Fairness, Isolation, Predictability and Performance Management in NVMe Devices: A Survey

Brynjar Ingimarsson
Vrije Universiteit Amsterdam
b.ingimarsson@student.vu.nl

Abstract

Modern NVMe devices are capable of millions of IOPS in throughput and low microsecond latencies. However, it remains a challenge to consistently provide peak throughput, low latencies, and fairness simultaneously when a device is shared between multiple tenants. In addition, modern NVMe devices depend on advanced controller logic that leads to behavior that can be hard to predict. This survey explores the challenges of providing QoS guarantees on NVMe devices and current solutions. We explore solutions across the storage stack, on the device controller, the OS block layer, and higher layer solutions in virtualized- and container environments.

1 Introduction

Solid State Drives (SSDs) are a popular means of storage in modern server, desktop, and mobile applications. Compared to traditional Hard Disk Drives (HDDs), they provide orders of magnitude higher throughput and lower latency. A modern SSD can offer latencies on the order of 10 µs and a throughput of 1 million IOPS [14]. In recent years, host interfaces and storage stacks have been rearchitected to better utilize SSD capabilities, for example with blk-mq [9] and NVMe [2].

However, providing good performance does not come without challenges. First, SSDs come with complex flash controllers that must perform tasks such as garbage collection, wear-leveling and address translation [50]. The cost of these tasks often interferes with requests from the host, resulting in QoS problems such as unpredictable behavior, lower throughput, and high tail latencies [72]. Second, when multiple tenants share a device, the predictability of each tenant's performance further degrades [36, 64]. As we will explore in this survey, designing SSD controllers and software stacks that can provide fairness between tenants without impacting performance is a big challenge.

Predictable performance and fairness are desirable in many applications. Database systems want to guarantee a minimum number of queries per second, or hypervisors guarantee a certain minimum bandwidth to all virtual machines. Such guarantees are also known as QoS (Quality of Service). The goal of this survey is to explore QoS challenges and solutions on NVMe devices. In this survey, we define QoS as the following goals.

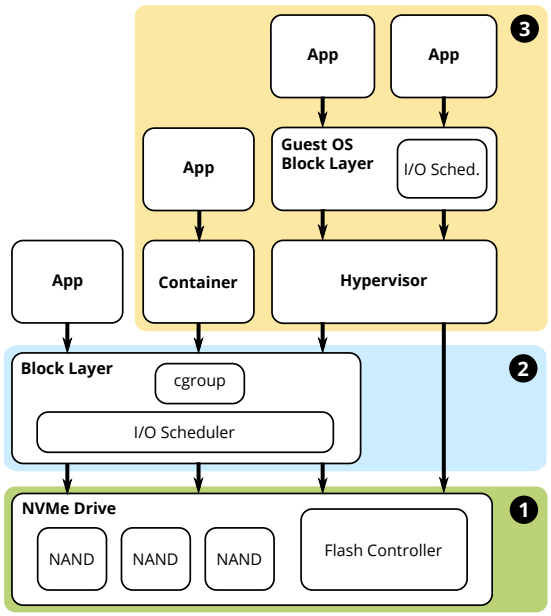

Figure 1: NVMe storage stack with example I/O paths. The survey explores solutions at the three highlighted layers.

G1 **Fairness** - The device should be shared fairly, i.e. all tenants should receive a proportional share of throughput and latency.

G2 **Isolation** - The performance of each tenant should be isolated. A tenant should not be able to degrade the throughput of other tenants or cause latency to spike.

G3 **Predictability** - Latency and throughput should be stable in both single- and multi-tenant settings, unpredictable

spikes are undesirable.

G4 **Performance** - It should be possible to saturate the device, i.e. the combined throughput received by tenants should be near the capabilities of the device.

The cloud computing market was valued at US$718 Billion in 2022 [1]. NVMe storage is widely used in the cloud, where both providing reliable service and using resources cost-effectively is of great importance. There is currently active research in improving QoS with I/O schedulers [24, 27, 51, 69], SSD controllers [34, 64, 72], and storage virtualization [55, 56, 75]. However, to the best of our knowledge, no surveys have been published on QoS in Solid State Drives. This motivates us to compile a survey on QoS in NVMe storage environments.

Figure 1 shows examples of paths that application I/O can take. In the simplest case, an app runs as an OS process and the OS I/O scheduler can provide QoS. Applications running in containers can make use of cgroup policies for QoS, and finally, some hypervisors bypass the host block layer, and the guest OS may be running its own scheduler. In all scenarios, the device-side flash controller can also make QoS decisions. In the following sections, these scenarios will be further explored.

The survey is organized as follows: In Section 3 we provide background on recent advancements in storage, flash device internals, new interfaces, and how QoS is provided. In Section 4 we discuss the current challenges with providing QoS on flash storage that motivate this survey. We then explore solutions at different layers as seen in Figure 1. We look at ❶ device-side solutions in Section 5, ❷ block-layer solutions in Section 6, and ❸ virtualization / containerization solutions in Section 7. Lastly, in Section 8 we look at emulation solutions to explore new storage stack designs. We summarize future work in Section 9 and finally we draw our conclusion in Section 10.

2 Survey Design

In this section we define the goal of the survey and present several research questions. We also define the scope of the work and the methodology used to find research papers.

2.1 Survey Goals

As mentioned in the introduction, the goal of this survey is to investigate the challenges in providing QoS guarantees on NVMe devices when the device is shared with multiple tenants. The main survey question is: *What are the challenges in providing fairness and reliable performance to multiple applications on modern SSDs?* We also ask the following sub-questions to help us understand various aspects of the larger problem.

RQ1 How can I/O schedulers preserve peak throughput on SSDs while providing other QoS guarantees?

RQ2 What are the causes of performance degradation on SSDs and how can they be prevented?

RQ3 What factors limit QoS in SSD-based virtualized and container environments?

RQ4 How can the design space of SSDs and block layers be explored with the goal of improving QoS?

2.2 Scope

In this survey, we focus on work that is specific to NVMe devices. Research that is aimed at HDDs or scheduling in general is not included unless we consider it important background knowledge. There is also extensive research on improving the design of SSD controllers, but we do not include it unless it contributes to understanding QoS aspects. For a paper to be included, it must satisfy all of the following inclusion criteria.

I1 The paper explores one or more of the QoS goals that we defined.

I2 The paper proposes a new design of an I/O scheduler, SSD controller or another storage stack component, or investigates QoS properties of existing solutions.

I3 The paper specifically targets NVMe devices.

I4 The paper is from around 2010 or later to keep our work time bound.

2.3 Methodology

We use several methods to find papers to include in the survey. We first start with several seed papers and use the Snowball methodology [68] to find related papers by looking at references in both directions. The seed papers are shown in Table 1. We also look at several conferences that include storage-related topics and look for papers that satisfy the inclusion criteria. The conferences considered are FAST, HotStorage, USENIX ATC, SYSTOR, ODSI and EuroSys. We look through all conferences from 2018 to 2023 and look for relevant papers. Finally, we do a manual search using several relevant keywords.

3 Background

Providing quality of service on NVMe devices comes with challenges at many layers of the storage stack. This section provides the relevant background on the storage stack and recent advancements. Subsection 3.1 discusses storage in the era of hard disk drives. Subsection 3.2 explains the internals of

Category	Paper	Source	Year
Interfaces	ZNS [8]	ATC	2021
	Open-Channel [10]	FAST	2017
	KAML [30]	HPCA	2017
	SALSA [28]	MASC.	2018
I/O sched.	D2FQ [69]	FAST	2021
	blk-switch [27]	OSDI	2021
	A+CFQ and H+BFQ [37]	IEEE	2019
Flash	FLIN [64]	ISCA	2018
	FlashBlox [26]	FAST	2017
VMs	MDev-NVMe [56]	ATC	2018
	SPDK Vhost-NVMe [75]	SC2	2018
Emulation	QEMU NVMe Driver [20]	UCC	2018
	FEMU [44]	FAST	2018
	NVMeVirt [39]	FAST	2023
	MQSim [63]	FAST	2018

Table 1: The seed papers that were used to find papers for this survey (with shortened titles)

modern SSDs and their design space. Subsection 3.3 goes into the details of host-to-device storage interfaces. Subsection 3.4 goes over how the Linux storage stack has been redesigned for the Flash age. Subsection 3.5 explains the main ideas of I/O schedulers and their evolution on Linux, and finally, Subsection 3.6 provides a background on queueing theory and fair queueing algorithms.

3.1 Hard Disk Drives

Hard Disk Drives (HDDs) have been the primary type of storage for decades. Thus, the design of operating systems, file systems and applications has largely been based on their characteristics. An HDD consists of one or more spinning platters, each with a number of tracks and each track with a number of sectors (typically 512 bytes). On a read / write operation, a mechanical disk head moves to the track containing the sector, known as seek time. It then waits until the sector is underneath the head, known as rotation delay.

HDD performance is limited by the rotation rate of the platters, typically around 7,200 - 15,000 RPM. The latency of an I/O operation is around 1 - 20 milliseconds, sequential throughput around 50,000 IOPS and random throughput around 250 IOPS [17]. The high cost of seeking makes random operations highly unfavorable. HDDs continue to advance but mainly in capacity, throughput continues to grow slowly, but latency has been stagnant for decades [15, 52].

These characteristics have made system and application programmers put great effort into minimizing I/O and making operations as sequential as possible [7]. For example, operating system I/O schedulers would merge and reorder operations using an elevator algorithm, as explained in Subsection 3.5.

3.2 The Advent of NVM Drives

Non-Volatile Memory (NVM) is a type of memory that is semiconductor-based and persists after losing power. A popular type is NAND flash, which has been available since the 1980s, but could not compete with the price and capacity of HDDs until recently [13]. Solid-State Drives (SSDs) typically consist of NAND flash chips along with a controller. In Section 4 we describe the challenges that SSDs bring with regards to QoS.

SSD Internals

A NAND flash chip contains one or more dies, each of which contains multiple planes (Figure 2). Each plane contains multiple blocks and a single register. A block contains multiple pages, typically 4 kB large. Read operations work on page granularity and can access any page in the chip. A write operation also works on page granularity but can only write pages that have been erased. An erase operation works on block granularity and zeroes out all pages in the block. Finally, the chips are rated for a limited number of erase cycles. A typical NAND chip might offer 25 µs reads, 200 µs writes, 1.5 ms erases and 100,000 erase cycles [3].

The flash chip has data lines to transfer data in or out of plane registers, and control lines to send commands. The time to transfer data may be lower than the operation time, making it possible to interleave multiple operations. However, interleaved operations must operate on different planes. An SSD consists of multiple flash chips along with a controller, DRAM for mapping data and host interface logic. Multiple flash chips can share data lines in units which are called channels, as seen in Figure 3.

One of the largest constraints of NAND flash is that blocks must be erased to make pages writable. The block interface exposed to the host allows writes to happen at any address at any time. To make writes efficient, the FTL (flash translation layer) maps between logical block addresses (LBA) exposed to the host, and physical block addresses within the device. When a write operation arrives, the device selects a free page and updates the mapping table. However, this means that writes do not happen in place and older pages turn into zombie pages that must be garbage collected. Selecting a free page should also aim to improve the parallelism and wear-leveling of the device, i.e. making sure that all parts of the device age uniformly [3, 50].

Garbage Collection

When the same LBA is written multiple times, the writes do not happen in place and the old data still exists in pages known as zombies, because nothing is mapped to them. To reclaim this space, garbage collection is required. The garbage collector will copy all active pages from a block into a new block and then erase the old block. Designing an efficient

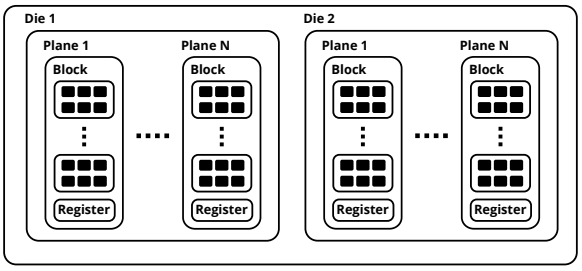

Figure 2: Internal structure of a NAND flash chip.

