






Does Linux Provide Performance Isolation for NVMe SSDs? Configuring cgroups for I/O Control in the NVMe Era

Krijn Doekemeijer 
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Zebin Ren 
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Tiziano De Matteis 
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Balakrishnan Chandrasekaran 
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Animesh Trivedi 
IBM Research Europe
Zurich, Switzerland

Abstract—Modern storage workloads commonly run in containers within data centers, such as machine learning, databases, caches, HPC, and serverless workloads. To facilitate the storage performance requirements (e.g., bandwidth, latency) of these workloads, data centers have adopted fast NVMe SSDs as a storage medium. At the same time, data centers virtualize and share these storage resources with multiple tenants to improve resource utilization and reduce costs. Such sharing leads to an inherent trade-off between tenant performance isolation and SSD utilization. Although various research studies demonstrate how to achieve various performance isolation properties, such as fairness, there is neither a unified definition for performance isolation nor a benchmark. Furthermore, the isolation capabilities of state-of-the-practice I/O control mechanisms in the Linux kernel are not well understood. In this paper, we address these three challenges. First, we survey the definition of performance isolation and uncover four common performance isolation desiderata. Second, we introduce `isol-bench`, a benchmark for evaluating these desiderata for I/O control mechanisms. Third, we use `isol-bench` to evaluate I/O isolation for Linux’s state-of-the-practice I/O control mechanism, cgroups. From our evaluation, we are able to conclude that out of cgroups’s knobs `io.cost` achieves the most isolation desiderata, but has a latency overhead past CPU saturation. We open-source the source code of `isol-bench` at <https://github.com/atlarge-research/isol-bench>.

Index Terms—cgroups, Measurements, NVMe, performance isolation

I. INTRODUCTION

A large number of storage workloads run in containers within data centers, including machine learning [64], databases [94], caches [77], HPC [43], and serverless applications [50], [93]. To facilitate the performance requirements of such containerized workloads, data centers have adopted fast

This work is partially supported by Netherlands-funded projects NWO OffSense (OCENW.KLEIN.209), NWO MLS (OCENW.KLEIN.561), and GFP 6G FNS. The work is also supported by EU-funded projects MSCA CloudStars (g.a. 101086248) and Horizon Graph Massivizer (g.a. 101093202). Krijn Doekemeijer is funded by the VU PhD innovation program.

TABLE I: Performance isolation desiderata for cgroups; the “+” indicates that a knob had to be evaluated together with an I/O scheduler (i.e., MQ-DL, BFQ) or other knob.

cgroups I/O control knob	Low Overhead	Proportional Fairness	Priority Utilization Trade-offs	Priority Bursts
io.prio.class + MQ-DL	✗	✗	✗	✗
io.bfq.weight + BFQ	✗	✗	✗	✗
io.max	✓	—	—	—
io.latency	✓	✗	—	✗
io.cost + io.weight	—	✓	✓	✓

NVMe SSD storage with single-digit microsecond latencies, millions of IOPS of throughput, and GiB/s of bandwidth [38], [78]. However, since SSD resource utilization by workloads is typically low [71], [100], storage resources are virtualized and shared between containerized tenants to improve SSD utilization and prevent resource stranding [64], [71], [73], [84], [100]. Sharing leads to an implicit trade-off between SSD utilization and tenant performance isolation; hence, various works investigate how to co-locate tenants while isolating tenant performance [34], [53], [83]. Nevertheless, there is limited understanding of the structural definition of performance isolation for storage and a common benchmark to compare isolation capabilities. In this work, we address these shortcomings by surveying storage performance isolation to understand and summarize the community’s understanding of isolation, and by benchmarking the isolation capabilities of Linux’s cgroups I/O control knobs.

The first challenge we address is the lack of a unified definition and common benchmark for storage performance isolation, i.e., research studies use different properties and metrics to evaluate isolation. For example, some works consider isolation as minimizing tail latency for priority workloads [61],

whereas others focus on fairness [82]. In short, there is no unified definition of storage performance isolation. This lack of definition limits the effectiveness of isolation efforts and leads to apples-to-pears comparisons when comparing solutions such as the various knobs exposed in cgroups. To address this challenge, we take a two-pronged approach. First, we define isolation by summarizing the state-of-the-practice isolation desiderata using a survey. From this survey, we derive four performance isolation desiderata, which we discuss in detail in §II. Second, to effectively compare isolation solutions, we propose *isol-bench*, a benchmark suite that evaluates all our isolation desiderata for a given system.

The next challenge we address is that the performance isolation capabilities of state-of-the-practice I/O control knobs on NVMe SSDs are unknown for Linux. Enabling isolation properties for NVMe SSDs on the host is challenging because (i) I/O workloads are highly dynamic [74]; (ii) NVMe SSDs have high performance requiring low CPU overhead for I/O control [91]; (iii) SSD performance models differ significantly [54]; and (iv) the SSD-internal flash medium has various performance idiosyncrasies [54] (e.g., asymmetric read and write performance, garbage collection). Various state-of-the-art solutions have been proposed to improve SSD isolation. Such works typically utilize hardware-specific isolation, requiring specialized SSDs such as open-channel SSDs [17], [72], [83], or employ software isolation, typically running in user-space or modified kernels [37]. However, access to specialized hardware is often infeasible due to costs or availability, and custom software solutions require domain expertise; hence, it is equally vital to understand what is already available in the Linux kernel by default. In this work, we use *isol-bench* to benchmark the isolation capabilities of the state-of-the-practice cgroups.

The state-of-the-practice for containers in Linux is to use platforms such as Docker, LXC/LXD, or Podman, where (virtual) SSD resources are managed with Linux’ cgroups [19], [34]. cgroups provides various knobs for I/O control; for example, to limit bandwidth [5], to prioritize workloads, or to do weighted sharing [66], [84]. *isol-bench* evaluates all our survey-defined performance isolation desiderata for cgroups using the *fio* workload generator [14]. We evaluate all I/O control cgroups knobs, which are `io.prio.class`, `io.bfq.weight`, `io.max`, `io.latency`, `io.cost`, and `io.weight`. We present our key findings in Tab. I. Note that some knobs such as `io.prio.class`, `io.bfq.weight` or `io.weight` require I/O schedulers or other cgroups knobs to be active to have a performance effect; hence, we list these as combinations in the table. From our analysis, we report that `io.cost` achieves the highest level of performance isolation on NVMe albeit with a small latency overhead beyond CPU saturation.

We summarize our key contributions as follows:

- A survey on the definition of performance isolation for data center storage, where we summarize the definition to include performance overhead and scalability, proportional fairness, trade-offs between prioritization and

utilization, and priority burst support.

- *isol-bench*, a benchmarking suite for storage performance isolation, which we use to evaluate the isolation capabilities of cgroups I/O control knobs.
- A first-of-its-kind study of performance isolation for cgroups on NVMe SSDs, where, through 10 observations, we determine that `io.cost` achieves most of the isolation desiderata, depending on its configuration.
- To facilitate reproduction, we open-source design and implementation of our code as FAIR data sets at <https://github.com/atlarge-research/isol-bench>.

II. SURVEY ON STORAGE PERFORMANCE ISOLATION

In this section, we survey the definition of performance isolation in the context of (NVMe) SSDs. With this survey, we aim to address two key challenges: there is currently a lack of understanding of the broader definition of storage performance isolation and a lack of a methodology for evaluating such performance isolation. Our survey addresses these problems by distilling a usable isolation definition that can be benchmarked and quantified; specifically, we discuss tenants’ performance requirements, and ubiquitous performance isolation desiderata and how to quantify them.

A. Tenant Performance Requirements

To evaluate performance isolation, we first need to understand tenant requirements in terms of performance. In the literature, we observe that such requirements are commonly defined in terms of *service-level objectives* (SLOs) [13], [20], [27], [30], [31], [35], [36], [40], [45], [46], [49], [55], [59], [65], [69], [93], [99], [101]. Common SLOs include throughput and bandwidth (average, minimum, maximum), latency/slowdown (average, P99 tail, minimum, maximum, CDF), and burstiness for throughput and bandwidth. Here, throughput refers to operations per second and bandwidth refers to I/O bytes per second read from or written to the SSD. Furthermore, in the cloud, many solutions allow selecting a performance profile [1]–[3], [6], [7], e.g., provisioned throughput/bandwidth and average/P99 latency. AWS EBS, for example, has “provisioned iops,” which gives a non-guaranteed approximation of expected throughput [2]. Such profiles are generally tied to volume size and are neither application-defined nor guaranteed.

In this work, we do not use SLOs or profiles; instead, we use the metrics they represent. Practitioners should be able to pick SLOs/profiles according to their needs and a benchmark should enable them to do so. In benchmarking, it is also common to group applications together, for example, into latency-sensitive (L-apps) and throughput-sensitive apps (T-apps) [29], [37], [57], [75]. L-apps have stringent requirements on tail latency, such as caches. T-apps are batch workloads with constraints on the total runtime or average throughput, such as AI training. Other works classify their workloads as latency-critical (LC-app) and best-effort (BE-app) apps [18],

[21], [26], [41], [42], [49], [59], [93], [99], where one workload has no SLOs and the other has latency SLOs. Here, LC-apps and L-apps are identical in practice. An example of a BE-app is a non-critical background job such as archiving. In *isol-bench*, we combine common SLO metric and app definitions and use (1) LC-apps that need low P99 tail latency, (2) batch-apps that require high bandwidth; (3) BE-apps that have no strict performance requirements. For consistency, we will refer to applications or workloads as *apps* in the rest of the work.

B. Performance Isolation Desiderata

Below, we define ubiquitous storage performance isolation desiderata discussed in academic literature in the context of (NVMe) SSDs. Together, these desiderata give insight into the isolation capabilities of I/O control.

