NONE, MQ-DEADLINE, and BFQ schedulers because they are work-conserving: As long as there are requests in the staging queues, these policies will promote them to the NVMe queues. On one hand, this improves global throughput since more requests in flight result in more opportunities for the drive to optimize. On the other hand, long NVMe queues lead to many competing requests, increasing the probability for real-time requests to be stalled. Therefore, higher background load leads to increased latencies of real-time applications.

A. Work-Constraining Scheduling and Performance Isolation

Generalizing the work by Reuther and Pohlack on Dynamic Active Subset [20], we apply the construction principle of work-constraining scheduling: We limit the choices of a second-tier scheduler by constraining the amount of work visible to it. In our case, the second-tier scheduler is the request arbitration within the controller of the NVMe drive. K2 constrains the amount of work seen by the controller by bounding the length of the NVMe queues. This length bound gives system designers a tunable parameter to trade latency reduction against throughput loss.

A similar strategy of request throttling is implemented by KYBER, but it does not differentiate between applications and therefore cannot ensure performance isolation. K2 needs to separate requests according to their I/O priority. Assuming the availability of trimmed blocks for write caching, we have observed that write requests and garbage collection are no longer problematic and the different request types do not influence each other significantly. Therefore, K2 opts for simplicity and does not distinguish between request types.

B. Design and Implementation of the K2 Scheduler

K2 is implemented as a kernel module for Linux. It maintains nine separate staging queues, one for each of the eight Linux real-time I/O priority levels and a ninth one for all non-real-time requests. Whenever a new request is submitted to the block layer, its I/O priority determines the queue the request is inserted into. Each queue is organized in FIFO order, so requests from real-time applications with equal I/O priority are processed in submission order.

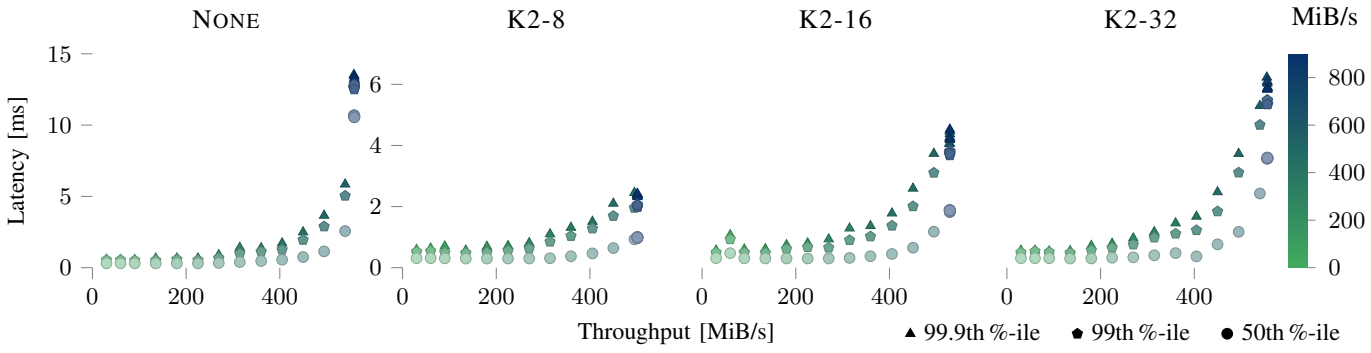

Figure 8. Bandwidth-latency plots for random read workload with 64 KiB block size.