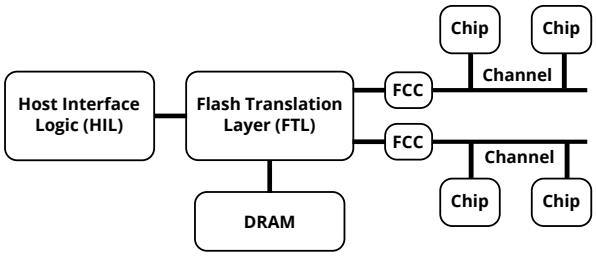

Figure 3: Structure of an SSD with multiple channels for NAND chips.

SSD garbage collector has been researched extensively. For example, a simple garbage collector might be triggered as soon as the ratio of active pages to zombie pages (known as cleaning efficiency) reaches a threshold.

Garbage collection is expensive, as it consists of an erase operation and multiple read and write operations. Depending on the controller design, the GC can block the plane, chip or entire device for other operations. Because GC can block requests from the host, this can lead to high tail latencies and unpredictable performance for applications. Some flash chips support copying data within a plane. If the GC selects a new block within the same plane, no data will cross the channel, which can improve performance. The ratio of writes issued by the host and the total number of writes including GC is known as the Write Amplification Factor (WAF), and a lower value indicates more efficient use [50, 72].

Data Placement

A single flash chip has limited bandwidth, so it is essential to exploit all the channels in parallel in order to achieve good performance. A mapping policy determines how operations are distributed across the channels. The two most commonly used policies used in practice are (i) LBA-based mapping, where LBAs are striped across channels, i.e. a given LBA is always mapped to channel LBA mod N, where N is the number of channels, and (ii) write-order-based mapping where incoming writes are striped across channels, i.e. write W_i goes to channel i mod N, where W_i is the i-th write. Depending on the mapping, some access patterns may cause operations to be spread unevenly across channels, crashing performance [11].

3.3 New Hardware Interfaces

Most early SSDs had a SATA or SCSI interface, which are designed for HDDs. As the latency and throughput of SSDs improved, these interfaces proved insufficient. For example SATA 2.0 has a maximum bandwidth of 600 MB/s, and devices are connected through an HBA (host bus adapter) which introduces latency (Figure 4) [70]. In addition, SATA devices can only have a single outstanding request, but SATA devices with NCQ (native command queueing) can have a queue depth of 32.

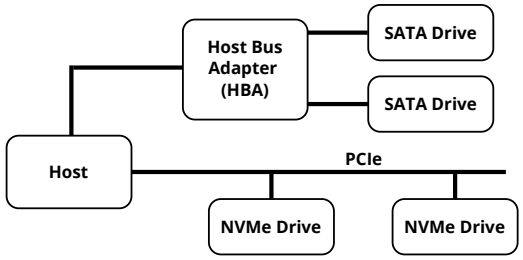

Figure 4: Comparison of NVMe and SATA host interfaces.

The NVMe protocol is designed for high-throughput SSDs and allows SSDs to attach directly to the PCIe bus. NVMe allows setting up multiple pairs of I/O submission and completion queues to fully utilize the internal parallelism offered by SSDs. The queues have a maximum depth of 64k and the standard allows up to 64k queues, however drives typically support 8-128 queues. The host may set up a queue pair for each application, or like the Linux block layer, for each core.

The mainstream storage interfaces present a block interface. Blocks are typically 512 B - 4 KB large and can not be partially read or written, but may be accessed in any order. There have been many proposals for alternative SSD interfaces. Bjorling et al. [8] argue that a block interface puts a tax on SSD performance, caused by the internal logic needed to abstract flash erase blocks. They introduce ZNS, which exposes zones that must be written sequentially. ZNS devices do not need internal garbage collection or large translation tables, thus performance is more predictable. However, the end performance depends on how well applications can adapt to the sequential write-order criteria. SALSA [28] implements an FTL on the host and eliminates the device FTL overhead by only writing large sequential chunks.

Other interfaces include Open-Channel SSD [10], which offers even less abstraction than ZNS by exposing the complete channel/chip/plane organization to the host. KAML [30] exposes a key-value interface and implements the key-value store on the SSD controller. Finally, multi-streamed SSDs [33] allow applications to specify the expected lifetime of blocks.

The blocks are arranged such that erase blocks contain data with a similar lifetime, improving the efficiency of garbage collection.

3.4 Advances in the (Linux) Storage Stack

In the era of HDDs and the single-queue Linux block layer, the journey of an I/O operation would look as follows. An application makes a standard read or write syscall, causing a context switch into the kernel, and the data is copied to kernel space. The virtual file system looks up which file system the file belongs to, and then asks the file system for the physical address that corresponds to the offset within that file. Then, a block I/O request (`struct bio`) is submitted to the block layer which stages the bio's in a request queue. While staged in the queue, an I/O scheduler can merge adjacent requests, reorder requests (e.g. to make them more sequential), and do accounting. Requests are then dispatched from the queue to the driver of the storage device. When the device has finished, a hardware interrupt triggers an IRQ routine, and the user process can be scheduled again.

This design could handle thousands of IOPS, but not millions. In this design, there is a single request queue for each storage device protected with a lock. When bio requests are submitted from multiple cores, contention on the lock increases and performance declines. In addition, all hardware interrupts used to be handled on the same core [9].

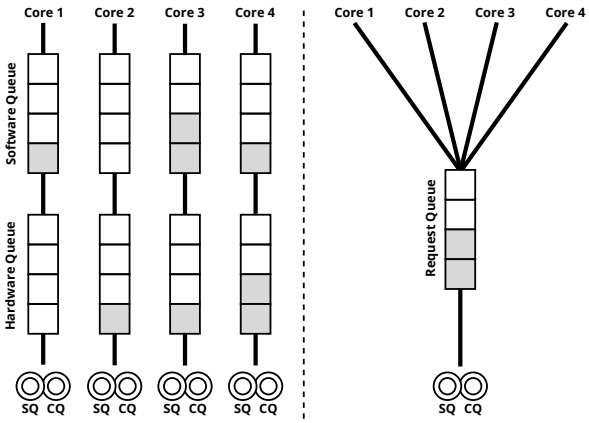

Figure 5: Comparison of single-queue (right) and multi-queue (left) block layer designs.

A Multi-Queue Block Layer

To make the Linux block layer future-proof for the next generations of NVM devices, a new multi-queue design was introduced (known as `blk-mq`) in Linux 3.13 (2014). Instead of a single global queue for all I/O requests, there are now a configurable number of software queues, commonly one per CPU core (Figure 5). In addition, there are a number of hardware dispatch queues to match with devices that offer multi-queue support. The software queues act as a staging area for I/O requests. I/O schedulers can be plugged in during runtime and they can reorder requests within each software queue. The hardware queues are responsible for managing back-pressure on the device, i.e. that the device does not get more requests than its queues can hold. The hardware queues do not allow modifications, requests are added from SW queues to the tail and the device driver consumes from the head of the queue. This separation of functions between two queues, and having a software queue on each core allows for better performance scalability. The performance of `blk-mq` grows roughly linearly with the number of cores [9].

Polling Instead of Interrupts

Traditionally, hardware interrupts have been the only mechanism for finishing an I/O request. With interrupt-driven I/O, the OS scheduler puts the issuing process in a waiting state, and schedules another task once the I/O submission has completed. The device issues an interrupt when the request has finished, causing a context switch into the interrupt handler of the driver, and the issuing process becomes schedulable again.

This works well for HDDs as milliseconds of CPU time is not wasted, and interrupts are relatively few and fast. For NVM drives that can reach millions of IOPS and latencies in the order of microseconds, the number of interrupts can become so high that it saturates the CPU core. In addition the end-to-end latency seen by the application may be worse due to more context switches and delay until it is scheduled again.

An alternative to interrupts is polling, where after submitting I/O, the kernel stays in a busy loop and constantly polls the device. The kernel then switches back to the same process once the I/O is completed, resulting in fewer context-switches and no delay. Polling might thus offer lower latency and higher throughput for modern devices, at the cost of wasting CPU cycles in a busy loop [73].

Rethinking the Kernel Interface (io_uring)

As described before, I/O was typically done through blocking system calls such as read or write or the vectorized variants preadv and pwritew. Linux also has an async interface, called aio, but it has limitations and is not considered a success. Linux 5.1 (2019) introduced io_uring [6], an async I/O interface that aims to be future-proof, easy to extend, and provide better performance, for example by minimizing copying, the number of context switches, and supporting polling.

io_uring offers three modes of passing submission and completion events between kernel and application.

Interrupt driven - The application puts one or more submission events in the SQ and calls the `io_uring_enter`

syscall to notify the kernel. The call is async and returns immediately, When the storage device interrupt routine is triggered, an event will be added to the corresponding CQ which will become visible to the application.

I/O polling - The uring is initiated such that the storage device will not throw interrupts. Instead, the application must call `io_uring_enter`, which will poll the device in a busy-loop and return when a specified number of events have completed.

SQ polling - This creates a kernel thread that polls the SQ for new events. The application then does not have to call `io_uring_enter` to submit events, making the operation almost free of context switches.

The modes are not mutually exclusive, e.g. when using SQ polling, the completion side can either be interrupt driven or use I/O polling. Using both polling options has been shown to give the best performance, but at the cost of higher CPU usage [16]. The SQ polling kernel thread requires a dedicated CPU core for optimal performance, as descheduling it will block all submission events.

Most system calls, including read and write, will copy data from the user's buffer into the kernel. To avoid copying, io_uring uses a pair of ring buffers (Figure 6) that are shared between userspace and kernel, one for I/O submissions and one for I/O completions. To submit I/O, the application looks at the SQ head pointer and places a request at that location in the ring buffer. When the kernel consumes from the queue, it will update the tail pointer to the index where it should next consume. If the head and tail pointers are the same, there is nothing to consume. The completion queue works similarly but with switched roles. Ring buffers are good for performance as they are lock-free data structures. They do however need memory fences to ensure memory operations are ordered, as modern CPUs may otherwise execute them out of order, leading to incorrect execution.

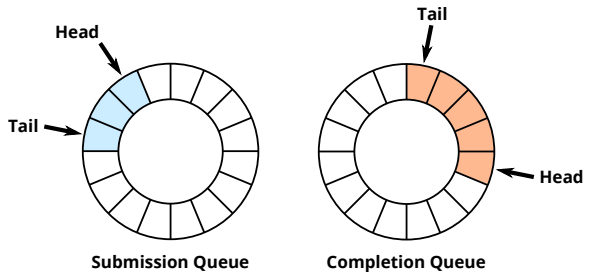

Figure 6: io_uring uses ring buffers for the submission and completion queues.

Bypassing the Block Layer (SPDK)

For high-performance storage applications, it may be worth it to bypass the I/O facilities provided by the OS. SPDK is one such solution, providing an NVMe driver that runs in userspace. The NVMe device is thus not managed by the OS and is only usable by a single application. The I/O path becomes shorter as there is no scheduling and fewer abstractions, and SPDK has been shown to perform better than the kernel block layer. SPDK always runs in polling mode, as forwarding interrupts to a user process is difficult and expensive [16, 58].

3.5 Evolution of I/O schedulers

The purpose of an I/O scheduler is to order and modify I/O requests before they are submitted to a storage device, e.g. in order to improve performance or application experience. The goal of I/O schedulers can be

- **Fairness** - Ensure that multiple processes get a fair share of I/O submitted, the processes do not degrade the performance of others and no process becomes starved.
- **Performance** - Reorder and merge requests in order to produce a better performing access pattern, for example by ordering requests in sequential order for HDDs.
- **Prioritization** - A scheduler may support giving different priorities to I/O requests.
- **Deadlines** - A scheduler can provide a deadline guarantee, i.e. guarantee that a request will be finished within a certain timeframe after it was submitted.

Note that these goals intersect with the QoS goals that we have defined. In this subsection we will describe how existing I/O schedulers have evolved to provide these goals, focusing on I/O schedulers on Linux.

Elevator Scheduling - Elevator schedulers attempt to minimize seek times by ordering requests by sector (LBA) in sequential sweeps. For example, the SCAN algorithm starts by dispatching its first request, then only dispatching requests with higher (or lower, depending on direction) LBAs. Other requests will be scheduled for the next sweep, in sequential descending (or ascending) order. The variant C-SCAN always sweeps in one direction, as going back and forth statistically gives more priority to sectors in the middle [15].