Isolation Overhead and Scalability (Desiderata 1): NVMe SSDs deliver microsecond access latencies and GiB/s of bandwidth, increasingly moving the performance bottleneck of I/O control to the host [75], [91]. We consider overhead in ubiquitous metrics such as latency, bandwidth, scalability, and CPU utilization. In this work, we evaluate latency (CDF, P99), bandwidth (utilization, scalability), CPU (utilization, context switches, cycles), and virtual memory (utilization). Overheads affect performance isolation as I/O control competes for resources with tenants, latency bounds can increase, and bandwidth scalability can reduce. Further on, the number of containers and apps on the host is increasing [11], [93], especially in serverless environments. This increase exacerbates potential overheads and requires scalable isolation capabilities. Therefore, to evaluate how many apps can run on a machine in isolation, we also evaluate scalability by scaling the number of apps (through the number of cgroups).

Proportional Fairness (D2): When sharing resources, a common data center objective is ensuring resources are distributed fairly [10], [13], [23], [26], [28], [34], [47], [55], [58], [60], [62], [65], [67], [79]–[82], [82], [85], [87], [92], [97], [101]. This objective is desirable as it allows data centers to indicate the approximate performance an app can expect, even when SSDs are congested. In addition, such control prevents one app from using more resources than allowed. For multi-tenant storage, “shared resources” are bandwidth and throughput. In this work, we evaluate bandwidth fairness as throughput is request size-agnostic. Further on, we evaluate proportional fairness which accounts for weighted sharing, i.e., some apps have higher priority than others. With proportional sharing and n apps, app x ($0 \leq x \leq n$) ideally receives $\frac{weight_x}{\sum_{i=0}^n weight_i}$ of the total bandwidth, where the weight is set by the data center. Storage research occasionally measures fairness by comparing the absolute slowdowns of apps [67] or by comparing the achieved bandwidth/throughput of apps directly [34]. However, such metrics do not yield a single quantifiable number, making it challenging to compare solutions. In this work, we take inspiration from network research and calculate fairness using Jain’s fairness index [39]; here, bandwidth is multiplied by its relative weight.

Prioritization and Utilization Trade-offs (D3): An overarching goal of sharing SSD resources is to maximize resource utilization while maintaining an app’s performance. Various research papers introduce I/O control mechanisms to prioritize the performance of high-priority apps at the cost of utilization [18], [21], [26]–[28], [30], [41], [42], [49], [55]–[57], [59], [61]–[63], [88], [90], [93], [99], [101]. This objective, which we refer to as *prioritization*, is necessary for apps with stringent performance requirements such as real-time apps. Research typically considers an I/O control mechanism capable of prioritization if predefined SLOs of high-priority apps are met or if a specific latency or bandwidth target is met. However, such prioritization is at odds with the utilization desideratum due to, e.g., throttling other apps. Therefore, practitioners should ideally be able to make a trade-off between the level of prioritization and utilization. Hence, in this work, we take a different approach and evaluate whether I/O control mechanisms are capable of making trade-offs between latency and system utilization. Specifically, can a prioritized app’s requirements in latency (LC-app) or bandwidth (batch-app) be fulfilled at a specified SSD utilization level? A ubiquitous metric/indicator for utilization in storage systems [12], [27], [34], [58], [67], [79], [82], [86], [92], [93] and beyond [80] is *work-conservation*. Work-conservation is a binary metric that can be summarized as “when a resource can do more work (i.e., is not saturated), requests to the resource should not be pending”. However, determining whether an SSD is idle is not trivial; the saturation point of throughput and bandwidth in SSDs involves a complex model [34], [54], [59] that depends on the device model, request type (i.e., read, write), access patterns (i.e., sequential, random) and device-internal state (e.g., garbage collection). Further on, while some works intend to measure work conservation with slowdowns [34], other works argue that any requests that are not immediately dispatched to the SSD are non-work-conserving by definition [37]. In this research, we adopt the latter definition of work-conservation. Other research evaluates the aggregated bandwidth/throughput decrement for a specified knob configuration [60], [89]. We use this definition; we evaluate a priority app’s performance trade-offs at a given aggregated bandwidth.

Burst Support (D4): In data centers, many apps are bursty [15], [27], [101]. Bursty apps are disruptive and lead to short I/O spikes. Metrics such as fairness and priority SLOs do not reliably capture I/O control’s capabilities to account for such spikes in contrast to static long-running allocations. I/O control needs to respond quickly to account for bursts. Therefore, in this work, we also explicitly evaluate performance objectives in bursty scenarios. Specifically we evaluate the response time for high-priority bursts for LC-apps and batch-apps.

III. BENCHMARKING SETUP

Our *isol-bench* benchmark uses *fio* [14] as a workload generator. We use cgroups directly for all measurements to avoid interaction effects (i.e., no Docker) and run our benchmarks in a QEMU VM (v9.0.1) with NVMe PCIe passthrough

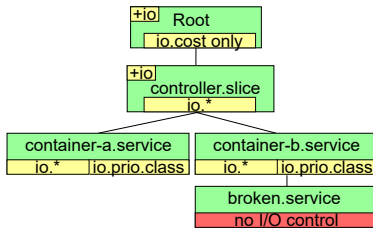


Figure 1: Example cgroups hierarchy, `io.*` is all I/O control except `io.cost` and `io.prio.class`.

as knobs such as `io.latency` frequently required reboots during our benchmarking. The VM’s OS is Ubuntu 22.04.1 with Linux 6.9.0. We run a sample of our workloads on bare-metal and confirmed the performance trends to be similar. All experiments use direct I/O without a file system to ensure only the I/O control of cgroups is evaluated, not the performance effects of other components. Direct I/O with NVMe is also common due to NVMe’s performance capabilities. We use the `io_uring` [76] storage engine in §IV and §V due to its high performance but use `libaio` in §VI due to issues with `fiio` and `io_uring` when throttling.

The host machine has a single-socket Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz (20 logical cores) and 256 GiB of DDR4. We run all benchmarks on Samsung 980 PRO SSDs [78] (flash) and to confirm generalizability we repeat our experiments on Intel Optane SSDs [38]. The Optane is a non-flash SSD; hence, is useful to confirm our results on a different SSD performance model. We use an `io.cost` model generated with Linux’s included `iocost_coef_gen.py` [9] script, which returned a model with a 2.3 GiB/s read saturation point. We disable BFQ’s “low_latency” option as it changes priorities dynamically.

We configure batch- and BE-apps with 4 KiB random reads unless stated at a queue depth (QD) of 256, and LC-apps with 4 KiB random reads at a QD of 1. We run all benchmark configurations for 1 minute unless there are writes. When there are writes, we run for 15 minutes and precondition the SSD a priori (sequential fill, followed by a random write overwrite). We evaluate latency as P99 or as a CDF, and bandwidth as the mean bandwidth. We evaluate CPU utilization (sum of user and system) and virtual memory with `sar` [25], context switches with `fiio` and CPU cycles with `perf`. Fairness experiments are repeated 5 times for standard deviation, others run long enough to have a low deviation.

IV. UNDERSTANDING CGROUPS I/O CONTROL

I/O control refers to all mechanisms used to distribute and manage I/O resources. In Linux, cgroups [4] manages such I/O control for containers and apps. In this section, we discuss cgroups and its five I/O control knobs.

A. I/O Control with cgroups

Control groups (cgroups) is a hierarchical control mechanism that groups processes in the unit of control groups. In Linux, every process is part of such a group by default, and

there is always a root cgroup from which all groups inherit. The core idea is to enable resource management at a higher level of abstraction than processes and to make such management inheritable. We visualize an example cgroups hierarchy in Fig. 1. The resource requirements and limits of a group are shared between all of a group’s (child) processes and groups. For example, if the “controller.slice” group has a defined maximum bandwidth, the maximum refers to the aggregated bandwidth of the processes in “container-a.service,” “container-b.service,” and “broken.service”. cgroups also separates resource management groups from process groups; groups can either delegate management or can hold processes, but never both. A “management group” does not allow any process to be part of the group, but it does allow this group and its direct child groups to set resource control mechanisms. A “process group” can hold any number of processes, but does not allow any of its child groups to have resource control mechanisms, e.g., “broken.service” can not have I/O control knobs. A group becomes a management group when it has a resource controller enabled in `cgroups.subtree_control`, e.g., an I/O controller (marked “+io” in the plot). cgroups is set through `sysfs`.

At the time of writing, cgroups exposes five knobs for I/O control (Note that we are using terminology as defined in `cgroupsv2` [4], [22]): `io.prio.class`, `io.bfq.weight`, `io.max`, `io.latency`, and the `io.cost + io.weight` combination. `io.cost` has two subknobs, `io.cost.model` and `io.cost.qos`, and can only be set in the root group. `io.prio.class` only works in process groups as `io.prio.class` is not inheritable. The other knobs only require the parent group to have an “io” resource controller.

B. cgroups I/O Control Knobs

Below, we detail all five cgroups I/O control knobs. We explain the knobs with examples visualized in Fig. 2. For this plot, we run `fiio` micro-benchmarks with three batch-apps (“A,” “B,” and “C”). “A” runs from 0s–50s, “B” from 10–70s, and app “C” from 20s–50s. Before we run a benchmark, we (re)configure the cgroups based on the evaluated knob. All apps are configured identically; they are rate-limited to 1.5 GiB/s (achievable in isolation, but not in contention), and use 64 KiB random read requests at QD=8. We plot the bandwidth for each app without cgroups knob in Fig. 2a and annotate each app’s start time.