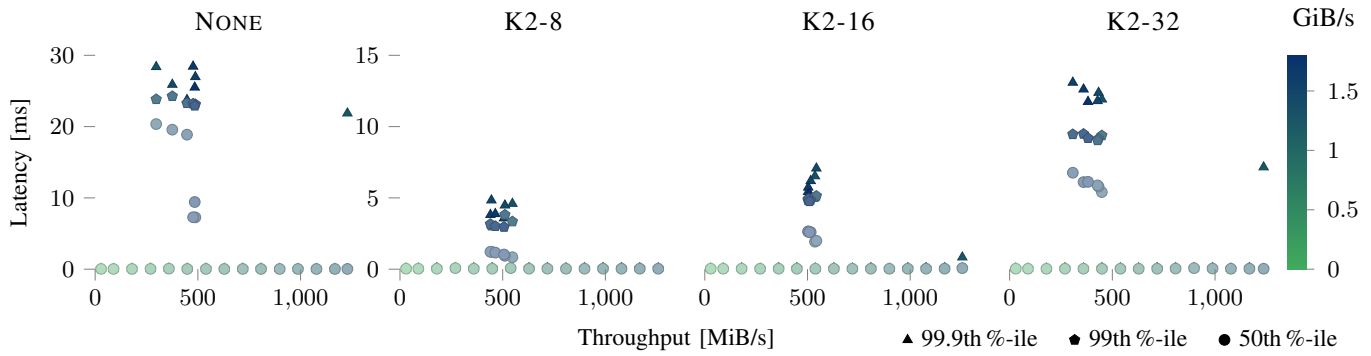

Figure 9. Bandwidth-latency plots for sequential write workload with 64 KiB block size.

When the kernel invokes K2 to promote requests to the driver buffers and ultimately to the NVMe queues, K2 treats requests in strict priority order: the head request of the highest priority non-empty staging queue is promoted to the NVMe queue. However, when the current *total length* of the NVMe queues exceeds the configured bound, work-constraining kicks in and K2 deliberately refrains from propagating a request. The request then remains in the staging queue until the total length of the NVMe queue is reduced by a request completion. Total length refers to the number of requests currently in-flight across all parallel NVMe queues, thus enforcing a device-wide upper load bound. We need to access cross-core data structures to implement this strategy, but we favored a global policy over a strictly core-local implementation. Otherwise, high load on one core could negatively impact real-time latencies on other cores.

Whenever an I/O request is completed, a slot in the NVMe queues becomes available. K2 then re-evaluates its staging queues to propagate the head request from the highest-priority non-empty queue.

K2 trades overall throughput for responsiveness. The throughput degradation depends on the number of requests allowed to be in flight simultaneously and therefore on the configured total length bound. Increasing this bound weakens the latency protection for real-time applications. When setting this parameter to infinity, K2 approximates the NONE policy.

V. EVALUATION

To evaluate K2, we use the same hardware as in Section III: a quad-core Intel i5-4590 with SMT disabled, 16 GiB of RAM, and a Samsung 970evo NVMe SSD with a capacity of 250 GB. K2 runs as an I/O scheduler provided by a kernel module on a Linux 4.15 kernel. We begin with a microbenchmark setup, also similar to Section III: three processes of the Flexible I/O Tester (FIO) simulate background load; a fourth instance constitutes the real-time load issuing a stream of requests with an interval between consecutive requests of 2 ms. I/O priorities are configured such that the real-time application takes precedence over the background load.

A. Microbenchmarks

Our first experiment executes random reads as both the real-time and background load. We used a block size of 64 KiB, because we have observed that larger block sizes emphasize differences between the schedulers. Figure 8 shows a bandwidth-latency plot with achieved bandwidth of the background load on the x-axis and request latency of the real-time task on the y-axis. We run K2 with different configurations of the queue length bound: 8, 16, and 32 in-flight requests. We give the results of the NONE policy for comparison. At the 99.9th percentile, the real-time task experiences latencies of up to 14 ms with NONE, which K2 reduces significantly to just 2.5 ms with a queue bound of 8. Background throughput