The Linus Elevator - The first I/O scheduler on Linux was an elevator named after its creator. It uses a SCAN-ordered queue and also merges incoming requests with adjacent requests in the queue. It also includes a mechanism to prevent starvation. When inserting a new request, it will not be inserted in front of a request that is older than a given threshold, where age is measured as the number of

succeeding requests. However, this maximum age guarantee does not sufficiently prevent starvation [7].

Deadline - To improve I/O scheduling, Linux 2.6 (2004) introduced support for hot-pluggable I/O schedulers, with the default being the new Deadline scheduler. To limit starvation, every request is given an expiry time, default 500 ms for reads and 5 s for writes. A pointer to the request is stored in two queues, one sorted by LBA (red-black tree) and one by expiry time (FIFO queue), with separate queue pairs for reads and writes (Figure 7). Finally, requests are moved to the dispatch queue in batches. Requests are moved from the read queues unless there are only writes, or if writes have been ignored too many times. It then takes requests from the LBA sorted queue, unless the head of the expiry queue has expired, in which case it moves that request and adjacent requests from the sort list, keeping the batch sequential [7, 59].

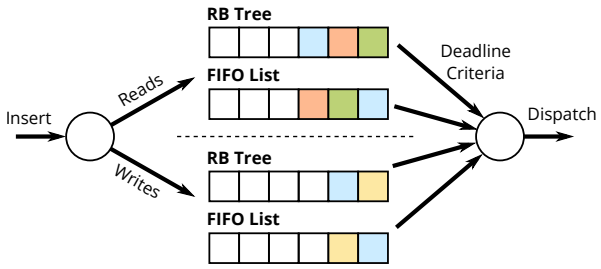

Figure 7: Queue structure of the deadline scheduler.

Anticipatory - The anticipatory scheduler is based on the deadline scheduler, but includes a few additional heuristics. Most notably it collects statistics about the I/O pattern of each process, and when dispatching the last request for a given process, will try to anticipate whether a new request is likely to come soon. This blocks potential requests from other processes, typically for a few milliseconds, but the prevented seeks may provide a greater benefit. Figure 8 shows such a scenario. An I/O scheduler that does not dispatch any work even though it is available is said to be non-work-conserving, and the anticipatory scheduler is one such example [46, 57].

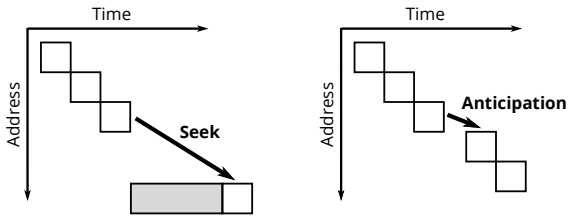

Figure 8: A scenario where anticipation improves performance. Dispatching an available request (gray) is more expensive than waiting for more sequential requests.

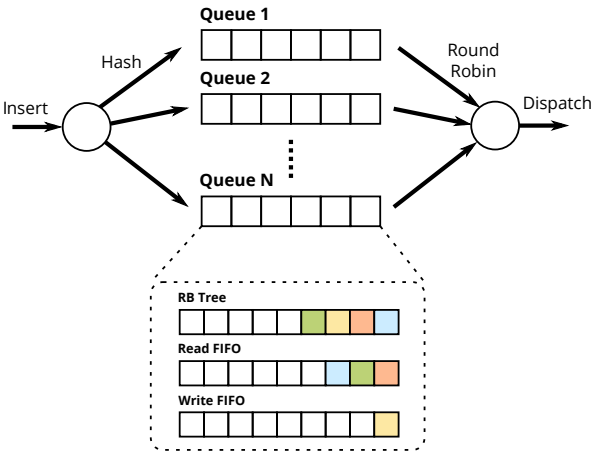

Figure 9: Queue structure of the CFQ scheduler.

CFQ - The first Linux I/O scheduler that aims to provide fairness between competing processes. It has a high number of queues (default 64) and requests are placed on a queue determined by hashing the process PID (Figure 9). Requests are then moved in batches to the dispatch queue in a round-robin manner. It was later improved by giving time slices to each queue rather than moving fixed batches, allowing a queue to sit idle and wait for more requests, similar to the anticipatory scheduler. It also introduced per process I/O priorities, which default to the process CPU priority level. Compared with the deadline scheduler, CFQ can provide better throughput and fairness, but latency may be higher [5, 46].

3.6 Queueing Theory

Although originally invented for network switches, fair queueing algorithms can be used to build I/O schedulers. Fairness is a metric of how proportionally the capabilities of a device are shared between multiple tenants. Assume that we have n tenants (T_1, T_2, ... , T_n). A tenant that runs alone on the device uses bandwidth $BW_{alone}(T_i)$, and the device is capable of BW_{total}. When all tenants run together, the bandwidth of each tenant is $BW_{shared}(T_i)$. We can then define the slowdown of each tenant as

$$S(T_i) = \frac{BW_{alone}(T_i)}{BW_{shared}(T_i)} \quad (1)$$

In a completely fair scenario, all tenants should experience the same slowdown. Thus, we can define the fairness F as the ratio between the greatest and lowest slowdown.

$$F = \frac{\min_{\forall t \in T_i} S(t)}{\max_{\forall t \in T_i} S(t)} \quad (2)$$

The fairness value is always between zero and one. We have $F = 1$ in a completely fair scenario and a lower value means

more unfairness. Note that this defines fairness in terms of throughput, however we also need to consider it in terms of latency. In that case, slowdown is the ratio of average latency when running alone over average latency when shared.

Example - A device is capable of $BW_{total} = 1.000$ MB/s. Tenant 1 wants full throughput $BW_{alone}(T_1) = 1.000$ MB/s and tenant 2 wants $BW_{alone}(T_2) = 250$ MB/s. To maximize fairness, both tenants should be slowed down by $S = 1.25$ and receive 800 MB/s and 200 MB/s respectively.

This is a simplified example, in reality the total bandwidth is not fixed but depends on data direction and access pattern, as we will explore in the next section.

Fairness scheduling can be done with different techniques. Linux's CFQ and BFQ schedulers use a timeslice approach, and so do Argos and FIOS, providing fairness in terms of device time. FLIN uses virtual time to order a single queue by the estimated departure time of each request if it was alone in the queue. WA-BC uses deficit-round-robin, where each tenant has a queue and requests are dispatched in round-robin until a per-tenant budget runs out. Lottery scheduling uses a probabilistic method to achieve fairness.

WFQ

Weighted Fair Queueing (WFQ) [53] is an algorithm that guarantees fair service to multiple queues on a shared resource (each queue might represent one tenant). The queues contain variable-sized tasks (e.g. packets or I/O requests) and the resource can consume one task at a time. The queues can also have different weights to control the proportionality.

When a task t_i is enqueued, it is assigned a virtual start time T_{start}, which is either the virtual finish time of the succeeding task t_{i-1} in the queue, or the global virtual time T_{global} if the queue is empty (Equation 3). The task's virtual finish time is the virtual start time plus the cost of the request, which accounts for the task size and queue weight as a fraction of the device capacity C (Equation 4). The global virtual time is the total service of the device, e.g. number of bytes processed.

$$T_{start}(t_i) = \max(T_{global}, T_{finish}(t_{i-1})) \tag{3}$$

$$T_{finish}(t_i) = T_{start}(t_i) + \frac{W}{\sum W_j} \cdot C \cdot \text{size}(t_i) \tag{4}$$

When the current task has finished, WFQ will look at the heads of all queues and choose the task with the lowest virtual finish time. This ensures that no queue's progress grows out of proportion, thus ensuring fairness. As WFQ always dispaches work if any queue is non-empty, it is a work conserving algorithm. Another characteristic is that queues accumulate shares when they are backlogged, but when they become empty, all shares are forfeited, ensuring fairness with bursty workloads.

WF2Q

Worst-case Fair Weighted Fair Queueing (WF2Q) [77] is an extension of WFQ that provides the same guarantees but aims to prevent bursts when queues have different weights. By adding a requirement that a task's virtual start time must be lower than the current global virtual time, all queues will move forward more equally, producing a more smooth workload. Figure 10 shows scheduling for 4 queues where the last queue has weight 3 and the others 1. In WFQ, queue 4 is dispatched in periodic bursts (yellow), while in W2FQ, all queues receive a smooth service.

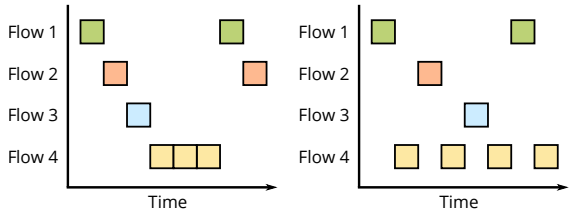

Figure 10: A comparison of schedules produced by WFQ (left) and WF2Q (right).

SFQ

Start-Time Fair Queueing (SFQ) [19] is also based on WFQ but dispatches requests in order of start times rather than finish times. SFQ has been shown to provide better fairness for devices where performance is non-fixed. Unfairness can occur when the device completes requests slower or faster than the configured cost and the virtual time drifts ahead or behind. That time is used to tag new requests, but backlogged queues may have diverged times. SFQ(D) is an extension of SFQ that allows dispatching D requests in parallel, which is useful for SSD devices with high parallelism. SFQ(D) is used in MQFQ, FlashFQ, and vFair.

DRR

Deficit Round Robin (DRR) [61] is a computationally efficient fair-queueing algorithm that can select a task to dispatch in O(1) time rather than O(log N) like the other algorithms. It works similarly to weighted round-robin (WRR) but it accounts for different task sizes. In each round, each queue can submit tasks until a given quantum is exceeded, and if the queue stopped on a task larger than the remaining quantum, the unused quantum (deficit) is added to the quantum in the next round. If all tasks have equal size, DRR is equivalent to WRR. A downside of DRR is that service can be bursty and latencies can grow long. DRR is used in WA-BC.

2DFQ

2 Dimensional Fair Queueing (2DFQ) [49] is an advanced fair queueing algorithm that supports parallel dispatch and variable request costs. Similar to WF2Q, it aims to minimize the burstiness of service. When tasks have a high variance in cost, 2DFQ provides lower tail latencies compared to other schedulers. This may be useful for I/O requests where e.g. 4K requests compete with 256K requests.

4 Motivation

In this section we explore the challenges of providing QoS guarantees on SSDs that motivate this survey. We describe the unique characteristics of SSDs, as QoS mechanisms must be designed around them. We then describe interference which can be caused by these characteristics. Finally, we explore the challenges in software design and how additional work in the I/O path can decrease performance.

4.1 SSD Characteristics

Making performance guarantees on SSDs is challenging because their throughput, latency, and IOPS are highly dependent on the access pattern and device state. This is caused by the complex internal logic of SSDs, for example the FTL logic that is needed to expose a block device, but also due to the parallelism offered by the SSD's internal organization.

The first characteristic of SSDs is that flash read and write (program) operations take different times, and TSUs (transaction scheduling units) may reorder operations to prioritize reads. Another is that internal tasks such as garbage collection and wear leveling also need to access the flash chips and thus compete with host operations. The intensity of these operations depends on the device state. A device has reached steady-state when all pages have been written to at least once and the internal tasks have become stable.

The internal parallelism of SSDs also causes certain performance characteristics. For peak throughput, an I/O flow must make use of all channels. A bad access pattern may hit some channels more than others and thus introduce a bottleneck. If an SSD uses a cache for translation table entries, some access patterns may cause more evictions so translations take more time. Many SSDs also contain a write-back buffer, with varying flushing policies which can produce latency spikes. Any attempt to share an SSD and provide QoS will need to account for these factors [23, 45].

4.2 Interference

Because of the characteristics that we just explained, mixing operations from multiple workloads can have unexpected performance results. Two different workloads may show good performance when running in isolation, but significantly worse performance when interleaved. Interference can be caused by different SSD functions, e.g. the write-back cache, translation tables, and in per-channel transaction schedulers [63]. The authors of FLIN [64] identify the four following categories of interference on SSDs.