MQ-DL + `io.prio.class`: `io.prio.class` sets the default “I/O scheduling class” for all a group’s processes; an I/O scheduling class is a hint for I/O schedulers, which practitioners can set to idle, best-effort, or realtime. These hints are used by (state-of-the-art) systems [24], [37], [68] for workload (de)prioritization (D3). However, due to its hint-based nature, `io.prio.class`’s behavior depends on the enabled I/O scheduler, *not* on cgroups. In Linux, MQ-DL and BFQ take these hints into consideration. We only evaluate MQ-DL since BFQ does not respect these hints across cgroups. MQ-DL uses separate queues for each priority and dispatches requests from

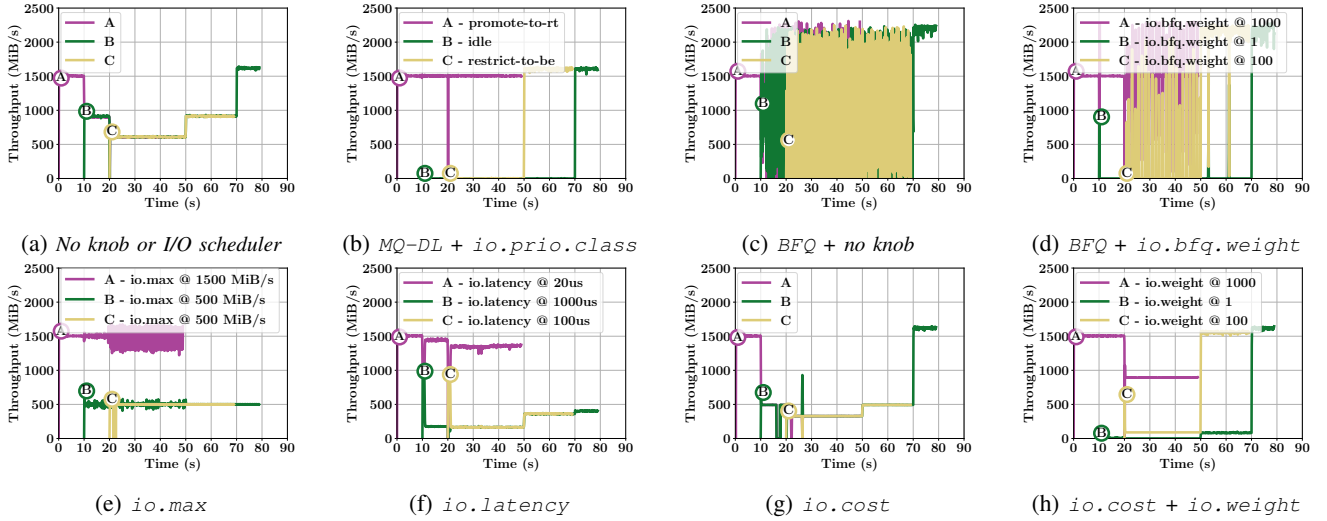


Figure 2: Illustrative examples of all cgroups I/O control knobs with three identical fio workloads: A, B, and C.

the highest priority queues first. To prevent starvation of the lower priority-queues, MQ-DL uses a dispatching timeout for lower-priority requests. In Fig. 2b, we visualize such behavior, where each app has a different class. Here we observe MQ-DL to enable ~ 1.5 GiBs for the highest-priority app, but tens of KiB/s for all other apps when higher-priority apps are running.

BFQ + io.bfq.weight: `io.bfq.weight` is unique to the BFQ scheduler and is usable for proportional fairness and prioritization (D2–3). In BFQ, every group has a relative weight that determines its fair share of throughput; e.g., if a group has a relative weight of two-thirds, it should only use two-thirds of the total throughput. `io.bfq.weight` represents a group’s absolute weight in the range of 1–1,000, where the default is 100. BFQ uses a group’s absolute weight to calculate each group’s relative weight through the cgroups hierarchy. For example, if group “A” has an absolute weight of 1,000 and “B” has a weight of 1, then the relative weight of “B” and its children is $\frac{1}{1,001}$. Weight-based mechanisms such as `io.bfq.weight` are considered challenging to configure in dynamic environments because relative weights change as workloads start and stop. To address such issues, Qiao et al. [70] propose dynamically changing absolute weights.

In Fig. 2c and Fig. 2d, we visualize BFQ with uniform and differing weights respectively. We observe that the bandwidth is unstable in both configurations, which is due to BFQ’s “slice_idle” mechanism, which is required for prioritization but idles every queue for a short while. Further on, BFQ indeed distributes bandwidth based on weight.

io.max: `io.max` allows practitioners to limit a group’s aggregated bandwidth or throughput [16], [60], [70] and allows setting different limits for reads and writes. When a group’s limit is reached, I/O is throttled. `io.max` is static and has no form of prioritization on its own, i.e., it does not guarantee a minimum performance, thereby limiting latency control. It can be used to limit the performance of other lower-priority tenants to prevent potential interference. However,

since `io.max` is static such approaches lead to low utilization (D3) in dynamic environments. Therefore, various state-of-the-art solutions modify the values of `io.max` dynamically [60], [70]. In Fig. 2e, we visualize `io.max`, including its static nature. We observe that all apps respect their maximum bandwidth, but setting a maximum does not guarantee a minimum performance, as B and C both change A’s bandwidth significantly. Further on, the overall utilization is low (33%) as B and C are free to reuse resources when A is inactive.

io.latency: `io.latency` is a knob for prioritization/utilization trade-offs (D3). It allows a group to set a target P90 (static percentile) tail latency per SSD. `io.latency` ensures this latency is met at the cost of the performance of all other groups with a higher target, i.e., a lower priority. Its I/O control works as follows: every 500ms, the kernel checks whether a group’s target latency is violated by comparing the achieved and target P90 latency. If the target is violated, all groups with a higher target are throttled by halving their effective QD, which is observable as “nr_requests” in `sysfs`. This throttling indirectly limits a group’s bandwidth. A group’s QD can only be halved once every interval. If the target is no longer violated, groups are unthrottled by adding “max_nr_requests” / 4 (256 in our experiments) to the effective QD. However, to prevent unthrottling too quickly, `io.latency` also uses a counter called “use_delay.” When QD = 1 and the target is still violated, `use_delay` increases, and every time a group can unthrottle, this variable decreases. The QD can only be recovered when `use_delay` is zero.

Fig. 2f visualizes `io.latency`. When a new workload starts, A’s bandwidth drops shortly. After a few intervals, A recovers and B and C are throttled. The bandwidth of B and C remains at a few hundred MiB/s as `io.latency`’s mechanism can only throttle both workloads to 64 KiB QD = 1 (request-size agnostic). We also observe that the throughput does not recover when A stops, which is due to `use_delay`.

io.cost + io.weight: `io.cost`’s design considers

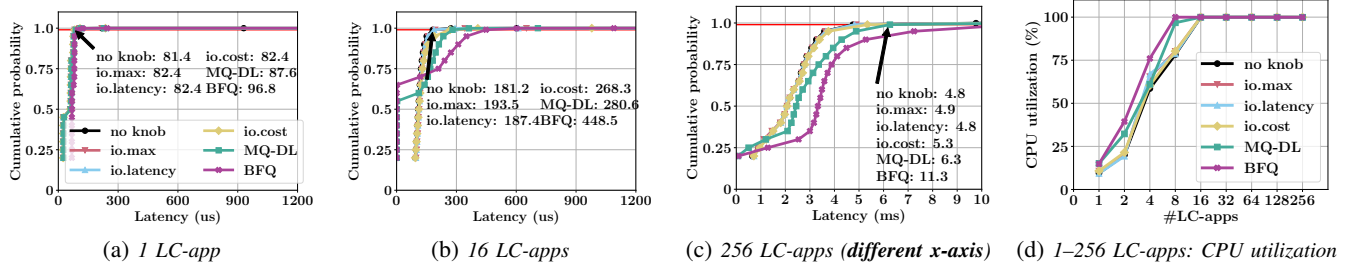


Figure 3: *cgroups* latency and CPU overhead when scaling up from 1 to 256 LC-apps on a single CPU core.

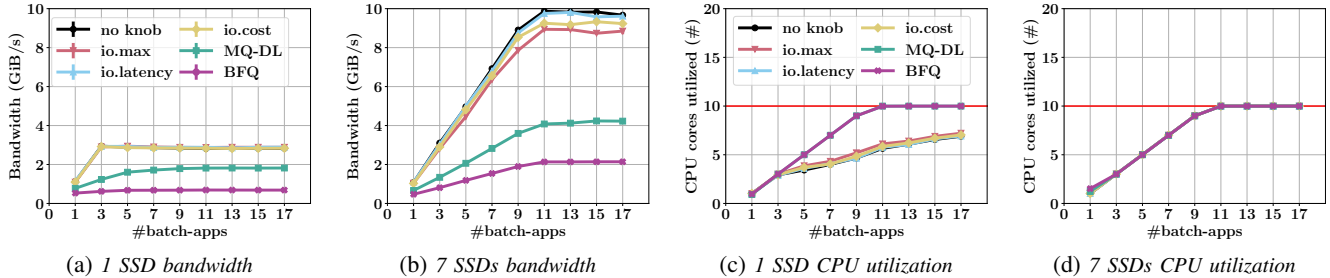


Figure 4: *cgroups* bandwidth and CPU scalability (10 cores) when scaling up batch-apps on 1 and 7 SSDs.

low overhead, proportional fairness, work conservation, and prioritization [33] (D1–3). `io.cost` uses virtual time to determine whether a cgroup can dispatch I/O. Each cgroup can only expend its I/O budget within a window of time. If the cost of an operation exceeds its budget, the operation is delayed by the timing difference between cost and budget. `io.cost` contains two global knobs: `io.cost.model`, a performance model that determines the saturation/congestion points per device, and `io.cost.qos`, which determines when, whether, and how much to restrain groups on saturation. As expected of flash storage, the model supports different costs for different types of operations (i.e., reads, writes), access patterns, and request sizes. Further on, the model can be configured either automatically or statically; see [8] for a database of SSD models. Automatic models are more sensitive to performance fluctuations such as GC; hence, we use the *fio* script included in the Linux kernel to aid in generating a model [9]. The `io.cost.model` is used to determine whether the SSD is congested. Additionally, congestion can be indicated by setting a target tail latency for reads or writes, with `io.cost.qos`. When the system is considered congested, all groups are restrained by changing their budget window, implicitly limiting their bandwidth. The extent to which the budget window can change depends on the `min` and `max` percentile variables in `io.cost.qos`. Lastly, to allow for prioritizing applications, `io.cost` supports `io.weight`, an absolute weight that is set per group between 1–10,000. The relative weight is determined similarly to `io.bfq.weight`. The cost of an operation (`io.cost.model`) is then multiplied by this weight.