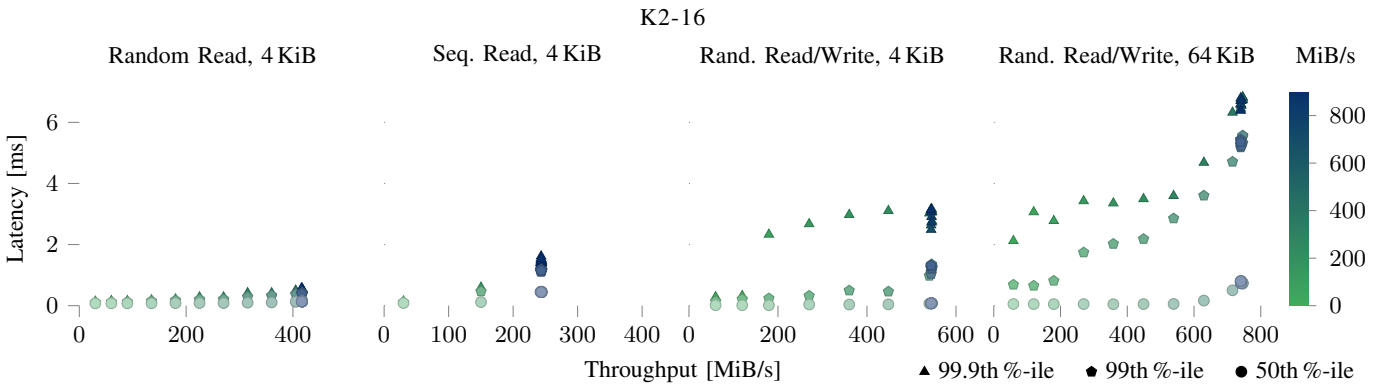

Figure 10. Bandwidth-latency plots for read and mixed read/write workloads with varying block size.

Table II
MAXIMUM ACHIEVED THROUGHPUT FOR ALL SCHEDULER/WORKLOAD CONFIGURATIONS AND THE CORRESPONDING 99.9TH PERCENTILE LATENCIES.

Maximum Throughput [MiB/s] (at 99.9th percentile Latency [ms])

	Random						Sequential			
	Read		Write		Read/Write		Read		Write	
	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB
NONE	557 (3.04)	559 (13.15)	898 (0.01)	898 (0.04)	714 (5.15)	799 (13.59)	478 (2.12)	2849 (1.37)	1055 (0.02)	1228 (21.87)
BFQ	545 (3.31)	543 (13.96)	721 (0.02)	897 (0.06)	704 (4.67)	797 (14.23)	695 (1.18)	2849 (1.34)	759 (0.02)	1257 (0.06)
KYBER	557 (2.87)	562 (12.84)	853 (0.02)	899 (0.08)	677 (3.66)	801 (16)	443 (2.32)	2850 (1.37)	901 (0.02)	1201 (21.69)
MQ-DL.	557 (3.14)	562 (12.8)	853 (0.02)	898 (0.07)	712 (4.3)	798 (13.13)	439 (2.37)	2849 (1.27)	909 (0.02)	1255 (0.07)
K2-8	288 (0.31)	501 (2.31)	899 (0.02)	898 (0.08)	394 (2.98)	696 (4.84)	219 (1.16)	1049 (0.97)	894 (0.02)	1257 (0.06)
K2-16	417 (0.55)	529 (4.19)	898 (0.02)	897 (0.04)	546 (2.77)	746 (6.83)	244 (1.44)	1186 (2.01)	890 (0.02)	1254 (0.84)
K2-32	493 (0.96)	557 (5.89)	856 (0.02)	899 (0.08)	648 (3.12)	794 (8.62)	345 (1.55)	1362 (3.23)	899 (0.02)	1237 (7.15)

Table III
MAXIMUM OBSERVED 99.9TH-PERCENTILE LATENCY FOR ALL SCHEDULER/WORKLOAD CONFIGURATIONS AND THE CORRESPONDING THROUGHPUT.

Throughput [MiB/s] (at Maximum 99.9th percentile Latency [ms])