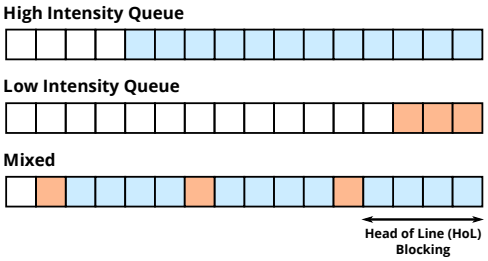

Figure 11: When high- and low-intensity workloads are mixed, the low-intensity transactions stay deeper in the queue and experience head of line blocking.

1. **I/O Intensity** - A high-intensity workload slows down a low-intensity workload. This is because when a low-intensity workload runs alone, the chip dispatch queues stay short and the waiting time is low. With a high-intensity workload, the queues are longer and thus transactions from the low-intensity workload get positioned deeper in the queue (Figure 11).

2. **Access Pattern** - A workload with a bad access pattern with respect to parallelism slows down a workload with a good access pattern. By not utilizing the parallelism, the workload causes congestion on a few flash channels or chips. Transactions from the good workflow will experience slowdown on those channels or chips, and the slowest transaction of an I/O command is what determines the latency.

3. **Read / Write Ratio** - A read-intensive workload slows down a write-intensive workload. Because write operations are much slower on flash (10-40x), existing schedulers prioritize reads over writes. This results in the slowdown of a write-intensive flow as there are more read requests being prioritized.

4. **Garbage Collection** - A workload that requires frequent garbage collection slows down a workload that is more GC friendly. Transactions from garbage collection are not scheduled with respect to which flow caused them.

Garbage collection is a frequent source of SSDs performance quirks and can affect interference. For example, consider two workloads. When running alone, A has a low WAF (write amplification factor) while B has a worse WAF. The resulting WAF of the combined stream is not the average of the

two, it can even be higher than the value of B, meaning that everyone loses [36]. Interference can also happen even if apps don't run concurrently, for example if a previous app leaves the device in an unfavorable state for the current app [20].

4.3 Software Overheads

Given the high performance of SSDs, software components such as the block layer must be able to withstand the throughput and not introduce additional latencies. For example BFQ, the most complex Linux scheduler, has been shown to perform poorly on fast SSDs. Recent studies argue that I/O schedulers are not needed anymore, as even light schedulers such as Kyber and mq-deadline introduce non-negligible latency (5-10%) and don't deliver peak throughput. However, this argument ignores the need for QoS and fairness [67].

Another study shows that without an I/O scheduler on Linux, the block layer still can not saturate an SSD using a single CPU core, as the CPU intensity is too great. Lightweight solutions such as SPDK require less CPU power to saturate a device. In addition, to achieve peak performance, modern techniques like io-uring and polling must be used [58]. However, this introduces challenges for hypervisors as these solutions can not easily be passed into a virtual machine.

5 Flash Device Controllers

In this section we explore solutions that propose changes to SSD firmware and architecture to improve QoS, corresponding to layer 1 in Figure 1. The solutions are summarized in Table 2, which will be discussed at the end of this section.

5.1 QoS-Aware Controllers

We first explore several solutions that implement QoS on the SSD controller, for example by implementing a fair queueing algorithm or an isolation scheme.

FLIN - The Flash-Level INterference-aware scheduler (FLIN) [64] is a transaction scheduler for SSDs that aims to provide fairness and performance. It accounts for four types of interference by reordering and prioritizing transactions within the per-chip TSU (transaction scheduling unit).

The FLIN scheduler consists of three steps that address different types of inference (Figure 12). When transactions arrive at the chip-level scheduler, they are inserted into the queue for their priority and direction (read or write) and each queue is ordered to optimize fairness. The second step does not address any inference problem, but implements priority awareness by using weighted-round-robin on the queues of each priority level. The third step addresses access pattern and GC inference by choosing one of four options from step 2, a read, write, GC read or GC write transaction.

Different I/O intensities are the most dominant source of interference, and thus step 1 provides most of the improvement to fairness. The key in step 1 is an algorithm that orders the queue to maximize fairness. It first checks whether the transaction belongs to a low- or high-intensity flow (default threshold is 32 MB/s for reads and 256 kB/s for writes). The queue consists of two regions for each intensity, so that low-intensity transactions are always in front. For each position in the region, the resulting fairness is computed using an efficient two-pass approach and the transaction is inserted in the optimal position.

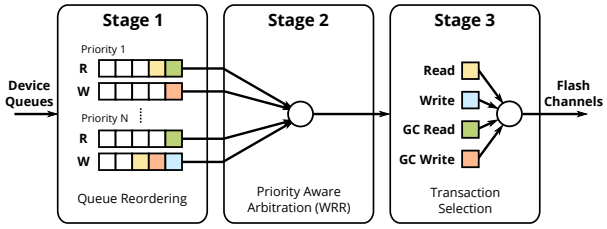

Figure 12: The three steps of the FLIN scheduler.

To address the other inference types, step 3 balances the wait time between read and write requests, and also ensures that GC transactions are distributed across flows. The algorithm computes the Proportional Wait Time (PWT) of the two available read and write transactions. The PWT indicates the resulting cost of the transaction if the other one is dispatched first. This ensures that write requests are not stalled for too long.

A benefit of FLIN is that it can be implemented within the SSD controller and requires no change to the host interface. It makes use of the multi-queue nature of NVMe to separate flows and the queue weight feature to set flow priorities. FLIN is implemented using the MQSim flash simulator and is shown to give better maximum slowdowns than existing solutions. It should also be noted that FLIN does not sacrifice throughput and only reorders transactions within the TSU.

WA-BC - Workload-aware budget compensation (WA-BC) [31] is a flash scheduler for SSDs that are shared between multiple virtual machines using SR-IOV, a mechanism to directly pass PCIe devices to multiple virtual machines. It gives the queue of each VM a budget and dispatches requests using round-robin, but jumps over non-empty queues that have exhausted their budget. When all non-empty queues have exhausted their budget, they are replenished.

The cost that is deducted from the budget is dependent on the workload. First, WA-BC uses linear regression to determine the different read and write costs, C_R and C_W. In a given interval, the number of read and write operations and the total time is collected. When K intervals have passed, there are thus K equations with 2 variables (Equation 5), and a linear regression along with the best fit for C_W and C_R can be com-

		QoS Goals Addressed					
Solution	**Interface**	**Tail Latencies**	**Fair**	**Isol.**	**Predict.**	**Perf.**	**Summary**
FLIN [64]	NVMe	✗	✓	✗	✗	✓	Flash I/O transaction scheduling that accounts for interference among tenants
WA-BC [31]	NVMe SR-IOV	✗	✗	✓	✗	✓	Round robin scheduling with budgets that account for GC overhead
OPS Isolation [36]	N/A	✗	✗	✓	✗	✓	Isolation of flash erase blocks to prevent interference during GC
FlashBlox [26]	Custom	✓	✗	✓	✗	✓	Hardware isolation of channels, dies, or blocks, that guarantees uniform aging
UPS [34]	N/A	✓	✗	✓	✗	✓	Hardware isolation that partitions resources dynamically based on load
ttFlash [72]	NVMe	✓	✗	✗	✓	✓	Elimination of tail latencies by minimizing resource blocking caused by GC
AutoSSD [35]	N/A	✓	✗	✗	✓	✓	Scheduler that reserves shares for internal functions based on urgency

Table 2: A summary of solutions that improve QoS at the device level

puted. The write cost is then adjusted by the WAF for each VM, i.e. the write cost may be higher for some VMs.

$$T_i = C_R \cdot N_R(i) + C_W \cdot N_W(i) \quad (5)$$

It is difficult to determine the WAF of each VM when an erase block contains pages from multiple VMs. Therefore WA-BC implements an erase-block isolation similar to other work such as OPS isolation and multi-stream SSDs. WA-BC achieves good performance isolation (QoS goal G2, isolation) when VMs with different WAF (write amplification factor) run side by side, which would otherwise cause high interference.

OPS Isolation - OPS (over-provisioning space) Isolation [36] is a hypothetical SSD that partitions the over-provisioning resources of SSDs. SSDs typically include more internal storage than can be accessed, known as over-provisioning space and is used for GC (garbage collection). When I/O streams from multiple workloads are interleaved on a single SSD, the flash blocks contain a mix of pages from all workloads. As a consequence, when one workload triggers GC of a block, it impacts the performance of all workloads.

The ratio of valid pages in a block selected by the GC is known as utilization (u) and a lower value corresponds to lower write amplification, resulting in better performance. It is shown that interleaved workloads can result in lower utilization values than if each workload is executed individually, resulting in lower throughput overall.

Using OPS Isolation, each workload is assigned a weight and both data blocks and over-provisioning blocks are

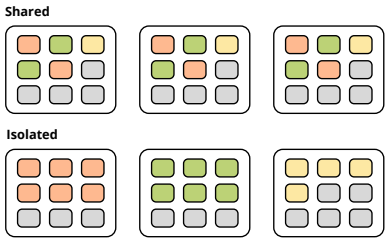

Figure 13: OPS Isolation separates pages from different workloads into different blocks.

partitioned in proportion. The I/O requests are tagged by workload so each block only contains either pages that are free or have data from a single workload (Figure 13). Each workload is also guaranteed a certain IOPS, as a proportion of the device peak IOPS based on weight. It is then possible to compute the required u value to sustain that IOPS. The algorithm dynamically distributes the OPS blocks so that the u target of each workload is reached.

FlashBlox - Similar to OPS Isolation which dedicates blocks to workloads in order to provide isolation, FlashBlox [26] provides hardware isolation by partitioning channels into so called virtual-SSDs (vSSD). Each channel of an SSD operates independently and thus there will be no interference between the vSSDs. However, channel isolation leads to a challenge with wear-leveling, as the workloads on different vSSDs may be causing unequal aging. To prevent unequal aging, FlashBlox introduces a migration mechanism that pe-

riodically swaps the least-aged channel with the most-aged channel.

FlashBlox also offers two other isolation modes with different trade-offs, (i) die-isolated vSSDs, which offers weaker isolation as multiple dies must share the same channel bus. In that case, wear-leveling must also be done to ensure equal aging of the dies. (ii) software-isolated vSSDs that is comparable to OPS Isolation. A token bucket rate limiter on the host ensures equal access to blocks within each die. FlashBlox exposes a ZNS-like interface enforcing sequential writes and each block is allocated to only one application to reduce GC interference.

FlashBlox is implemented on an open-channel SSD and benchmarks are done using LevelDB, which integrates nicely with the log-based block interface of FlashBlox. A limitation of FlashBlox is that a workload running on a channel-isolated vSSD has a strict throughput ceiling which is proportional to the number of its channels. For example, if using 4 out of 8 channels, only half the SSD throughput can be reached. A software-isolated workload may borrow unused IOPS from other workloads, but may also be causing interference and stealing the IOPS. Another limitation is the periodical channel migration for wear-leveling. An average workload may need a 15-minute migration every 3 weeks, during which throughput and latency decrease by one-third.

UPS - Utilitarian Performance Isolation (UPS) [34] is a scheme for multiple tenants to share an SSD. Rather than using a fixed partitioning such as FlashBlox, UPS partitions the NAND chips of an SSD according to each tenant's *utility* and adjusts the partitioning periodically. Pages are always written to chips within the tenant's partition, but as the partitioning changes, the tenant's data may reside within another tenant's partition.

Given a partition of NAND chips, the *utility* is a value that indicates how well a given tenant would utilize the chips. A partitioning is chosen so that the utility of all tenants is the same and is computed periodically based on previous usage. Having one tenant's reads go to another tenant's partition can affect performance isolation. To improve isolation, UPS uses the opportunity of garbage collection to relocate pages back to their tenant's partition. UPS reduces interference compared to a shared device, but also allows for more flexibility than a static partitioning, for example when one workload experiences a burst of traffic.

5.2 Eliminating Tail Latencies

Tail latencies caused by internal FTL functions such as garbage collection are a big source of QoS problems on SSDs. Solutions that attempt to hide the cost of house-keeping functions can be categorized as follows.