In Fig. 2g and Fig. 2h, we visualize `io.cost`. Without weights, we observe that the overall bandwidth is lower than without knobs (Fig. 2a), which is because we use a P95 latency

target of $100\mu\text{s}$ and `io.cost` tries to uphold the target. With weights, we observe that `io.cost` can prioritize apps according to their weight (D3).

V. ISOLATION OVERHEAD (D1) ANALYSIS OF CGROUPS

In this section, we analyze the performance overhead of *cgroups* I/O control in terms of tail latency, bandwidth, CPU utilization, and scalability (D1). Each app runs in a different cgroup. We only measure the overhead of I/O control mechanisms that do the actual I/O control, i.e., we evaluate MQ-DL, BFQ, `io.max`, `io.latency`, and `io.cost`. Furthermore, we ensure that we only measure the overhead of I/O control by limiting the performance impact of the I/O control mechanisms themselves. Specifically, we set a maximum bandwidth beyond device saturation for `io.max`, a multiple-second target latency for `io.latency`, a saturation point beyond the SSD’s saturation point for `io.cost`, and disable BFQ’s “slice_idle”. We use read-only workloads as write workloads will require longer runs and pre-conditioning due to SSD garbage collection. The read and write path are similar; hence, the performance trends should be similar. For generalizability, we evaluated on multiple devices similar to the other evaluations. All of these benchmarks are part of *isol-bench* as described.

Q1: What is the latency overhead of *cgroups* I/O control, and what is its impact on the CPU saturation point? We answer this question by scaling up the number of LC-apps (§III) on a single CPU core from 1 to 256, and evaluating the latency (CDF, P99) and CPU utilization.

In Fig. 3 (a–c), we plot the latency overhead on a single core. We plot the CDF for 1, 16, and 256 co-located apps. We specifically plot 16 apps, as we observed that 16 apps are sufficient to saturate the CPU for all knobs, demonstrating the latency cost when the CPU becomes the bottleneck. On the x-axis, we plot the latency (lower is better), and on the

y-axis, we plot the cumulative probability; we also annotate the P99 latency red. First, `io.max` and `io.latency` do not incur a significant latency increase (although not plotted, the P99 increase observed in some points is unstable). Second, we observe that `io.cost` has a latency overhead past the CPU saturation point; for example, 48.02% compared to none with 16 apps (181.2 to 268.3us). Third, we observe that MQ-DL and BFQ incur a significant increase in tail latency for any number of apps. For example, with just one LC-app, MQ-DL and BFQ have 7.55% and 18.87% higher P99 latency respectively, compared to no knob. We confirmed that this difference increases with the number of LC-apps.

To confirm if this latency increase also translates to a higher CPU utilization, we plot single-core CPU utilization (0–100%) as a function of the number of LC-apps in Fig. 3d. Here, we observe that MQ-DL and BFQ also have a high CPU utilization, e.g., only BFQ saturates the CPU with 8 apps. Furthermore, `io.cost` and `io.max` exhibit a slight increase in CPU utilization; for example, with 8 apps, `io.cost` has 80.27% utilization compared to 78.22% for none. We also run a system profile for 16 apps (past the saturation point) and repeat it 10 times to calculate the standard deviation. First, we observed that the effect on virtual memory is negligible. Second, the CPU cost (not plotted) of MQ-DL and BFQ is indeed higher. Specifically, there are 5.0% and 5.8% more context switches per I/O for BFQ and MQ-DL, respectively, than for none (1.05 and 1.06 to 1.0). Further on, there are 76.2% and 26.8% more cycles per instruction for BFQ and MQ-DL, respectively (44.0K and 31.7K to 25.0K). **Observation 1:** *The latency and CPU overhead of BFQ and MQ-DL are higher than using no I/O control (up to 18.87% higher latency with 1 LC-app). `io.cost` has a latency overhead past the CPU saturation point (e.g., 48.02% with 16 LC-apps). `io.max` and `io.latency` have little overhead for LC-apps.*

Q2: What is the bandwidth scalability of cgroups I/O control? With this experiment, we determine if I/O control can saturate modern NVMe SSDs and bandwidth scalability. To evaluate scalability, we scale up batch-apps (§III) from 1 to 17, which we observed is past the bandwidth saturation point. Further on, we run our benchmarks on 1–7 SSDs (round-robin to all SSDs) to evaluate scalability beyond one SSD. All apps have access to 10 CPU cores.

In Fig. 4 (a) and (b), we plot the bandwidth scalability; on the x-axis, the number of apps, and on the y-axis, the aggregated bandwidth (higher is better). With 1 SSD, MQ-DL and BFQ achieve significantly lower bandwidth. Specifically, MQ-DL achieves a maximum of 1.81 GiB/s and BFQ 0.69 GiB/s, compared to no knob’s 2.94 GiB/s (38.14% and 76.54% lower respectively). This performance plateau is not due to CPU utilization. As we scale up to 7 SSDs, bandwidth increases for all knobs. For example, MQ-DL’s and BFQ’s performance increases to 4.24 GiB/s and 2.14 GiB/s respectively. However, both I/O schedulers are unable to reach less than half the peak bandwidth of none, which is 9.87 GiB/s with 7 SSDs and 10 cores (Fig. 4b). Additionally, we observe that with 7 SSDs, `io.max` and `io.cost` have an overhead

as they achieve 8.94 GiB/s and 9.32 GiB/s, respectively.

To determine if the overhead is due to compute, we plot the accompanying CPU utilization in Fig. 4 (c) and (d). With 1 SSD, every batch-app requires a full core from MQ-DL and BFQ, and the CPU utilization of `io.max` and `io.cost` is higher than without a knob (4.51% and 1.98% respectively, 17 apps). `io.max`’s overhead is thus higher with our batch-app scaling than with LC-app scaling, indicating a higher overhead for bandwidth-heavy scenarios. With 7 SSDs, we observe all knobs to require a full core per batch-app and the CPU overhead differences to be negligible. **O2:** *BFQ and MQ-DL have a bandwidth scalability bottleneck as they are unable to saturate NVMe SSDs (a 38.14% and 76.54% reduction, respectively). Further on, `io.cost` and `io.max` incur a slight decrease in bandwidth (up to 9.41%) and CPU as we scale up the number of NVMe SSDs.*

VI. EVALUATING CGROUPS’ ISOLATION CAPABILITIES

Below, we evaluate the performance isolation capabilities (D2–D5, §II) of all five cgroups I/O control knobs. D1 is addressed in the previous section. We do not combine knobs to evaluate what each knob is able to achieve on its own, i.e., no interaction effects. Each of the desiderata achieved here is a sub-benchmark of `isol-bench` and the implementation closely follows the description of the respective subsections.

A. Proportional Fairness (D2)

We evaluate fairness between cgroups using Jain’s fairness index [39] with weights. This metric has the disadvantage that it does not account for one app sending less than its fair share. Further on, bandwidth fairness is only relevant when the system is congested; limiting an app when more bandwidth is available is counter-intuitive. Therefore, in our experiments, we use four batch-apps per cgroup, which is enough to saturate bandwidth, see Fig. 4.

We evaluate fairness across four scenarios and two weight distributions. These evaluations intend to verify under which (app) conditions fairness is achievable and whether bandwidth can be distributed fairly by practitioners using (approximations of) weights. First, we evaluate fairness with random-read-only batch-apps and uniform weights while scaling up the number of cgroups from 2 to 16. Here, we evaluate fairness without weights and its scalability. Next, we evaluate fairness with weights, where we repeat previous experiment and give each cgroup a “weight” that increases linearly with the number of cgroups. We set weights as follows: `io.weight` for `io.cost`, `io.bfq.weight` for BFQ, `io.prio.class` for MQ-DL, latency targets for `io.latency`, and for `io.max` we use a naive calculation as an approximation for weights ($maximum = \frac{weight}{total_weight} \times max_read_bandwidth$). Lastly, we evaluate if fairness holds under various storage and flash idiosyncrasies. In particular, we evaluate fairness when we configure half of the cgroups to issue I/O with large request sizes of 256 KiB, when we configure half of the groups to issue sequential I/O (i.e., access

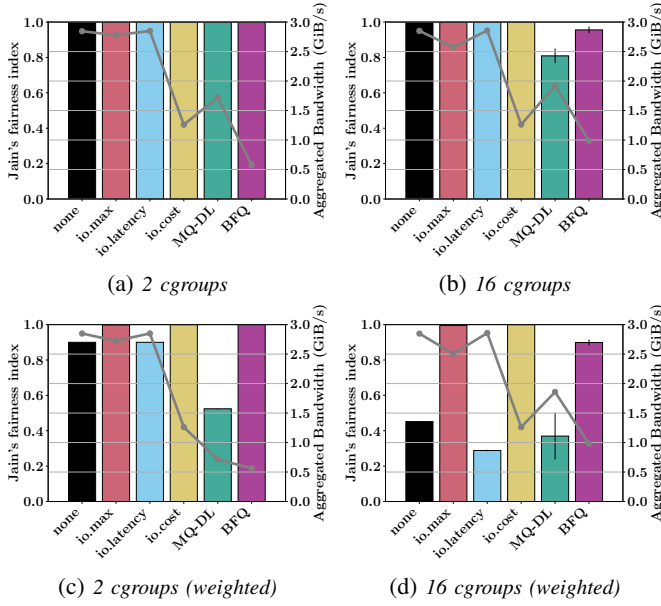


Figure 5: Bandwidth fairness scalability; uniform workload.

patterns), and a configuration where half issue writes (read–write interference, GC).