	Random						Sequential			
	Read		Write		Read/Write		Read		Write	
	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB	4 KiB	64 KiB
NONE	556 (3.13)	558 (13.53)	60 (0.06)	224 (0.09)	714 (5.15)	795 (15.14)	299 (2.36)	2147 (4.62)	90 (0.07)	477 (28.44)
BFQ	545 (3.48)	543 (14.74)	30 (0.07)	224 (0.08)	704 (4.67)	794 (14.67)	680 (1.29)	2202 (7.26)	90 (0.07)	479 (27.3)
KYBER	554 (3.12)	562 (13.41)	30 (0.05)	224 (0.09)	676 (3.87)	797 (17.36)	434 (2.52)	2146 (4.36)	30 (0.05)	479 (33.11)
MQ-DL.	557 (3.14)	562 (13.39)	90 (0.03)	853 (0.08)	711 (4.75)	796 (14.24)	436 (2.51)	2193 (4.98)	30 (0.04)	353 (24.99)
K2-8	288 (0.34)	495 (2.45)	60 (0.06)	314 (0.08)	388 (3.11)	684 (4.91)	217 (1.27)	1015 (1.71)	30 (0.06)	445 (4.83)
K2-16	416 (0.58)	529 (4.52)	30 (0.06)	134 (0.09)	545 (3.17)	746 (6.83)	243 (1.62)	1184 (2.12)	30 (0.03)	543 (7.07)
K2-32	492 (1.13)	555 (6.23)	30 (0.05)	134 (0.09)	647 (3.38)	787 (9.45)	299 (1.79)	1349 (3.54)	90 (0.06)	308 (13.09)

degrades moderately from 600 MiB to 500 MiB. As expected, the other K2 configurations achieve higher throughput at the cost of higher latency.

Next, we evaluate sequential writes with 64 KiB blocks. We have seen that write requests are generally fast as long as there is room in the drive's write cache. Once the cache is full, write throughput drops and latency sharply increases as the drive starts to perform garbage collection. Figure 9 shows the results again under NONE and the same configurations for K2. Note that the K2 plots use a different y-axis than NONE. We observe the same low write latencies with all schedulers. From the green towards the blue plot points, the background load increases its target throughput. As it reaches roughly 1250 MiB/s, the expected drop in throughput occurs, with all schedulers performing around the 500 MiB/s mark. However, K2 manages to constrain the resulting increase in request latencies.

We also performed a measurement of random writes with 64 KiB blocks for a configuration, where 30% of the available drive capacity is trimmed. In this setup, fewer blocks are available for write caching. As we have observed in Section §III for BFQ, K2 achieves the same latencies with 50% and 30% trimmed blocks.

Figure 10 shows a selection of additional workload scenarios. We have selected K2 with a queue bound of 16 requests. We again observe the peculiarity, that random reads are faster

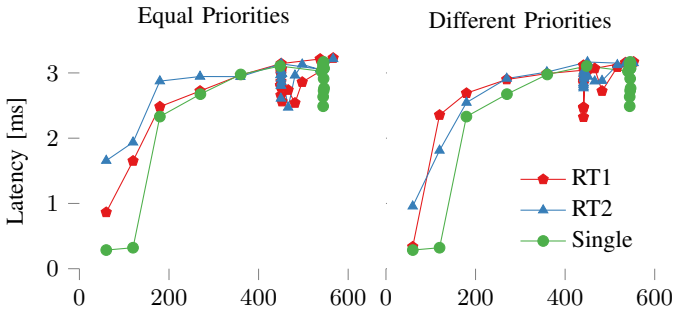

Figure 11. 99.9th percentile latencies for K2 with multiple real-time tasks with the same priority (left) and RT1 having a higher priority than RT2 (right).

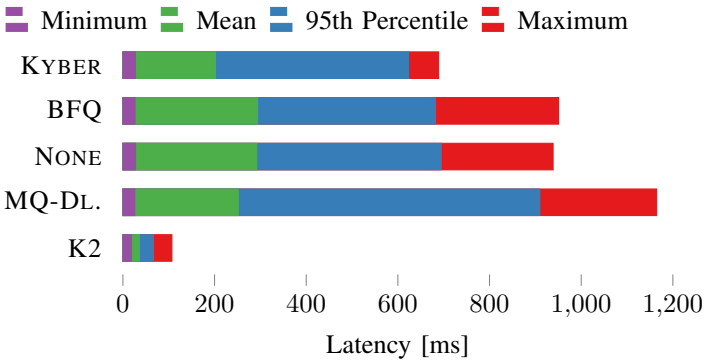

Figure 12. Event processing times of the Sysbench SQL benchmark for different I/O schedulers with a read background load.