- **Exploiting Idle-Times** - The best time to perform the FTL functions is when throughput from the host is low.
- **Data Redundancy** - Similar to RAID, data can be striped across channels, and if one channel is busy with internal tasks, data can be reconstructed from other channels.
- **Feedback-Based Scheduling** - Internal tasks get as little share as possible of the device, only enough to keep up with its work.

We now explore two of these solutions in detail.

ttFlash - Tiny-Tail Flash (ttFlash) [72] is an SSD controller architecture that significantly reduces tail latencies. It makes use of three recent hardware advancements.

- When GC is triggered, SSDs block requests on the same channel, and simple SSDs might block the whole device (e.g. USB sticks). With the increased computing power of controllers, it is possible to implement plane-blocking garbage collection. This makes use of flash chips that support intra-plane copy commands, as explained in Subsection 3.2. Thus, the GC's intra-plane copy commands can be interleaved with commands for other planes, leaving only one plane blocked.
- Modern SSDs implement RAIN (Redundant Array of NAND) which uses parity pages to prevent data loss, as NAND chips are getting denser and thus corruption is becoming more frequent. The redundancy can also be used to quickly read a page that is blocked by GC in one plane, by reading from other planes. This requires a smart layout of the pages within planes and GC must be scheduled so that only one plane is blocked at a time for each stride (i.e. the redundancy group)
- DRAM buffers are now frequently used as write-back caches for writes, backed by capacitors in case of power loss. This removes any spikes in write latency, but the write-back operation must keep up with incoming writes. ttFlash coordinates the write-back with its GC scheduling so that data is written back to planes when they are not blocked by GC.

Using these three techniques, ttFlash can almost eliminate tail latencies regardless of workload. A downside of using parity pages is that a portion of the SSD capacity is wasted. In addition, reconstructing pages using parity data requires more reads, making operations slightly more expensive. The same applies to writes, as parity pages must also be updated.

AutoSSD - AutoSSD [35] is an architecture for SSD controllers that schedules flash operations from all FTL functions together. Flash operations can come from host I/O requests, garbage collection, wear-leveling, and read scrubbing. Read scrubbing is a preventative operation that prevents corruption

of pages that have been read too often by migrating them elsewhere.

The scheduler uses a request windowing technique that limits how many requests can be in-flight from each function based on their share. The shares are dynamically controlled using a feedback system that aims to correct an error value for each FTL function. To begin with, all shares belong to the host I/O requests, but as garbage collection, read sweeping and wear-leveling become more critical, their shares grow proportionately. It should be noted that AutoSSD is not work-conserving, a non-empty queue may be blocked by an empty queue that has a higher share.

5.3 Discussion

The solutions that we have explored aim to improve QoS with different approaches. The three most visible approaches are improvements to SSD internal scheduling, isolation schemes for SSD resources, and elimination of tail latencies. Table 2 summarizes which QoS goals are considered in each solution. We can see that device-side solutions do not sacrifice performance in general, but only one of the first three goals (fairness, isolation, predictability) is considered in each solution. Only FlashBlox imposes a new host interface, other solutions are based on NVMe or do not specify a host interface, typically because they are implemented in emulators.

6 I/O Schedulers

In this section, we look at I/O schedulers for SSDs that aim to improve QoS in terms of fairness and performance reliability, or investigate performance problems. This section corresponds to layer 2 in Figure 1. The solutions are summarized in Table 3, which will be discussed at the end of this section.

6.1 Linux Schedulers

We begin with an exploration of the I/O schedulers that are currently offered on Linux.

BFQ - The Budget Fair Queueing (BFQ) scheduler is intended to improve the fairness and latency guarantees of CFQ. Similar to CFQ, each application has its own queue, but each queue now has a budget measured in block count. Rather than selecting queues in a round-robin way, the B-WF2Q+ algorithm is used to select the next active queue. A new queue is chosen when the budget is exhausted or the queue is empty, and a new budget is computed in a way that high-throughput queues get a high budget, but latency-sensitive queues get a low budget. The B-WF2Q+ algorithm selects low-budget queues earlier, resulting in lower latency.

BFQ also comes with multiple heuristics. One heuristic is Early Queue Merge (EQM), which merges queues with requests to adjacent blocks, often seen in applications with multiple I/O worker threads. It also has special heuristics for flash and NCQ devices with internal queues, where the device may optimize the ordering at the cost of unfairness or higher latencies. Overall, BFQ has been shown to provide the same throughput as CFQ and better fairness and latency guarantees [65].

Kyber - For NVMe devices with high IOPS throughput, BFQ proved to be too complex and CPU intensive to give good performance. Kyber is a simpler scheduler designed for the multi-queue block layer. Applications typically care more about read latencies, because the application may be waiting for the data to become available, while writes are less likely to block execution. Kyber separates reads and writes to separate queues, and then moves requests to a dispatch queue that is kept short enough to make certain latency guarantees for read requests. It uses a histogram to compute the 90th percentile latency in the dispatch queue and adjusts its length accordingly. It is based on bitmap queues to give good performance [12, 16].

None - Another option for I/O scheduling on Linux is to use no scheduler (none). After the multi-queue block layer, none was the only option for around 3 years, and it is the default option on many distributions today for NVMe devices [12]. Due to the high performance of NVMe devices, all schedulers have a non-negligible impact on the throughput, and latency and fairness may be less important for some users, because the drive provides abundant performance [58].

6.2 Multi-Queue Schedulers

MQFQ - Multi-Queue Fair Queueing [24] is an I/O scheduler designed for the Linux multi-queue block layer. It introduces a fair queueing algorithm for multi-queue systems, and uses clever data structures to minimize the overhead of coordination between CPU cores. It is intended for multi-queue SSDs which have internal parallelism. An assumption is made that by limiting the number of in-flight requests to D, the device can be saturated but the SSD will not need to arbitrate between requests, which might otherwise be destructive to host-side fairness decisions. The parameter D can be discovered with a probing method.

MQFQ is based on SFQ(D), but a relaxed ordering criteria is added to decrease the required coordination between queues, as every dispatch would otherwise have to look at the virtual time of all other queues, which would limit the scalability of MQFQ. When a queue goes ahead of the window T however, it must be throttled for a short time period. In addition, all queues must obtain a dispatch slot so that the global in-flight count can be controlled. MQFQ uses three clever data structures to manage virtual time, throttling and, dispatch slots.

Solution	Required Interface	QoS Goals Addressed: Multi Queue	Multiple Dispatch	Fair	Isol.	Predict.	Perf.	Summary
MQFQ [24]	Any	✓	✓	✓	✗	✗	✓	Fair queueing scheduler that introduces efficient multi-core data structures
D2FQ [69]	NVMe WRR	✓	✓	✓	✗	✗	✓	Fair queueing scheduler that exploits device-side scheduling (NVMe WRR)
FlashFQ [60]	Any	✗	✓	✓	✗	✗	✗	Single queue fair queueing scheduler with anticipation
blk-switch [27]	Any	✓	✓	✗	✗	✓	✓	Block layer that provides low latencies while device is at peak throughput
A+CFQ H+BFQ [37]	Any	✗	✓	✓	✗	✗	✗	Improvements of CFQ and BFQ timeslice schedulers for SSDs
FIOS [54]	Any	✗	✓	✓	✗	✗	✗	Single queue timeslice scheduler that exploits parallelism with multiple active timeslices
vFair [48]	Any	✗	✓	✓	✗	✗	✗	Single queue fair queueing scheduler with a per-request cost model
MittOS [22]	OC-SSD	✓	✓	✗	✗	✓	✓	Storage stack that improves predictability with a fast rejection mechanism
K2 [51]	Any	✓	✓	✗	✓	✓	✓	Scheduler that reserves a fast path for urgent requests
TABS [38]	NVMe	✓	✓	✗	✗	✓	✓	Scheduler that prevents SSD interference due to FUA write requests

Table 3: A summary of I/O schedulers that improve QoS

1. **Mindicator** - All queues must be able to determine the global virtual time, which is the lowest time across all queues. The naive approach would be scanning an array of per-queue times, but that would cause too many cache invalidations. The Mindicator data structure uses a tree where the value of each queue is in a leaf node. When one queue's value is updated, the minimum value is traversed upwards and relatively few operations update the root value, i.e. the global minimum. The tree is aligned with respsect to thread, core, and NUMA structure for better cache utilization.

2. **Token Tree** - To keep track of the number of in-flight requests and prevent device-side arbitration, a global atomic counter does not scale. The token tree data structure keeps track of tokens that each queue must obtain before dispatching and gets released after completion. Normally, all queues have equal shares of tokens and no coordination is needed to use the local tokens. If one queue does not use all tokens, nearby cores can request unused tokens. The tree-based approach resembles the Mindicator tree and prefers tokens from the same core and NUMA domain.

3. **Timing Wheel** - When a queue is throttled, the time at which it can be unthrottled must be stored. The timing wheel is an array of bitmaps where each bit indicates a queue that can be unthrottled and a bitmap at offset i indicates the event after i periods. When an event has passed, the bitmap becomes the event after n periods, making it resemble a wheel.

The MQFQ scheduler can provide the same level of fairness as BFQ does for HDDs. However, IOPS throughput is 25% lower than when the `nosched` option is used.

D2FQ - Device-Direct Fair Queueing [69] is an I/O scheduler that is also implemented for Linux. D2FQ makes use of the NVMe Weighted Round Robin (WRR) feature to offload a part of the scheduling to the device. WRR is an optional NVMe feature that allows device queues to be given one of three priority levels (low, medium, and high) and each level a weight between 1 and 256. The device will then process I/O

requests in proportion to each queue's weight. D2FQ adds 3 queues on each core with low, medium, and high priorities and weights are dynamically adjusted (Figure 14). I/O requests get high priority by default but medium and low priority if they need to be throttled.

In the Linux block layer, D2FQ uses a virtual time-based fair queueing mechanism for scheduling. D2FQ maintains a virtual time for each flow which grows in proportion to bytes dispatched and normalized for the flow's weight. When an I/O request comes in, the scheduler compares that flow's virtual time to the global virtual time. In strict SFQ, only the flow with the lowest virtual time would be allowed to dispatch a request. In D2FQ however, all requests are dispatched immediately, but if the flow's virtual time is too far ahead, it will be dispatched to a lower priority queue. That causes the flow's virtual time to pass more slowly and thus balances all flow's virtual times.

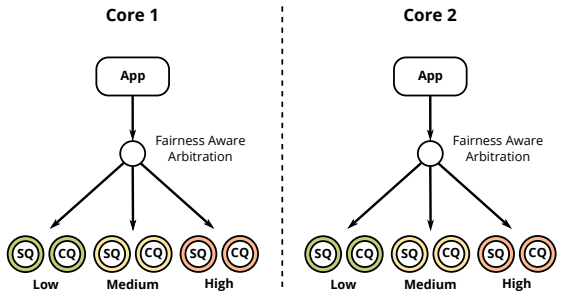

Figure 14: D2FQ with three device queue pairs per core, each with a different priority.

For D2FQ to achieve fairness, the weights of the priority levels must have enough contrast to balance the flows, i.e. the high/low ratio must be high enough. However, choosing a ratio that is too high will cause the lower-priority queues to delay requests for too long, introducing higher tail latencies. D2FQ introduces a dynamic mechanism to periodically adjust the weights if the virtual times start diverging too much, which would indicate unfairness.

Because D2FQ adds little complexity to the block layer, it is only slightly more CPU intensive than the no-op scheduler. Unlike other fair queueing schedulers, D2FQ does not provide a theoretical bound on unfairness. However empirical results show that it achieves comparable fairness to other schedulers. D2FQ shows better CPU utilization compared to MQFQ, but unlike MQFQ which works on any NVMe drive, D2FQ only works on devices with WRR support.

blk-switch - blk-switch [27] builds on ideas from network switches to modify the Linux multi-queue block layer. It separates latency-sensitive apps (L-apps) and throughput-intensive apps (T-apps) by reserving two `ionice` I/O weight values. Users must explicitly choose a profile for each process. blk-switch addresses the problem of tail latencies that occur in the Linux block layer due to CPU contention. The high IOPS from T-apps can induce high CPU usage in the block layer which can slow down the processing of requests from L-apps.