Q3: What is the fairness of cgroups I/O control with uniform weights and workloads, and what is fairness scalability? In Fig. 5a, we plot fairness as a bar plot (higher is fairer). Here, all knobs have a fairness close to 1; hence, little I/O control is needed. Additionally, in the same plot, we use a line plot for the average aggregated bandwidth per knob to evaluate the utilization cost paid for I/O control. Our findings here largely conform with our prior overhead benchmarks. However, `io.cost` has a notably lower throughput, 1.26 GiB/s compared to `none`'s 2.92 GiB/s. This performance decrement is due to the use of a different, achievable `io.cost.model` and a “min” window of 50% for this experiment (§III). `io.cost` is restricting apps to uphold the model; therefore, `io.cost`'s configurations significantly impact the achievable bandwidth.

In Fig. 5b, we plot fairness for 16 apps, which is past the CPU saturation for all knobs. The aggregated bandwidth does not change as expected. However, when the CPU is saturated, MQ-DL and BFQ get lower fairness than the other knobs (19.09% and 4.43% lower, respectively). This difference increases as we scale up further (not plotted). Neither I/O schedulers thus enables fairness past the CPU saturation. **O3: Workload fairness decreases for MQ-DL and BFQ beyond the CPU saturation point, e.g., we observed 19.09% lower fairness. The other knobs achieve fairness irrespective of CPU saturation, but `io.cost` configurations impact the bandwidth saturation point.**

Q4: What is the fairness of cgroups I/O control with non-uniform weights? In Fig. 5c and Fig. 5d, we plot fairness when the cgroups are assigned linearly increasing weights. We observe that `io.cost`, `io.max`, and BFQ are all able to achieve high fairness with weights. When scaling

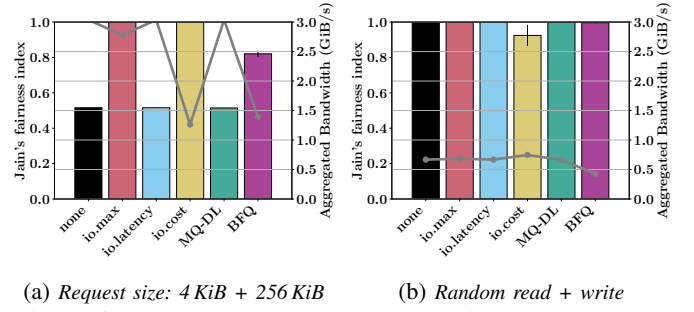


Figure 6: Bandwidth fairness; mixed workloads (2 cgroups).

up to 16 apps, BFQ has the same scalability challenges we observed with uniform weights. Setting “weights” with `io.latency` and MQ-DL notably leads to lower fairness than setting no weight at all; hence, both `io.latency` and `io.prio.class` should not be used to ensure fairness. **O4: `io.cost`, `io.max`, and BFQ (before CPU saturation) are capable of enabling weighted fairness.**

Q5: What is the fairness of cgroups I/O control with non-uniform workloads such as differing request sizes, access patterns, and read–write interference? In Fig. 6a, we plot fairness with large requests. We observe that `io.cost` and `io.max` can maintain high fairness with differing request sizes (4 KiB and 256 KiB), but the other knobs have lower fairness. Specifically, BFQ has a fairness of 0.82 and the others less than 0.52. Here, the groups with large requests get a larger share of bandwidth, e.g., with `none` less than 50 MiB/s is from 4 KiB requests. We do not plot the fairness for different access patterns, as we observe that all knobs lead to high fairness values close to 1. In Fig. 6b, we plot fairness with writes. Writes lead to read–write interference and cause GC, which leads to a significant bandwidth drop, i.e., the aggregated average bandwidth is less than 0.6 GiB/s for all knobs compared to the prior experiment's 3 GiB/s. Here, we observe that only `io.cost` has a lower fairness of 0.89 as the model we used with `io.cost` assigns a higher cost to writes, which indirectly preferentializes read apps. This is expected as `io.cost` assigns a different budget to reads and writes exactly to limit interference (and later GC) effects. Our used fairness metric does not account for such effects; hence, it considers such behavior unfair. **O5: `io.max` enables fairness irrespective of the evaluated workload characteristics. `io.cost` has high fairness as well but exhibits preferential behavior for certain workload characteristics, such as reads, leading to lower fairness in mixed read–write workloads. The other knobs do not allow for fairness under differing request sizes.**

B. Prioritization and Utilization Trade-offs (D3)

In this section, we evaluate prioritization and utilization concurrently as both desiderata are at odds, i.e., prioritizing the performance of one app is frequently at the expense of others. In short, there is an inherent trade-off between utilization and prioritization. Below, we quantify the effect

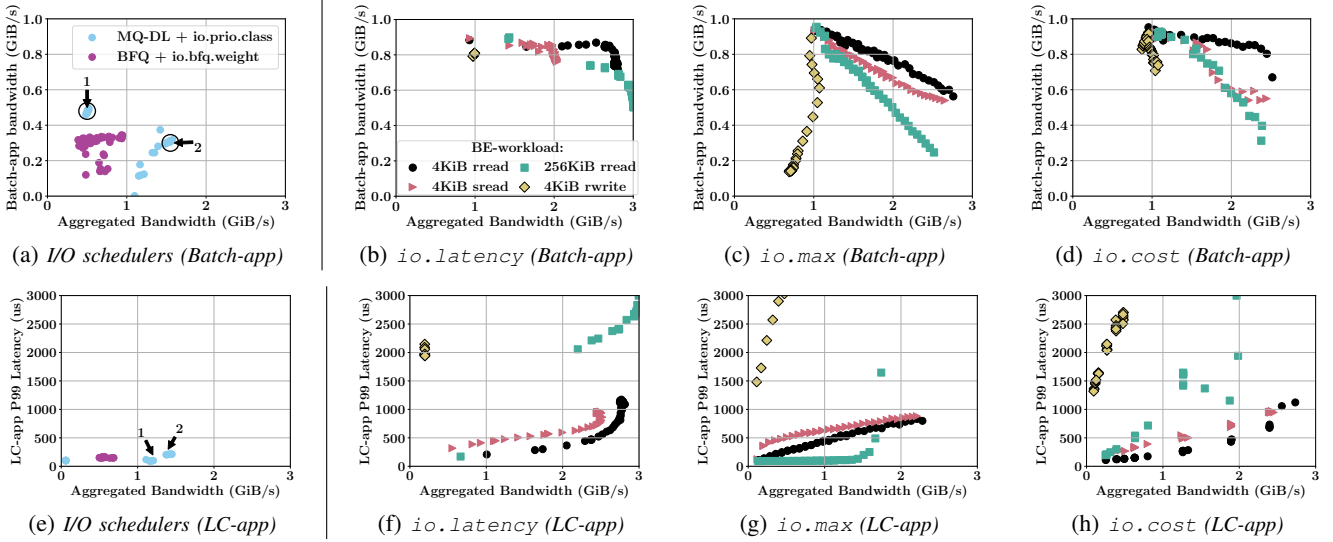


Figure 7: *cgroups* performance/utilization trade-offs; note that the workloads in (a) and (e) are 4 KiB random reads.

on utilization and prioritization, i.e., what is the achievable tail latency or bandwidth for a priority app at a given SSD utilization level. In *cgroups*, a priority can be set with priority classes (`io.prio.class` with MQ-DL), weights (`io.bfq.weight` with BFQ, `io.weight` with `io.cost`), or latency targets (`io.latency` and `io.cost.qos` in `io.cost`). Further on, `io.cost` allows setting a minimum scaling window in `io.cost.qos`, and `io.max` can prioritize by throttling all other workloads.

We evaluate two different prioritization scenarios in a multi-tenant environment: What is the latency of an LC-app at a certain utilization?; what is the bandwidth of a batch-app at a certain utilization? Concurrently with the priority app, we run 4 BE-apps that saturate the SSD in isolation (Fig. 4). BE-apps are low-priority apps. While the BE-apps saturate the SSD isolation, we ensure the batch-apps and LC-apps do not; hence, the trade-off is how much we can reduce the BE-app’s bandwidth to improve the batch-app’s bandwidth or an LC-app’s latency. For each *cgroups* knob, we explore its configuration space and create a Pareto front with the prioritized apps performance on one axis and the aggregated bandwidth utilization on the other. Such a front shows a knob’s trade-off capabilities. Additionally, to account for flash idiosyncrasies, we run the BE-app with various request sizes, access patterns (i.e., random, sequential), and with writes for read–write interference and GC effects.