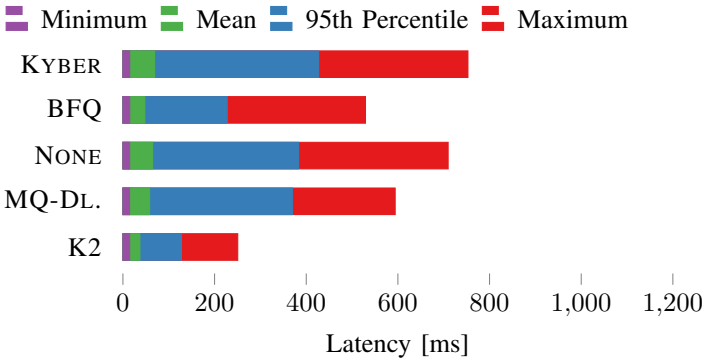

Figure 13. Event processing times of the Sysbench SQL benchmark for different I/O schedulers with a read/write background load.

than sequential reads. The 50/50 random read/write workload experiences higher latencies, but K2 keeps all results well below 8 ms.

Finally, we present the maximal throughput in Table II and maximal latencies in Table III for all scheduler and workload combinations. As expected, K2 with a queue bound of 8 loses some throughput for read operations, but with significant reduction in 99.9th-percentile latencies.

B. Multiple Real-Time Tasks

To show composability of K2, we now run two real-time applications RT1 and RT2 simultaneously. The background load executes an 80/20 random read/write mix, while the real-time applications issue random read requests, all with 4 KiB block size. We evaluate two different setups: First, we set the same I/O priority for RT1 and RT2, and we keep the 2 ms inter-request interval, which we used in all previous experiments. Second, we decrease the I/O priority of RT2 and also decrease its inter-request interval to 250 µs. We chose this setup to show that the lower-priority task RT2 does not impact the higher-priority RT1, even though RT2 exercises a higher load.

Figure 11 shows the 99.9th percentile latencies for both setups against a baseline of running a single real-time task against background load. We can see that these setups do not differ in their achieved bandwidth or latency except for some latency differences in the low-bandwidth runs. As long as the drive is not overloaded, requests from both real-time applications can be served with latencies below 3 ms.

C. Application Benchmark

We round out the evaluation with the Sysbench benchmark[6] in the OLTP read-only configuration. This setup emulates a database querying workload using the MySQL InnoDB engine. The benchmark executes mostly random read requests plus a small number of writes to maintain the transaction log. We run the Sysbench load generator and the MySQL database with real-time priority and we add sixteen Unix `dd` processes in direct I/O mode with 64 KiB block size as background load. We use such a high number of processes, because `dd` performs synchronous I/O and therefore does not generate enough load on its own to impact the real-time application.

Figure 12 compares all schedulers for a read-only background load. The length of the bars indicate the completion latencies of database queries, where each query accesses the storage multiple times. Figure 13 presents the same comparison for a read/write background load, where eight `dd` processes are reading and one `dd` is writing to a file. Since writes hit the Linux buffer cache and are persisted asynchronously, a single `dd` generates enough load. Both experiments support our claim that K2 is able to trade throughput of background applications for improved response times of latency-critical applications, not only for micro-benchmarks, but also for complex database workloads.

VI. Related Work

There is a broad range of existing work on I/O scheduling, so we group works related to K2 into three categories: research with an approach similar to our work-constraining scheduling, current works on SSD and NVMe scheduling for off-the-shelf devices, and finally ideas for modifications of drive internals to benefit applications with timing requirements.

A. Work-Constraining Approaches

At the heart of work-constraining scheduling is the idea of restricting the choice of a second-tier scheduler by exposing

[6]https://github.com/akopytov/sysbench

fewer requests to it. In the case of K2, the second-tier scheduler is located within the NVMe drive controller's firmware, which independently decides in which order requests available in the NVMe hardware queues are fulfilled. By constraining the amount of requests in these queues, the maximum wait time for any request in the queue is reduced.

Reuther et al. pioneered this concept for spinning disks in their Dynamic Active Subset work [20]. Here, the second-tier scheduler is either the NCQ implementation within the disk or an OS-level throughput-optimizing disk scheduler like SATF [4]. The Dynamic Active Subset has been revisited by Povzner et al. for the Fahrrad scheduler [18], which uses a different reservation mechanism based on RBED [3], but a similar enforcement approach called Disk Scheduling Set. Fahrrad also targets spinning disks.