For each CPU core, blk-switch creates two egress queues, for L-apps and T-apps. Requests from L-apps are always prioritized over T-apps. To prevent complete starvation of T-apps, if the load of a CPU is above a threshold, T-app requests are routed to an egress queue of a less loaded core. This differs from the base Linux block layer, where I/O requests never move between cores. The processing of L-apps and T-apps is done by separate kernel threads and the L-app thread is given a higher priority. Thus if a core has a high CPU load (e.g. due to block layer overhead) it will still process L-app requests with low latency.

blk-switch has been shown to successfully maintain low latencies in high-throughput scenarios (QoS goal G4, performance). However, it achieves slightly less total throughput than the base Linux block layer.

MittOS - MittOS [22] is a mechanism to quickly reject I/O requests if their SLA can't be satisfied. An application can specify a threshold for tail latencies that it can tolerate, and use a modified `read` syscall that returns an `EBUSY` error if there is too much contention to complete the request in time. For database applications with multiple replicas, the request can be moved to a different replica if the first one returns EBUSY. Without knowledge of contention, applications can also use a wait-and-speculate method, where if a request has not completed before a threshold, it assumes contention and tries a different replica. Figure 15 shows an application that immediately sees contention compared to an application that waits and speculates after a threshold, which results in more wasted time.

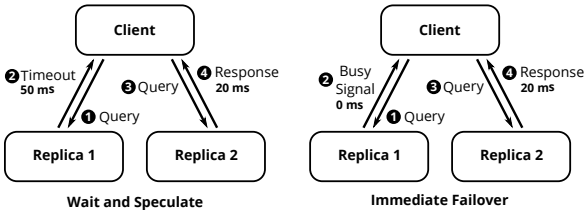

Figure 15: A comparison of speculation and a MittOS busy signal.

To implement MittOS, the latency of an incoming request must be accurately speculated. For mechanical drives with no device side queueing, the estimation can be done in the OS scheduler. With a FIFO scheduler, the latency of the request is simply the sum of all requests ahead in the queue. With schedulers that reorder requests, however, a request may initially appear to get low latency, but a later request with higher priority may push it back. In that case, MittOS may cancel older requests while still respecting the SLA. The latency estimation becomes more complex on NVMe drives,

as knowledge of the internal state is required. MittOS uses Open Channel SSD to observe per-channel contention. Large requests may access many channels, in which case the slowest channel determines the overall latency.

In multi-tenant environments with shared storage, noisy neighbors can lead to high tail latencies for database queries. However, it is unlikely that multiple database replicas experience contention at the same time. With the MittOS fast rejection mechanism, tail latencies are reduced almost to the level of a single-tenant environment.

K2 - K2 [51] is a multi-queue scheduler that provides real-time guarantees for requests, but does not guarantee fairness. It uses 8 staging queues corresponding to the eight `ionice` priority levels on Linux. It always consumes from the highest priority queue that is not empty in order to provide lower latency to high-priority requests. K2 also limits the number of in-flight requests, similar to MQFQ, to prevent device-level scheduling from doing arbitration that is destructive to host-side decisions. When a latency-sensitive process runs alongside high-throughput processes, K2 is able to provide lower latencies than other Linux I/O schedulers, however, it sacrifices considerable throughput.

6.3 Single-Queue Schedulers

FlashFQ - FlashFQ [60] is an I/O scheduler for Linux's single-queue block layer. Firstly, FlashFQ argues that timeslice scheduling as done by CFQ, BFQ, Argos, and FIOS is not suitable for Flash as it impacts responsiveness. Instead, a fine-grained per request queueing algorithm should be used. That is because when a flow's timeslice is out, it must wait in round-robin for its next slice, leading to high latencies.

FlashFQ uses SFQ(D), an extension of SFQ that can dispatch D requests in parallel. Dispatching requests in parallel is essential to utilizing SSDs full performance, but it also introduces interference between requests. FlashFQ recognizes that interference on the flash device may lead to unbalanced performance between two parallel requests. To balance flows, FlashFQ introduces a throttled dispatch mechanism that throttles flows when their virtual time is too far ahead.

As described in Section 3, anticipation improves throughput on mechanical drives by idling for short periods in the hopes of receiving more sequential requests, rather than immediately serving requests that require more seek time. Although SSDs do not benefit from maximizing locality, FlashFQ finds that anticipation can be used to improve fairness. A characteristic of fair queueing algorithms is that when a flow becomes inactive, it can not accumulate shares, instead it starts with a new virtual time when it becomes active again. By holding the virtual time of a flow for a short window after becoming inactive, the flow can use accumulated resources in the case of deceptive idleness.

A+CFQ and H+BFQ - Kim et al. [37] look at I/O proportionality of the CFQ and BFQ schedulers. To satisfy SLA requirements, processes and VMs can be given different I/O weight using Linux cgroup. However, CFQ and BFQ do not reach the expected proportionality on Flash devices. This is caused by device queue idling on fast SSDs, as CFQ will constantly be switching between request queues, making it difficult to reach proportionality goals.

To improve proportionality, A+CFQ implements anticipation in CFQ which prevents excessive switching between queues. In the case of BFQ, proportionality is affected by excessive budget allocations. H+BFQ (History added BFQ) gives less budget to aggressive processes. Although satisfying proportionality is an important goal for modern I/O schedulers, BFQ and CFQ have been shown to perform poorly on SSDs.

FIOS - FIOS [54] is a Flash I/O scheduler that uses a timeslice method for fairness. It addresses limitations in the CFQ scheduler, which does not fully utilize SSD parallelism, and does not recognize the different read and write costs on Flash storage. Two features are added to the timeslice mechanism. First, multiple timeslices may be active at the same time, and second, timeslices may be fragmented within the round-robin cycle (also known as epoch). For every dispatched request, the cost is subtracted from the timeslice. In CFQ, the cost is the request size in bytes. However, FIOS builds two linear models, for reads and writes, to estimate their cost based on their size, improving fairness.

vFair - vFair [48] is a single-queue I/O scheduler that provides fairness by computing per-request IO costs. For each flow, vFair determines the saturation point of the access pattern, i.e. the peak throughput that could be reached if running alone. Initially, four models must be calibrated that give the peak IOPS that can be reached for a given request size and access type (sequential, random, read, or writes). vFair collects the ratio of the four different access types for each flow, and periodically computes the peak IOPS by combining the four models. vFair then reaches fairness by slowing down all flows equally with respect to each flow's saturation point.

vFair also shows that synchronous flows are likely to be slowed down by asynchronous flows, because synchronous flows can only submit more requests after digesting the responses of previous requests. This makes it harder to quantify unfairness, as synchronous flows do not have a visible backlog, thus showing deceptive fairness.

TABS - TABS [38] is an I/O scheduler that throttles writes to ensure fairness to reads. In particular, some database applications make heavy use of the NVMe FUA (forced unit access) flag that makes writes go straight to flash rather than the device's write-back cache. In that case, strong interference occurs between reads and FUA writes. TABS limits the

number of outstanding FUA writes in order to speed up read requests. It uses a dynamic feedback mechanism to adjust the queue depth limit that is required to achieve fairness. TABS only considers fairness between operation types rather than between processes.

6.4 Other Work

Yang et al. [74] argue that fairness can not be achieved by an I/O scheduler alone. First, when read and write requests are cached in the page cache, processes may unfairly evict pages from other processes. Second, the I/O scheduler is not allowed to reorder certain write requests for consistency requirements. The Argon I/O scheduler [66] explores similar cache unfairness in device-side read-ahead and write-back buffers. To prevent unfairness, Argon implements a cache partitioning scheme.

Most fairness schedulers need to determine the cost of each request to either advance virtual time or subtract from budgets. The simplest schedulers use request latency or size in bytes as cost. Some schedulers (e.g. FIOS, vFair) build models that require initial calibration. Liu et al. [47] propose D-IOCost, a cost model that dynamically adjusts flow weights.

6.5 Discussion

We have seen multiple approaches to improving QoS with I/O schedulers, such as fair queueing algorithms, utilizing special hardware features, special treatment of latency-sensitive requests, and throttling heuristics. Table 3 shows a summary of the solutions. Several solutions offer good performance (QoS goal G4), however, only D2FQ can fully saturate a device. For the first three goals (fairness, isolation, predictability) all solutions only consider one out of three, apart from K2. Most solutions can run on a standard host interface, however, D2FQ and MittOS require special hardware features.

7 Virtualization

For cloud service providers, it is often practical to offer virtual machines and container-based services in order to better utilize resources and provide more flexibility. For example, resources can be oversubscribed and tenants can easily scale up and down. In these multi-tenant environments, providers are often bound by service level agreements (SLA) and must guarantee certain throughput or latency. This chapter explores the storage stack design for hypervisors and containers, corresponding to layer 3 in Figure 1, and how QoS can be integrated into such designs. The solutions are summarized in Table 4, which will be discussed at the end of this section.

7.1 Storage Virtualization

There are many options for providing storage to a virtual machine. Perhaps the simplest method is hardware passthrough, where an entire device is passed through using an IOMMU mapping. The NVMe standard also supports SR-IOV, where the device is partitioned and appears as multiple separate devices on the PCI bus. Hardware passthrough typically offers near-native performance but limits flexibility and the number of virtual machines.

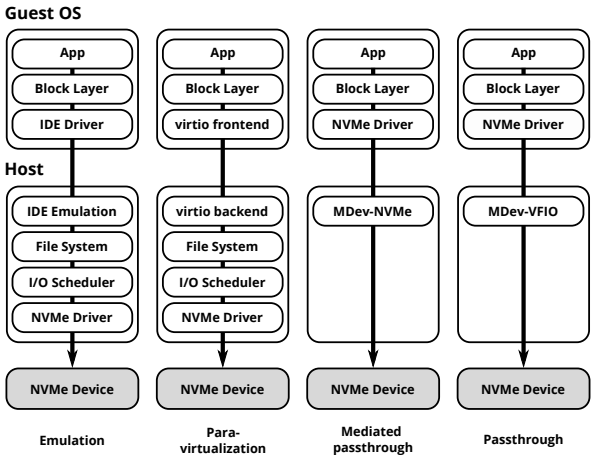

Figure 16: I/O paths of four storage virtualization options.

On the other end of the spectrum is full software emulation, where the hypervisor uses the *trap and emulate* method to expose a real NVMe or SCSI device to the guest OS, but every operation gets trapped by the hypervisor, which then emulates the device. However, emulating a real device can be CPU-intensive. Para-virtualization is a technique where the guest OS is presented with a simplified I/O interface, which is more friendly to the hypervisor compared to real hardware interfaces. KVM provides para-virtualized storage with virtio-blk and virtio-scsi which typically perform better than emulation. Figure 16 shows the different I/O paths of these solutions.

Recent research on storage virtualization has focused on bridging the performance gap between bare metal and virtual machines with different approaches. Two popular approaches are SPDK storage drivers that run entirely in userspace, and mediated passthrough where most device functions are passed directly to a VM with minimal emulation. Dowty and Sugerman [18] introduce the following taxonomy for GPU virtualization, which can also explain the different trade-offs in storage virtualization.

- **Performance** - How does the virtualized storage perform compared to the underlying device and how much additional CPU usage is introduced.
- **Fidelity** - How feature-rich is the virtualized storage?

Solution	CPU Overhead	Required Hardware	QoS Goals Addressed: Fair	Isol.	Predict.	Perf.	Summary
MDev-NVMe [56]	Polling thread	NVMe	✗	✗	✗	✓	NVMe storage virtualization with polling based mediated passthrough
SPDK vhost-NVMe [75]	Polling thread	NVMe	✗	✗	✗	✓	Userspace NVMe virtualization backend that uses SPDK
FVM [42]	Minimal	FPGA NVMe	✗	✓	✗	✓	Hardware assisted NVMe virtualization that uses an FPGA card
LeapIO [43]	Minimal	SmartNIC NVMe	✗	✓	✗	✓	Hardware assisted NVMe virtualization with an ARM co-processor
LPNS [55]	Polling thread	NVMe	✓	✓	✗	✓	Extension of MDev-NVMe that adds QoS and latency predictability
mClock [21]	Unknown	None	✓	✓	✗	✗	VMware ESX scheduler with QoS based on limits and reservations
blk-iocost [25]	Minimal	None	✓	✗	✗	✓	Cgroup block layer policy that throttles I/O based on a cost model

Table 4: A summary of solutions that improve QoS in virtualized- and container environments

For example trim support and NVMe admin commands such as WRR configuration.