Q6: What is the trade-off between prioritization and utilization for the MQ-DL and BFQ I/O schedulers? We configure BFQ with `io.bfq.weight` weights from 1 to 1,000 in steps of 25 for the priority app, and MQ-DL with all `io.prio.class` permutations between the priority and BE-app. In Fig. 7a, we plot the trade-offs between a batch-app’s bandwidth (y-axis, higher is better) and the aggregated bandwidth (x-axis, higher is better). First, we observe that the achievable aggregated bandwidth of the I/O schedulers is limited; this is due to overhead, see §V. Second, MQ-DL

is only capable of strict prioritization where either the batch-app, the BE-app or both have full performance, but there are no finer-grained configurations. Here, we marked the scenario where the BE-app is prioritized as “1” and the scenario with equal priorities as “2”. Third, BFQ is ineffective at prioritizing the bandwidth of a single application, despite prior fairness results.

In Fig. 7e, we plot the performance trade-offs for BFQ and MQ-DL between the P99 latency of an LC-app (y-axis, lower is better) and the aggregated bandwidth. We observe that BFQ does not exhibit a latency/utilization trade-off, as the latency differences are minimal. MQ-DL’s trade-offs are coarse-grained with two optimal clusters, annotated 1 and 2. Contrary to our expectations, a lower-priority class in MQ-DL can lead to a lower latency as it reduces the aggregated bandwidth—note that the LC-app has a lower priority than the BE-app at annotation 1. Since the trade-offs are limited for read-only workloads, we do not further evaluate I/O schedulers for request sizes, access patterns, or writes. **O6:** *BFQ is not effective at prioritizing the latency or bandwidth of a high-priority application. MQ-DL supports coarse-grained prioritization where one application benefits from increased bandwidth or lower latency, while other workloads experience low performance.*

Q7: What is the trade-off between prioritization and utilization for `io.latency`? We configure `io.latency` by increasing the P90 latency target of the priority app from 75 μ s (achievable in isolation) to 1.2 ms in steps of 25 μ s. In Fig. 7b, we plot the trade-offs for the batch-app. Additionally, we plot the impact of changing the workload of the BE-app, including random and sequential 4 KiB reads, large 256 KiB random reads, and random 4 KiB random writes. First, we observe that `io.latency`’s configurations allow making trade-offs between priority and utilization. Second, the trade-offs are not work-conserving; various configurations do not lead to higher batch-app bandwidth but do limit the aggregated bandwidth.

For every workload combination, there is an optimal “dent”, where the batch-app performance reduces. Since the optimal setting depends on the app, `io.latency` needs to be configured for the most intrusive BE-app to ensure a priority app’s performance. Third, its trade-offs are less effective for large requests and writes. `io.latency` considers all operations equally expensive and does not account for request sizes. It throttles QD down to a minimum of 1, irrespective of its size, and unthrottles in large steps of 256. Such behavior is ineffective for large requests, as we also observe in the plot, where increasing the target slightly leads to a P99 of more than 2ms. Further on, `io.latency` is also a reactive knob and considers latency a sign of congestion. Such an approach is ineffective for writes, as writes have a delayed effect due to GC, which can happen after a burst of writes, and QD can not be throttled down below 1.

In Fig. 7f, we plot the trade-offs for an LC-app. We observe that `io.latency` achieves the LC-app’s target for random and sequential 4KiB requests at the cost of bandwidth utilization, allowing for latency–bandwidth trade-offs. However, similar to the batch-app experiments, `io.latency` does not achieve its targets with large requests or write apps. **O7:** *`io.latency` is effective for bandwidth-utilization trade-offs and for latency-utilization trade-offs with equal-sized read workloads. However, it is not effective for trade-offs when a workload issues large requests or writes.*

Q8: What is the trade-off between prioritization and utilization for `io.max`? We configure `io.max` by setting a maximum for reads/writes of the BE-app from 80MiB/s to 2.3GiB/s (saturation) in steps of 80MiBs (covering the spectrum of achievable bandwidth). In Fig. 7 (c) and (g), we plot the trade-offs for `io.max`. We observe that `io.max` is effective at making a trade-off between the bandwidth and latency of two applications, confirming its weighted fairness capabilities. Specifically, by limiting the aggregated bandwidth of BE-apps, we can effectively increase the available bandwidth for high-priority batch-apps, and this control works for various request sizes and workloads. However, `io.max` has some challenges. The first challenge with `io.max` is that it, like `io.latency`, does not account for GC or assymmetric operation costs. The second challenge with `io.max` is that, like `io.latency`, it is not work-conservative; however, unlike `io.latency`, it has no prioritization capabilities at all. Specifically, increasing the BE-apps bandwidth improves aggregated bandwidth but reduces the batch-app’s bandwidth or increases the LC-app’s latency. The cost of increasing utilization is thus higher than with `io.latency`. Third, unlike other knobs, `io.max` is static and does not unthrottle in the absence of other workloads. **O8:** *`io.max` allows making trade-offs between prioritization and utilization but is not work-conservative as bandwidth is throttled statically. Further on, it lacks prioritization mechanisms, resulting in lower bandwidth and higher latency for priority apps as utilization increases.*

Q9: What is the trade-off between prioritization and utilization for `io.cost`? `io.cost` has various con-

figuration knobs. For prioritization, we determined through benchmarks (not plotted) that a high value, 10,000 in our case, for the `io.weight` knob and differing the “min” value of `io.cost.qos` allows for trade-offs. We find that changing `io.weight` is efficient for fairness but less for improving utilization. For LC-app trade-offs, we further differ the latency target in `io.cost.qos` at the 99 percentile (we fixed the target at 500 us for the batch-app experiment). In Fig. 7 (d) and (h), we plot the trade-offs. First, we observe that `io.cost` allows for bandwidth and latency trade-offs irrespective of workload. It does not have the same issues with large requests or writes as the other knobs, because `io.cost` assigns a different budget based on a request size, request access pattern, or whether the request is a read or a write. Second, similar to `io.latency`, it is non-work-conservative and allows for prioritization of the high-priority app. It thus has benefits of both `io.max` and `io.latency`. Third, like the other knobs, we find that the trade-offs depend on the characteristics of the BE-app and the SSD model. For example, we find that the trade-offs are different for an Optane SSD. **O9:** *`io.cost` supports prioritization and utilization trade-offs irrespective of request size or operation type, but the trade-offs do differ based on workload characteristics or SSD models.*

C. Performance Isolation during Bursts (D4)

To evaluate each knob’s response time to bursts, we evaluate the time in milliseconds it takes for a priority app (LC-app, batch-app) to get its latency or bandwidth when another low-priority app is running. We run a high-priority app concurrently with a BE-app, similar to §VI-B.

Q10: What is the response time for high-priority bursty apps? We observe (not plotted) that for both the LC-app and the batch-app, `io.latency` can take multiple seconds before the I/O control mechanism stabilizes. This behavior is because `io.latency` (un)throttles after a predefined window, i.e., 500 ms in the evaluated kernel. When throttling QD is halved on our evaluated SSD, this requires 10 throttling events to fully throttle down (1024 to 1), i.e., 5 seconds. In our experiments, the other knobs do not have similar issues. Instead, we observe `io.cost`, `io.max`, and the I/O schedulers to respond in the order of milliseconds to priority apps. **O10:** *`io.latency` is unable to prioritize bursty apps as scaling down can take seconds based on the SSD’s max QD.*

VII. DISCUSSION

Below, we discuss the achievable desiderata for each cgroups knob based on our observations in §V–§VI. From this analysis, we conclude Tab. I. While our conclusions are for direct I/O without a file system, future work can extend `isol-bench`’s abilities and results to higher layers in the storage stack. For example, does the page cache or Linux’s file systems maintain the desiderata of `io.cost`, or is more control needed at higher layer to use its capabilities? Such research can give an end-to-end overview of the isolation capabilities applications can expect in OSes such as GNU/Linux. Further on, tenant goals might conflict. For

example, a latency and a bandwidth goal. In such scenarios, our tool can help with making a trade-off on what knobs are effective. Practitioners will then need to make a decision themselves on what desiderata is more important.

Overhead: I/O schedulers exhibit high latency and CPU overhead, as well as limited bandwidth scalability. `io.cost` incurs a latency overhead beyond the CPU saturation point, a bandwidth overhead as the number of SSDs scales up, and, depending on the used `io.cost.model`, can lead to reduced bandwidth saturation (§VI-A). Both `io.latency` and `io.max` have low overhead.

Proportional Fairness: MQ-DL and `io.latency` do not ensure fairness when workloads are non-uniform or when using weights, and BFQ does not ensure fairness beyond the CPU saturation point. `io.max` enables fairness, but not by itself; it requires practitioners to dynamically translate weights to maximums and adjust values as new groups start or stop. `io.cost` enables fairness, but exhibits preferential behavior to certain apps depending on the used SSD model.

Priority Utilization Trade-offs: BFQ can not prioritize the latency of a prioritized app, and MQ-DL is limited to coarse-grained (3 options) trade-offs for latency and bandwidth. Both `io.latency` and `io.max` allow trade-offs for latency or bandwidth prioritization, but do not include a performance model requiring practitioners to model SSDs themselves. Further on, `io.latency` does not distinguish between request sizes or operation types and has challenges with GC due to its reactive nature and its inability to throttle beyond reducing the maximum QD. `io.max` has no prioritization capabilities on its own, leading to lower performance for high-priority apps as system utilization increases due to resource contention. `io.cost` allows trade-offs between latency and bandwidth in the evaluated scenarios.

Priority Bursts: we evaluate the response time for knobs that have prioritization capabilities, which is crucial for high-priority, bursty apps. `io.cost` and `io.max` can respond in a millisecond, but `io.latency` can take multiple seconds.