TimeGraph [11] applies work-constraining to GPU scheduling: real-time GPU workloads are prioritized by delaying the dispatching of other workloads to the GPU subsystem.

B. Scheduling Off-the-Shelf SSDs

Kim et al. compiled one of the first analyses of SSD behavior and designed an I/O scheduler from these findings [13]. They focus on write requests, because on these early SSDs, write latencies were erratic due to garbage collection. For modern NVMe devices, the picture has changed, so these considerations no longer apply.

FIOS [17] is a more recent work, but still targets the single-queue Serial-ATA block layer architecture of Linux and consequently compares against pre-NVMe Linux I/O schedulers. FIOS' design is similar to the BFQ algorithm [24], which was published in the same year. BFQ is NVMe compatible, so we compared K2 against BFQ rather than FIOS. To the best of our knowledge, no comprehensive survey of existing I/O schedulers has been conducted for NVMe devices, which prompted us to do so in this paper.

Other works try to increase overall SSD performance by optimizing request data alignment [14] or database indexing structures on flash-equipped sensor devices [5]. The MAP+ I/O scheduler [8] reorders I/O requests to use the page translation cache within the drive's FTL implementation more efficiently. These ideas rely on intimate knowledge of internal device parameters.

C. Changes to SSD Controllers

In order to improve end-to-end behavior, some researchers propose changes to the FTL controller firmware. Qin et al. improve garbage collection latency by splitting collection operations into multiple steps, which execute within the slack of real-time read and write requests [19]. Their approach guarantees an upper bound for all request completion times, but requires $2\times$ over-provisioning in storage size.

Instead of splitting operations temporally, PartFTL [15] partitions flash storage spatially. Read and write operations target different flash chips to ensure that reads are never blocked by writes. To implement the request partitions, PartFTL relies on 25 % storage over-provisioning.

Other works like AutoSSD [12] and FlashShare [25] do not offer hard guarantees. Like K2, they reduce tail latencies to improve quality of service. AutoSSD schedules device-internal background jobs like garbage collection, read scrubbing, and mapping management so that they do not harm application requests. FlashShare introduces an end-to-end notion of request prioritization, which the system can use to express application requirements to the SSD firmware.

A third group of works focusses on improving inter-application fairness without reducing overall throughput. Jun and Shin investigate fairness and accounting for SSDs with single-root I/O virtualization (SR-IOV) [10]. Tavakkol et al. reduce inter-application unfairness with an interference-aware scheduler [23]. K2 deliberately ignores fairness, but allows system designers to control request latencies of real-time applications at the expense of reduced service for background load.

VII. Conclusion

We have presented a survey of existing NVMe I/O schedulers and found all of them lacking in terms of tail latencies and performance isolation. Therefore, we designed and implemented K2, a new I/O scheduler for Linux, specifically targeting NVMe-attached storage. It is based on the concept of work-constraining scheduling: K2 constrains the scheduling opportunities of the second-tier scheduler located within the drive controller. In doing so, it trades reduced throughput for lower tail latencies. K2 reduces read latencies up to $10\times$ and write latencies up to $6.8\times$ in the 99.9th percentile, while penalizing throughput for non-real-time background load by at most $2.7\times$.

K2 operates with off-the-shelf drives and can therefore be deployed today. It is lightweight and device-agnostic, but only provides empirical improvements and no hard upper bound. To fix this, combining K2 with the existing work on real-time capable flash controllers should prove beneficial. Also, K2 currently restricts the second-tier scheduler by a fixed number of requests. In the future, we want to consider a dynamic constraint depending on upcoming real-time load.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their helpful suggestions. This work was supported by public funding of the federal state of Saxony/Germany. The research and work presented in this paper has also been supported in part by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance", in *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)*, Boston, MA, USA: USENIX, Jun. 2008, pp. 57–70.

[2] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: Introducing multi-queue SSD access on multi-core systems", in *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, Haifa, Israel: ACM, Jun. 2013, 22:1–22:10.