- **Multiplexing** - Can the underlying device be shared with multiple tenants? And if so, are there any constraints on the scalability.
- **Interposition** - Can additional features be introduced between the VM and storage device? This could be features such as checkpointing, compression, backups, and live-migration.

We will now take a closer look at two state of the art storage virtualization techniques that offer near-native performance. In a later section, we will see an extension of this work that offers QoS.

MDev-NVMe - MDev-NVMe [56] is a mediated passthrough mechanism for NVMe. It uses the VFIO mdev (mediated devices) Linux kernel module. MDev-NVMe exposes an emulated NVMe device to the guest OS with a number of admin queues and I/O queues. However, each virtual I/O queue is a shadow of a physical queue on the device. When I/O is submitted, MDev-NVMe translates both the DMA address of the data and the LBA of the device block. All admin commands are fully emulated however in order to present a virtual device to the guest, which may have different properties than the physical device.

When I/O is submitted over NVMe, the host typically writes to the device doorbell register with MMIO. In a virtual machine, this can be expensive as each MMIO write is trapped and requires emulation. NVMe offers an admin command to set up a shadowed doorbell register, which MDev-NVMe can poll to prevent frequent VM exits. MDev-NVMe thus enables zero-copy NVMe virtualization with minimal emulation overhead and provides near-native performance.

SPDK vhost-NVMe - To improve VirtIO performance, the vhost protocol can be used to move emulation from QEMU into a kernel module or a user process. SPDK vhost-NVMe [75] is a userspace storage target that communicates with the guest OS driver through shared memory and can poll the guest NVMe queues to minimize VM exits. Similar to MDev-NVMe, this also allows for zero-copy operation with few VM exits, thus providing near-native performance. SPDK is a complete userspace NVMe driver that bypasses the overhead of the block layer.

Other vhost targets also exist, such as a kernel-based virtio-scsi backend and userspace virtio-scsi and virtio-blk SPDK drivers, these protocols don't match NVMe data structures as well and require more conversion, impacting performance. Running the target in userspace also improves performance as there are fewer kernel context switches.

7.2 Hardware Offloading

A limitation of MDev-NVMe and SPDK vhost-NVMe is that they rely on polling to provide good performance, which requires at least one dedicated CPU core. Kwon et al. [42] argue

that if the performance of NVMe devices grows faster than CPUs, the CPU overhead of those solutions will grow. They find that SPDK vhost-NVMe requires 61% more CPU time compared to native storage. As a response, they propose FVM, an FPGA-assisted storage virtualization technique. The FPGA card presents SR-IOV virtual devices to VMs and interacts with SSDs directly through PCIe peer-to-peer communication.

Li et al. [43] also find that host-based storage virtualization comes with a high CPU tax, as it takes up 10-20% of hypervisor CPU time. They propose LeapIO, an ARM-based SmartNIC for offloading storage virtualization. The two solutions offer limited QoS however. FVM implements round-robin dispatching between virtual I/O queues, but does offer bandwidth throttling for individual VMs. LeapIO dispatches from virtual queues in FIFO order.

7.3 Fair Mediated Passthrough

As we have seen, the latest storage virtualization techniques have focused on high throughput, but have limited QoS support. Most techniques also bypass the OS block layer, and thus QoS control in the I/O scheduler does not help. LPNS [55] is a mediated passthrough virtualization system inspired by MDev-NVMe, that also provides predictable latencies. LPNS can be used in hybrid deployments where some virtual queues are used by the host NVMe driver, while others are passed through to VMs.

Using LPNS, a virtual storage device can be tagged to indicate that it is latency-sensitive. The NVMe hardware queues are split into two groups, queues that are 1:1 mapped to a virtual device, and queues that are 1:N mapped to multiple devices (Figure 17). Latency-sensitive VMs are given 1:1 queues to prevent head-of-line blocking. LPNS uses a method called deterministic network calculus, borrowed from networking theory, to compute upper bounds on latency for a given throughput and throttles virtual queues when the throughput would cause latency goals to be exceeded.

LPNS can give an upper bound on the latency of VMs, but compared to MDev-NVMe the throughput loss is around 20%. LPNS uses a polling thread to mediate the passthrough. A single thread is enough to utilize a single SSD, however more threads may be needed to achieve predictable latency goals. This technique thus suffers from a CPU tax.

mClock [21] is an I/O scheduler implemented on VMware ESX. It offers setting I/O weights for each VM, but also a minimum reservation and a maximum limit in terms of throughput. It combines a fair-queueing scheduling method with a constraint-based method, satisfying the fairness criteria and also minimum and maximum constraints.

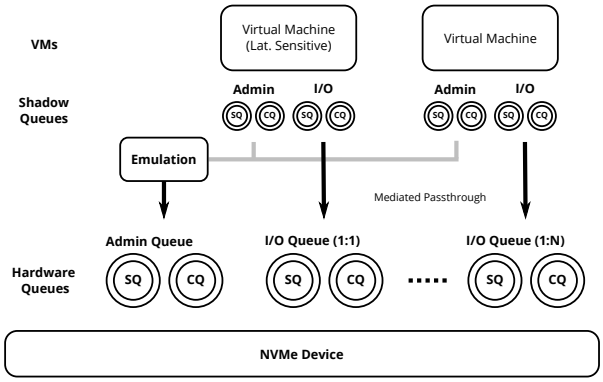

Figure 17: The structure of LPNS.

7.4 Container Storage

Containers offer lightweight virtualization by isolating processes on the same operating system. They allow for higher levels of consolidation compared to VMs because resources are not statically reserved. On Linux, the cgroup feature can be used to isolate and limit resources for processes. For I/O resource control, the simplest mechanism is `blk-throttle`, which allows limiting read/write IOPS or bytes per second. Since the total throughput of storage devices varies with access patterns, setting fixed limits is not effective for QoS control. [4, 25]

IOCost (`blk-iocost`) [25] improves QoS with an I/O cost model. The model returns the cost of a request in virtual time. When a request is submitted, the cgroup local virtual time progresses by the cost, which is adjusted by the cgroup I/O weight. Requests are throttled if the cgroup is ahead of global virtual time. By default, IOCost uses four linear models to give different costs to read/write and random/sequential requests. An arbitrary model can be specified in the form of an eBPF program. Both `blk-throttle` and IOCost are built into Linux and have been shown to have near-zero overhead on high-performance SSDs.

7.5 Other Work

Ahn et al. [4] observed that blk-throttle could not achieve proportionality, i.e. balance the throughput of different cgroups according to their weights. They propose WDT, which achieves proportional sharing through dynamic throttling. Spool [71] is a userspace SPDK driver for QEMU that improves the reliability of NVMe storage for hypervisors. They find that device errors are regular occurrences in large-scale clouds, most of which can be solved by restarting the storage system. Spool minimizes the restart duration so that tenants experience a milder latency spike. vMigrater [29] shows that when virtual CPUs are time-shared (i.e. not dedicated cores), I/O may be poorly utilized when a vCPU becomes inactive. They propose migrating I/O operations between vCPUs if the

VM has another active vCPU.

7.6 Discussion

We have seen several approaches to improving QoS in virtualized environments, including mediated passthrough, userspace SPDK drivers, and hardware-assisted storage virtualization. Table 4 summarizes the solutions from this section and the QoS goals that they address. Most solutions focus on improving performance, as older solutions typically perform poorly compared to bare-metal. Several solutions also provide isolation or fairness, however, no solution looks into predictability. A notable trade-off is that software solutions all use polling and thus sacrifice one or more CPU cores, while specialized hardware may introduce additional costs.

8 SSD Emulation

As we have seen, the storage stack of flash devices has a large design space. The organization of flash chips, design of the flash translation layer, different host interfaces, and application side offer many design options. To prototype new designs, emulation and simulation platforms can model device internals. Many of the solutions we explored in previous chapters rely on emulation platforms to prove their viability.

SSD prototyping platforms can be divided into three classes that have different features and challenges.

- **Emulators** - Emulators run in real-time and typically appear as virtualized devices inside a VM or on a bare-metal host. Emulators therefore allow running unmodified applications on top. The real-time criteria poses a big challenge for emulator design.
- **Simulators** - Simulators are not bound by real-time criteria and typically play trace files from real applications. This makes simulators simple and cheap to run, but trace files are more limited than running live applications.
- **Real hardware** - Hardware prototyping platforms are real devices that allow prototyping different firmware or possibly device organization. They offer running live applications in real-time, but they can be expensive and some design aspects such as chip organization may be fixed.

In the following subsections we will explore the details of each class of solutions.

8.1 Emulators

Real-time emulation of a flash device is cost-efficient and allows experiments with modifications on all levels of the storage stack. Figure 18 shows the typical structure of an emulator. FEMU [44] is a state-of-the-art SSD emulator that extends QEMU to expose an NVMe device within a VM. FEMU communicates with a VM through a shared memory region, similar to SPDK userspace storage drivers, enabling fast and interrupt-free execution. Because FEMU is backed by DRAM and is polling-based, the guest OS sees latencies on par with modern SSDs, both average and tail-latencies, which is a requirement for the emulation of a fast SSD. With a low base latency, FEMU can introduce a delay model to accurately reflect a real device.

The delay model adds the same delay to a request that it would experience on a real device. The delay model can account for queueing delays due to internal contention, transfer times to flash chip registers, and the read, write, and erase times of the chip. It can also account for different flash translation and garbage collection schemes. FEMU can be used to explore the QoS improvements of new designs, e.g. with new GC schemes, performance isolation designs, and extending NVMe commands for split-level designs. Both ttFlash and MittOS are evaluated using FEMU.

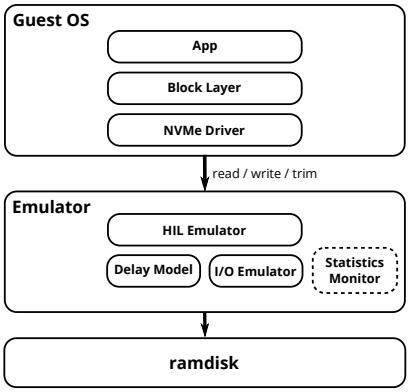

Figure 18: Typical structure of a VM-based storage emulator.

ConfZNS [62] is an extension of FEMU to emulate different designs of ZNS devices. In particular, the zone size and how zones are mapped to the internal channels, chips, and planes can affect the performance and performance-isolation of each zone. An evaluation using ConfZNS finds that ZNS offers stronger performance isolation when internal parallel units are mapped to single zones.

VSSIM [76] is an older SSD emulator that operates on QEMU, but is based on a virtual IDE interface which limits scalability and multi-queue experiments. VSSIM can be either backed by DRAM or a real SSD for emulating a large device, as long as the real device is faster than the emulated device. Gugnani et al. [20] extend the built-in QEMU NVMe emulator to support QoS research by implementing the weighted round robin (WRR) and deficit round robin (DRR) arbitration schemes. Experiments with their emulator show that DRR can be used to provide bandwidth guarantees at the hardware level.

Most emulators are built for virtual machines. Kim et

al. [39] show that this can limit advanced features such as userspace NVMe drivers and peer-to-peer PCIe functionality. Their solution, NVMeVirt emulates a real PCIe device by setting up a memory region that appears as a PCI bus, and then implementing NVMe emulation on top. NVMeVirt thus appears as a real device to both the host and other PCIe devices. It supports experiments with bare-metal applications and produces lower latencies than FEMU, which the authors show is only slightly faster than modern Optane devices.

8.2 Simulators

Simulators work with I/O trace files and can easily simulate all SSD details in the absence of time constraints. One challenge of simulators is accounting for the relations between requests in a trace, e.g. whether there is a causal dependency between requests and the host processing time between requests. In addition, traces can not be scaled for experiments with different device sizes. Not running in real-time also rules out experiments with host and application design. However, the simplicity of simulations has lead to several popular solutions.

MQSim [63] is a state-of-the-art simulator that simulates both NVMe and SATA SSDs. It improves upon older simulators by supporting multi-queue internals and accounts for more sources of latencies. Most older simulators consider flash chip operations, chip transfers, and channel queue delays to be the most significant sources. MQSim argues that as flash chips become faster, other sources become non-negligible such as PCIe transfers, translation table lookups, and DRAM cache time. MQSim also introduces a fast method for preconditioning the simulated device, as experiments should be done after the SSD has reached a steady-state, i.e. after all pages have been written to at least once.

MQSim is shown to be useful for QoS research, e.g. to analyze unfairness caused by interference between flows on current SSD designs. An experiment with MQSim shows that (1) write performance of a modest flow can be slowed down by an intense flow by causing evictions in the write cache, (2) when translation tables are cached, one flow that causes frequent evictions can slow down a translation cache-friendly flow, and (3) contention from an intense flow in per-channel queues can slow down modest flows.

Other simulators include WiscSim [23] which simulates an NCQ SATA drive, and FlashSim [40], one of the first flash simulators. Both are limited by not supporting multi-queue designs.

8.3 Hardware Prototyping Platforms

Hardware prototyping platforms can present a storage device without any host software. OpenExpress [32] implements a complete NVMe controller on a Xilinx FPGA board backed with DRAM. It can produce a bandwidth and latency comparable to an Optane SSD. However, it does not come with a ready-to-use delay model like many simulators and emulators. The low clock frequency of the FPGA also limits performance, which may not be enough for prototyping next-generation devices. Cosmos+ OpenSSD [41] is a similar FPGA platform but it is backed with NAND chips and thus the flash characteristics are fixed.

9 Future Work

In this survey, we have seen a variety of solutions that aim to improve QoS in different ways. However, despite many proposed solutions, few have found their way into real-world systems. The two state-of-the-art fair I/O schedulers, MQFQ [24] and D2FQ [69], have limitations that may hinder adoption. Namely, using MQFQ results in a non-negligible throughput reduction, and D2FQ requires a hardware feature (WRR) that few devices have. Future work might further explore the bottlenecks of fair I/O scheduling. The most advanced solution with real-world adoption might be IOCost [25], used to provide I/O fairness to Facebook's entire container fleet. There is currently limited work comparing cgroup policies and I/O schedulers.

We have seen multiple QoS strategies in this work, such as fair queueing algorithms, strict resource isolation, and special treatment of latency-sensitive requests. A better understanding may be needed of how these strategies affect end applications. We have also seen how I/O fairness can be provided by hypervisors with a mediated passthrough solution [55]. These methods may need to be improved to support advanced interposition features such as live migration, in order to get wider adoption. Finally, there is limited work that explores cross-stack solutions, e.g. adding features to host interfaces or exposing QoS information to applications.

A trend that is visible at all layers of the survey is that keeping up with ever-improving SSD performance is a challenge. Thus, solutions that provided good performance a few years ago may not be sufficient today. Finally, a growing concern in cloud computing is energy efficiency. However, most solutions do not explore their impact on energy usage. For example, some solutions depend on polling to achieve good performance, at the cost of dedicating one or more CPU cores for polling, thus increasing energy costs.

10 Conclusion

This survey has explored various aspects of QoS challenges on NVMe drives. We conclude this survey by answering the questions that we presented in Section 2, which also answer our main survey question: *What are the challenges in providing fairness and reliable performance to multiple applications on modern SSDs?*

RQ1: How can I/O schedulers preserve peak throughput on

SSDs while providing other QoS guarantees?

We have seen that providing fairness requires a global view of all queues. Multi-queue I/O schedulers are essentially distributed systems, and a major challenge is efficient coordination between CPU cores [24]. Coordination can either be done using optimized data structures or with the assistance of device-side features. All the solutions explored in Section 6 bring some throughput reduction, apart from D2FQ [69], which uses device-side arbitration to simplify scheduling. Thus, preserving throughput without specialized device features is still an open problem.

RQ2: What are the causes of performance degradation on SSDs and how can they be prevented?

As we explored in Section 4 and Section 5, SSDs can suffer from performance degradation, which is influenced by their internal functions and parallelism. In particular, latency spikes and reduced throughput can occur due to a bad access pattern, the SSD's internal tasks, or by inference from another flow. Requests must be balanced across channels and planes to fully utilize the device, and scheduling must account for fairness and conflicts due to Flash characteristics. Several solutions mitigate these problems effectively, for example ttFlash [72] and FLIN [64].

RQ3: What factors can limit QoS in SSD-based virtualized and container environments?

We discussed the many storage virtualization solutions that are available in Section 7. For NVMe drives, standard solutions result in poor I/O performance within virtual machines, and many solutions have thus been proposed for improving throughput [56, 75]. However, these solutions bypass the host block layer, making it harder to provide QoS. A single solution exists that offers high throughput and QoS with mediated passthrough [55]. Containers go through the block layer of the OS like any other process, and can be adjusted with cgroup policies, as we have seen with IOCost [25].

RQ4: How can the design space of SSDs and block layers be explored with the goal of improving QoS?

The emulators and simulators that we discussed in Section 8 can facilitate a detailed exploration of the design space of SSDs. Many of the solutions explored in this survey use emulators to confirm their viability. The most apparent limitation for further QoS research is the difficulty of keeping up with the performance of the latest SSDs. The solutions that we explored offer a best-case latency that is not far ahead of modern devices.

References

[1] Global cloud computing services market to reach $2.1 trillion by 2030. https://finance.yahoo.com/news/global-cloud-computing-services-market-153000451.html. Accessed: 2023-09-25.

[2] Nvm express, revision 1.3a. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf. Accessed: 2023-09-25.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for {SSD} performance. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[4] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving {I/O} resource sharing of linux cgroup for {NVMe}{SSDs} on multi-core systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[5] Jens Axboe. CFQ IO scheduler. http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.pdf. Accessed: 2023-08-22.

[6] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. Accessed: 2023-08-22.

[7] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[8] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[10] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. {LightNVM}: The linux {Open-Channel}{SSD} subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, 2017.

[11] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277. IEEE, 2011.

[12] Jonathan Corbet. Two new block i/o schedulers for 4.12. https://lwn.net/Articles/720675/. 2017-04-24.

[13] Michael Cornwell. Anatomy of a solid-state drive. *Communications of the ACM*, 55(12):59–63, 2012.

[14] Intel Corporation. Optane SSD P5800X series product brief, 2023.

[15] Yuhui Deng. What is the future of disk drives, death or rebirth? *ACM Computing Surveys (CSUR)*, 43(3):1–27, 2011.

[16] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[17] Western Digital. Ultrastar dc hc560 data sheet, 2022.

[18] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[19] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 157–168, 1996.

[20] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. Analyzing, modeling, and provisioning qos for nvme ssds. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 247–256. IEEE, 2018.

[21] Ajay Gulati, Arif Merchant, and Peter J Varman. {mClock}: Handling throughput variability for hypervisor {IO} scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[22] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O Suminto, Cesar A Stuardo, Andrew A Chien, and Haryadi S Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 168–183, 2017.

[23] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proceedings of the twelfth European conference on computer systems*, pages 127–144, 2017.

[24] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. {Multi-Queue} fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019.

[25] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. Iocost: block io control for containers in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 595–608, 2022.

[26] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. {FlashBlox}: Achieving both performance isolation and uniform lifetime for virtualized {SSDs}. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[27] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 113–128, 2021.

[28] Nikolas Ioannou, Kornilios Kourtis, and Ioannis Koltsidas. Elevating commodity storage with the salsa host translation layer. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 277–292. IEEE, 2018.

[29] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis CM Lau, Yuexuan Wang, et al. Effectively mitigating {I/O} inactivity in {vCPU} scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 267–280, 2018.

[30] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.

[31] Byunghei Jun and Dongkun Shin. Workload-aware budget compensation scheduling for nvme solid state drives. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2015.

[32] Myoungsoo Jung. {OpenExpress}: Fully hardware automated open research framework for future fast {NVMe} devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 649–656, 2020.

[33] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed {Solid-State} drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[34] Bryan S Kim. Utilitarian performance isolation in shared {SSDs}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[35] Bryan S Kim, Hyun Suk Yang, and Sang Lyul Min. {AutoSSD}: an autonomic {SSD} architecture. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 677–690, 2018.

[36] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards {SLO} complying {SSDs} through {OPS} isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, 2015.

[37] Jaeho Kim, Eunjae Lee, and Sam H Noh. I/o schedulers for proportionality and stability on flash-based ssds in multi-tenant environments. *IEEE Access*, 8:4451–4465, 2019.

[38] Jieun Kim, Dohyun Kim, and Youjip Won. Fair i/o scheduler for alleviating read/write interference by forced unit access in flash memory. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 86–92, 2022.

[39] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. {NVMeVirt}: A versatile software-defined virtual {NVMe} device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 379–394, 2023.

[40] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *2009 First International Conference on Advances in System Simulation*, pages 125–131. IEEE, 2009.

[41] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[42] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. {FVM}:{FPGA-assisted} virtual device emulation for fast, scalable, and flexible storage virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971, 2020.

[43] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[44] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, 2018.

[45] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S Gunawi. Fantastic ssd internals and how to learn and use them. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 72–84, 2022.

[46] Hao-Ran Liu. Linux I/O schedulers. https://hzliu123.github.io/linux-kernel/Linux%20IO%20Schedulers.pdf. Accessed: 2023-08-22.

[47] Yachun Liu, Dan Feng, Jianxi Chen, and Chao Guo. D-iocost: Dynamic cost-aware fair queueing for better i/o proportionality and performance. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 373–391. Springer, 2022.

[48] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vfair: latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138, 2015.

[49] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 144–159, 2016.

[50] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of {SSDs} in enterprise storage systems: A {Large-Scale} field study. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 165–180, 2022.

[51] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. K2: Work-constraining scheduling of nvme-attached storage. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 56–68. IEEE, 2019.

[52] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[53] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993.

[54] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, volume 12, pages 13–13, 2012.

[55] Bo Peng, Cheng Guo, Jianguo Yao, and Haibing Guan. {LPNS}: Scalable and {Latency-Predictable} local storage virtualization for unpredictable {NVMe}{SSDs} in clouds. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 785–800, 2023.

[56] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. {MDev-NVMe}: A {NVMe} storage virtualization solution with mediated {Pass-Through}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 665–676, 2018.

[57] Stephen Pratt and Dominique A Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Linux symposium*, volume 2, pages 425–448, 2004.

[58] Zebin Ren and Animesh Trivedi. Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 35–45, 2023.

[59] Seetharami Seelam, Rodrigo Romero, Patricia Teller, and Bill Buros. Enhancements to linux i/o scheduling. In *Proc. of the Linux symposium*, volume 2, pages 175–192, 2005.

[60] Kai Shen and Stan Park. {FlashFQ}: A fair queueing {I/O} scheduler for {Flash-Based}{SSDs}. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.

[61] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[62] Inho Song, Myounghoon Oh, Bryan Suk Joon Kim, Seehwan Yoo, Jaedong Lee, and Jongmoo Choi. Confzns: A novel emulator for exploring design space of zns ssds. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, pages 71–82, 2023.

[63] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. {MQSim}: A framework for enabling realistic studies of modern {Multi-Queue}{SSD} devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[64] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.

[65] Paolo Valente and Mauro Andreolini. Improving application responsiveness with the bfq disk i/o scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[66] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, volume 7, pages 5–5, 2007.

[67] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. Do we still need io schedulers for low-latency disks? In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 44–50, 2023.

[68] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[69] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. {D2FQ}:{Device-Direct} fair queueing for {NVMe}{SSDs}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 403–415, 2021.

[70] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[71] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, et al. Spool: Reliable virtualized {NVMe} storage pool in public cloud infrastructure. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 97–110, 2020.

[72] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[73] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

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

[75] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[76] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. Vssim: Virtual machine based ssd simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2013.

[77] Hui Zhang and Jon CR Bennett. Wf2q: worst-case fair weighted fair queueing. In *IEEE INFOCOM*, volume 96, pages 120–128, 1996.