[3] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes", in *Proceedings of the 24th Real-Time Systems Symposium (RTSS)*, Cancun, Mexico: IEEE, Dec. 2003, pp. 396–407.

[4] P. J. Denning, "Effects of scheduling on file memory operations", in *Proceedings of the AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey: ACM, Apr. 1967, pp. 9–21.

[5] A. J. Dou, S. Lin, and V. Kalogeraki, "Real-time querying of historical data in flash-equipped sensor devices", in *Proceedings of the 29th Real-Time Systems Symposium (RTSS)*, Barcelona, Spain: IEEE, Nov. 2008, pp. 335–344.

[6] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3D XPoint technology", *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, Sep. 2017.

[7] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-queue fair queuing", in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, USA: USENIX, Jul. 2019, pp. 301–314.

[8] C. Ji, L.-P. Chang, C. Wu, L. Shi, and C. J. Xue, "An I/O scheduling strategy for embedded flash storage devices with mapping cache", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 4, pp. 756–769, Apr. 2018.

[9] K. Joshi, K. Yadav, and P. Choudhary, "Enabling NVMe WRR support in linux block layer", in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Santa Clara, CA, USA: USENIX, Jul. 2017.

[10] B. Jun and D. Shin, "Workload-aware budget compensation scheduling for NVMe solid state drives", in *Proceedings of the 2015 Non-Volatile Memory System and Applications Symposium (NVMSA)*, Hong Kong, China: IEEE, Aug. 2015, pp. 19–24.

[11] S. Kato, K. Lakshmanan, R. (Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments", in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, USA: USENIX, Jun. 2011, pp. 17–30.

[12] B. S. Kim, H. S. Yang, and S. L. Min, "AutoSSD: An autonomic SSD architecture", in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, USA: USENIX, Jul. 2018, pp. 677–689.

[13] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drives", in *Proceedings of the 9th International Conference on Embedded Software (EMSOFT)*, Grenoble, France: ACM, Oct. 2009, pp. 295–304.

[14] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)", *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 636–649, May 2012.

[15] K. Missimer and R. West, "Partitioned real-time NAND flash storage", in *Proceedings of the 39th Real-Time Systems Symposium (RTSS)*, Nashville, TN, USA: IEEE, Dec. 2018, pp. 185–195.

[16] *NVM express*, Revision 1.3, NVM Express, Inc., May 2017. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf.

[17] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler", in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, USA: USENIX, Feb. 2012, pp. 155–170.

[18] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad", in *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, UK: ACM, Apr. 2008, pp. 13–25.

[19] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time flash translation layer for NAND flash memory storage systems", in *Proceedings of the 18th Real Time and Embedded Technology and Applications Symposium (RTAS)*, Beijing, China: IEEE, Apr. 2012, pp. 35–44.

[20] L. Reuther and M. Pohlack, "Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS)", in *Proceedings of the 24th Real-Time Systems Symposium (RTSS)*, Cancun, Mexico: IEEE, Dec. 2003, pp. 374–385.

[21] *Samsung V-NAND SSD 970 evo*, Revision 1.0, Samsung, 2018. [Online]. Available: https://www.samsung.com/semiconductor/global.semi.static/Samsung_NVMe_SSD_970_EVO_Data_Sheet_Rev.1.0.pdf.

[22] O. Sandoval, *blk-mq: Introduce Kyber multiqueue I/O scheduler*, Commit message for the Linux kernel, 2017. [Online]. Available: https://patchwork.kernel.org/patch/9672023 (visited on 08/27/2019).

[23] A. Tavakkol, M. Sadrosadati, S. Ghose, J. S. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu, "FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives", in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, USA: IEEE, Jun. 2018, pp. 397–410.

[24] P. Valente and M. Andreolini, "Improving application responsiveness with the BFQ disk I/O scheduler", in *Proceedings of the 5th International Systems and Storage Conference (SYSTOR)*, Haifa, Israel: ACM, Jun. 2012, 6:1–6:12.

[25] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs", in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, USA: USENIX, Oct. 2018, pp. 477–492.