VIII. RELATED WORK

A few works complement our cgroups evaluation. Heo et al. [34] discuss `io.cost` and benchmark/discuss a few isolation properties for I/O control, including throughput overhead, work-conservation in dynamic environments, and proportional throughput sharing for reads on SSDs. Park et al. [66] show that file fragmentation reduces isolation for BFQ and `io.cost`. Ahn et al. [12] propose a cgroups knob for weighted proportionality. Two works have characterized I/O scheduler performance overhead and interference for NVMe [75], [91]. We confirmed their results for cgroups and additionally demonstrated that these schedulers have performance isolation challenges. Additionally, various works benchmark and model SSD performance [48], [54], [102] or interference in multi-tenant environments [40].

Various researches propose systems to improve storage performance isolation, i.e., storage stack modifications [32], [53], I/O schedulers [37], [46], [57], [67], [85], [92], [96], [98],

or user-space device sharing or virtualization [68], [95]. Some of these utilize novel SSD interfaces such as open-channel, ZNS or modify flash-internal mappings [35], [44], [51], [51], [52], [63], [71], [72], [83], or are designed for distributed systems [59], [62], [65], [72]. Others involve application adjustments such as adjusting application accuracy based on I/O intensity [69], [70].

IX. CONCLUSION

In this paper, we evaluate the performance isolation capabilities of Linux’s state-of-the-practice cgroups framework with NVMe SSDs. Our results show that the MQ-DL and BFQ I/O schedulers are unable to isolate performance, even with cgroups’s `io.prio.class` and `io.bfq.weight` knobs. Past research has already demonstrated that these I/O schedulers should not be used on NVMe for performance-sensitive workloads; however, here we find that their isolation capabilities in multi-tenant environments are also limited. Further on, `io.latency` and `io.max` knobs have lower overhead, but their isolation capabilities for workloads that use larger requests or writes are limited, and can reduce SSD utilization. `io.max` further requires practitioners to dynamically change configurations to ensure isolation and is not usable for isolation when set statically. `io.cost` has the highest isolation capabilities for NVMe SSDs in terms of fairness, priority-utilization trade-offs, and priority bursts. However, `io.cost` has a latency overhead beyond CPU saturation. Since `io.cost` achieves the most desiderata, we consider `io.cost` to have the highest isolation capabilities. We believe that our results and benchmark are valuable for developers looking for ways to control performance isolation or evaluate their performance isolation implementation.

ACKNOWLEDGMENTS

We thank the IISWC’25 reviewers for their helpful and constructive feedback. We also want to thank the AtLarge group at the Vrije Universiteit Amsterdam for their input.

REFERENCES

- [1] “Amazon EBS General Purpose SSD Volumes,” <https://docs.aws.amazon.com/ebs/latest/userguide/general-purpose.html>, Accessed: 2025-09-02.
- [2] “Amazon EBS Provisioned IOPS Volumes,” <https://aws.amazon.com/ebs/provisioned-iops/>, Accessed: 2025-09-02.
- [3] “Azure Managed Disk Types,” <https://learn.microsoft.com/en-us/azure/virtual-machines/disks-types>, Accessed: 2025-09-02.
- [4] “Control Group v2,” <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html>, Accessed: 2025-09-02.
- [5] “DockerDocs,” <https://docs.docker.com/reference/cli/docker/container/run/>, Accessed: 2025-09-02.
- [6] “GCP – About Local SSD Disks,” <https://cloud.google.com/compute/docs/disks/local-ssd>, Accessed: 2025-09-02.
- [7] “GCP – Configure Disks to Meet Performance Requirements,” <https://cloud.google.com/compute/docs/disks/performance>, Accessed: 2025-09-02.
- [8] “iocost-benchmarks,” <https://github.com/iocost-benchmark/iocost-benchmarks>, Accessed: 2025-09-02.
- [9] “iocost_coef_gen.py,” https://github.com/torvalds/linux/blob/master/tools/cgroup/iocost_coef_gen.py, Accessed: 2025-09-02.

- [10] “VMWare vSphere 7.0 – Managing Storage I/O Resources,” <https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vsphere-resource-management-7-0/managing-storage-i-o-resources.html>, Accessed: 2025-09-02.
- [11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Pionka, and D.-M. Popa, “Firecracker: Lightweight Virtualization for Serverless Applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 419–434.
- [12] S. Ahn, K. La, and J. 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 2016, Denver, CO, USA, June 20-21, 2016*, N. Agrawal and S. H. Noh, Eds. USENIX Association, 2016.
- [13] S. Angel, H. Ballani, T. Karagiannis, G. O’Shea, and E. Thereska, “End-to-end Performance Isolation through Virtual Datacenters,” in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 233–248.
- [14] J. Axboe, “Flexible I/O Tester,” <https://github.com/axboe/fio/tree/bcd46be2adaa4afc32b836ad6137798544a3d80>, Accessed: 2025-06-20.
- [15] B. Berg, D. S. Berger, S. McAllister, I. Grosz, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter *et al.*, “The CacheLib Caching Engine: Design and Experiences at Scale,” in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 753–768.
- [16] S. Bergman, N. Cassel, M. Björling, and M. Silberstein, “ZNSwap: Unblock your Swap,” *ACM Trans. Storage*, vol. 19, no. 2, pp. 12:1–12:25, 2023.
- [17] M. Björling, J. Gonzalez, and P. Bonnet, “LightNVM: The Linux Open-Channel SSD Subsystem,” in *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, G. Kuenning and C. A. Waldspurger, Eds. USENIX Association, 2017, pp. 359–374.
- [18] S. Chen, C. Delimitrou, and J. F. Martínez, “Parties: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 107–120.
- [19] G. E. de Velp, E. Rivière, and R. Sadre, “Understanding the Performance of Container Execution Environments,” in *Proceedings of the 6th International Workshop on Container Technologies and Container Clouds, WOC@Middleware 2020, Delft, The Netherlands, December 07-11, 2020*. ACM, 2020, pp. 37–42.
- [20] C. Delimitrou and C. Kozyrakis, “Paragon: QoS-aware Scheduling for Heterogeneous Datacenters,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds. ACM, 2013, pp. 77–88.
- [21] —, “Quasar: Resource-efficient and QoS-aware Cluster Management,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramanian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 127–144.
- [22] C. Down, “5 Years of cgroup v2: The Future of Linux Resource Control,” 2021.
- [23] A. Fedorova, M. Seltzer, and M. D. Smith, “Improving Performance Isolation on Chip Multiprocessors Via an Operating System Scheduler,” in *16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*. IEEE Computer Society, 2007, pp. 25–38.
- [24] X. Ge, Z. Cao, D. H. Du, P. Ganesan, and D. Hahn, “Hintstor: A Framework to Study I/O Hints in Heterogeneous Storage,” *ACM Trans. Storage*, vol. 18, no. 2, pp. 18:1–18:24, 2022.
- [25] S. Godard, “sysstat,” <https://github.com/sysstat/sysstat>, Accessed: 2025-06-14.
- [26] A. Gulati, I. Ahmad, C. A. Waldspurger *et al.*, “PARDA: Proportional Allocation of Resources for Distributed Storage Access,” in *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA, Proceedings*, M. I. Seltzer and R. Wheeler, Eds. USENIX, 2009, pp. 85–98.
- [27] A. Gulati, A. Merchant, and P. J. Varman, “pClock: an Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems,” in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*, L. Golubchik, M. H. Ammar, and M. Harchol-Balter, Eds. ACM, 2007, pp. 13–24.
- [28] —, “mClock: Handling Throughput Variability for Hypervisor IO Scheduling,” in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 437–450.
- [29] J. Gupta, K. Kant, A. Pal, and J. Biswas, “Configuring and Coordinating End-to-end QoS for Emerging Storage Infrastructure,” *ACM Trans. Model. Perform. Evaluation Comput. Syst.*, vol. 9, no. 1, pp. 4:1–4:32, 2024.
- [30] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi, “MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-aware OS Interface,” in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 168–183.
- [31] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, “The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments,” in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, A. D. Brown and F. I. Popovici, Eds. USENIX Association, 2016, pp. 263–276.
- [32] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, “Multi-Queue Fair Queuing,” in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafir, Eds. USENIX Association, 2019, pp. 301–314.
- [33] T. Heo, D. Schatzberg, A. Newell, S. Liu, S. Dhakshinamurthy, I. Narayanan, J. Bacik, C. Mason, C. Tang, and D. Skarlatos, “IOCost: Block IO Control for Containers in Datacenters,” in *ASPLOS ’22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 595–608.
- [34] —, “IOCost: Block Input-Output Control for Containers in Datacenters,” *IEEE Micro*, vol. 43, no. 4, pp. 80–87, 2023.
- [35] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, “FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs,” in *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, G. Kuenning and C. A. Waldspurger, Eds. USENIX Association, 2017, pp. 375–390.
- [36] L. Huang, A. Parayil, J. Zhang, X. Qin, C. Bansal, J. Stojkovic, P. Zardoshti, P. Misra, E. Cortez, R. Ghelman *et al.*, “Workload Intelligence: Punching Holes through the Cloud Abstraction,” *CoRR*, vol. abs/2404.19143, 2024.
- [37] J. Hwang, M. Vuppapapati, S. Peter, and R. Agarwal, “Rearchitecting Linux Storage Stack for μ s Latency and High Throughput,” in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 113–128.
- [38] Intel, “Intel® Optane™ SSD 900P Series 280GB,” <https://www.intel.com/content/www/us/en/products/sku/123623/intel-optane-ssd-900p-series-280gb-2-5in-pcie-x4-20nm-3d-xpoint/specifications.html>, Accessed: 2025-09-02.
- [39] R. K. Jain, D.-M. W. Chiu, W. R. Hawe *et al.*, “A Quantitative Measure of Fairness and Discrimination,” *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, no. 1, 1984.
- [40] L. N. Jalimiche, C. N. Chakrabortii, C. Choi, and H. Litz, “Enabling Multi-tenancy on SSDs with Accurate IO Interference Modeling,” in *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 2023, pp. 216–232.
- [41] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, “Rubik: Fast Analytical Power Management for Latency-critical Systems,” in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, M. Prvulovic, Ed. ACM, 2015, pp. 598–610.
- [42] H. Kasture and D. Sanchez, “Ubik: Efficient Cache Sharing with Strict QoS for Latency-critical Workloads,” in *Architectural Support for*

- Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 729–742.
- [43] R. Keller Tesser and E. Borin, “Containers in HPC: A Survey,” *J. Supercomput.*, vol. 79, no. 5, pp. 5759–5827, 2023.
- [44] B. S. Kim, “Utilitarian Performance Isolation in Shared SSDs,” in *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*, A. Goel and N. Talagala, Eds. USENIX Association, 2018.
- [45] J. Kim, D. Lee, and S. H. Noh, “Towards SLO complying SSDs through OPS Isolation,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, J. Schindler and E. Zadok, Eds. USENIX Association, 2015, pp. 183–189.
- [46] J. Kim, E. Lee, and S. H. Noh, “I/O Schedulers for Proportionality and Stability on Flash-based SSDs in Multi-tenant Environments,” *IEEE Access*, vol. 8, pp. 4451–4465, 2019.
- [47] J. Kim, D. Kim, and Y. 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*, ser. HotStorage ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 86–92.
- [48] J. Kim, P. Park, J. Ahn, J. Kim, J. Kim, and J. Kim, “SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-box SSDs,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 455–468.
- [49] A. Klimovic, H. Litz, and C. Kozyrakis, “ReFlex: Remote Flash \approx Local Flash,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 345–359.
- [50] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic Ephemeral Storage for Serverless Analytics,” in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 427–444.
- [51] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung, “DC-Store: Eliminating Noisy Neighbor Containers Using Deterministic I/O Performance and Resource Isolation,” in *2019 IEEE International Conference on Web Services, ICWS 2019, Milan, Italy, July 8-13, 2019*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, Eds. IEEE, 2019, pp. 291–295.
- [52] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi, “IODA: A Host/Device Co-design for Strong Predictability Contract on Modern Flash Storage,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 263–279.
- [53] J. Li, R. Shu, J. Lin, Q. Zhang, Z. Yang, J. Zhang, Y. Xiong, and C. Qian, “Daredevil: Rescue Your Flash Storage from Inflexible Kernel Storage Stack,” in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025, pp. 991–1008.
- [54] N. Li, M. Hao, H. Li, X. Lin, T. Emami, and H. S. Gunawi, “Fantastic SSD Internals and How to Learn and Use Them,” in *SYSTOR ’22: The 15th ACM International Systems and Storage Conference, Haifa, Israel, June 13 - 15, 2022*, M. Malka, H. Kolodner, F. Bellosa, and M. Gabel, Eds. ACM, 2022, pp. 72–84.
- [55] N. Li, H. Jiang, D. Feng, and Z. Shi, “Storage Sharing Optimization under Constraints of SLO Compliance and Performance Variability,” *IEEE Trans. Serv. Comput.*, vol. 12, no. 1, pp. 58–72, 2019.
- [56] H. Litz, J. Gonzalez, A. Klimovic, and C. Kozyrakis, “RAIL: Predictable, Low Tail Latency for NVMe Flash,” *ACM Trans. Storage*, vol. 18, no. 1, pp. 5:1–5:21, 2022.
- [57] M. Liu, H. Liu, C. Ye, X. Liao, H. Jin, Y. Zhang, R. Zheng, and L. Hu, “Towards Low-latency I/O Services for Mixed Workloads Using Ultra-low latency SSDs,” in *ICS ’22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, L. Rauchwerger, K. W. Cameron, D. S. Nikolopoulos, and D. N. Pnevmatikatos, Eds. ACM, 2022, pp. 13:1–13:12.
- [58] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu, “vFair: Latency-aware Fair Storage Scheduling via Per-IO Cost-based Differentiation,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, S. Ghandeharizadeh, S. Barahmand, M. Balazinska, and M. J. Freedman, Eds. ACM, 2015, pp. 125–138.
- [59] L. Ma, Z. Liu, J. Xiong, Y. Wu, R. Chen, X. Peng, Y. Zhang, G. Zhang, and D. Jiang, “zQoS: Unleashing Full Performance Capabilities of NVMe SSDs While Enforcing SLOs in Distributed Storage Systems,” in *Proceedings of the 53rd International Conference on Parallel Processing, ICPP 2024, Gotland, Sweden, August 12-15, 2024*. ACM, 2024, pp. 618–628.
- [60] R. Macedo, Y. Tanimura, J. Haga, V. Chidambaram, J. Pereira, and J. Paulo, “PAIO: General, Portable I/O Optimizations With Minor Application Modifications,” in *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, D. Hildebrand and D. E. Porter, Eds. USENIX Association, 2022, pp. 413–428.
- [61] T. Miemietz, H. Weisbach, M. Roitzsch, and H. Härtig, “K2: Work-constraining Scheduling of NVMe-Attached Storage,” in *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*. IEEE, 2019, pp. 56–68.
- [62] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, “Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs,” in *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, F. A. Kuipers and M. C. Caesar, Eds. ACM, 2021, pp. 106–122.
- [63] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, “eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization,” *ACM Trans. Storage*, vol. 20, no. 3, p. 16, 2024.
- [64] A. Mohan, R. Walkup, B. Karacali, M. Chen, A. Kayi, L. Schour, S. Salaria, S. Wen, I. Chung, A. Alim, C. Evangelinos, L. Luo, M. Dombrowa, L. Schares, A. Sydney, P. Maniotis, S. Koteswara, B. Tang, J. Belog, R. Odaira, V. Tarasov, E. Gampel, D. Thorstensen, T. Gershon, and S. Seelam, “Vela: A Virtualized LLM Training System with GPU Direct RoCE,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 1348–1364.
- [65] M. Nanavati, J. Wires, and A. Warfield, “Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage,” in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 17–33.
- [66] J. Park and Y. I. Eom, “Filesystem Fragmentation on Modern Storage Systems,” *ACM Trans. Comput. Syst.*, vol. 41, pp. 3:1–3:27, 2023.
- [67] S. Park and K. Shen, “FIOS: a Fair, Efficient Flash I/O Scheduler,” in *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, W. J. Bolosky and J. Flinn, Eds. USENIX Association, 2012, p. 13.
- [68] B. Peng, C. Guo, J. Yao, and H. Guan, “LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSDs in Clouds,” in *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, J. Lawall and D. Williams, Eds. USENIX Association, 2023, pp. 785–800.
- [69] Z. Qiao, Q. Liu, N. Podhorszki, S. Klasky, and J. Chen, “Taming I/O Variation on QoS-less HPC Storage: What Can Applications Do?” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cui, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 11.
- [70] Z. Qiao, Q. Tian, Z. Qin, J. Wang, Q. g Liu, N. Podhorszki, S. Klasky, and H. Zhu, “Tango: A Cross-layer Approach to Managing I/O Interference over Local Ephemeral Storage,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*. IEEE, 2024, p. 14.
- [71] B. Reidys, J. Sun, A. Badam, S. Noghabi, and J. Huang, “BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms,” in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 17–33.

- [72] B. Reidys, Y. Xue, D. Li, B. Sukhwani, W.-M. Hwu, D. Chen, S. Asaad, and J. Huang, "RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 182–199.
- [73] B. Reidys, P. Zardoshti, I. Goiri, C. Irvine, D. S. Berger, H. Ma, K. Arya, E. Cortez, T. Stark, E. Bak, M. Iyigun, S. Novakovic, L. Hsu, K. Trueba, A. Pan, C. Bansal, S. Rajmohan, J. Huang, and R. Bianchini, "Coach: Exploiting Temporal Patterns for All-Resource Oversubscription in Cloud Platforms," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 164–181.
- [74] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, M. J. Carey and S. Hand, Eds. ACM, 2012, p. 7.
- [75] Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, "BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024*, S. Balsamo, W. J. Knottenbelt, C. L. Abad, and W. Shang, Eds. ACM, 2024, pp. 154–165.
- [76] Z. Ren and A. 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, CHEOPS 2023, Rome, Italy, 8 May 2023*, J. Acquaviva, S. Ibrahim, and S. Byna, Eds. ACM, 2023, pp. 35–45.
- [77] L. V. Rodriguez, A. Gonzalez, P. Poudel, R. Rangaswami, and J. Liu, "Unifying the Data Center Caching Layer: Feasible? Profitable?" in *HotStorage '21: 13th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, USA, July 27-28, 2021*, P. Shilane and Y. Won, Eds. ACM / USENIX Association, 2021, pp. 50–57.
- [78] Samsung, "Samsung 980 PRO PCIe® 4.0 NVMe® SSD 1TB," <https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/>, Accessed: 2025-09-02.
- [79] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC 2013, San Jose, CA, USA, June 26-28, 2013*, A. Birrell and E. G. Sirer, Eds. USENIX Association, 2013, pp. 67–78.
- [80] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in Serving Large Language Models," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, A. Gavrilovska and D. B. Terry, Eds. USENIX Association, 2024, pp. 965–988.
- [81] P. J. Shenoy and H. M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," *Real Time Syst.*, vol. 22, no. 1-2, pp. 9–48, 2002.
- [82] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 349–362.
- [83] J. Sun, B. Reidys, D. Li, J. Chang, M. Snir, and J. Huang, "FleetIO: Managing Multi-Tenant Cloud Storage with Multi-Agent Reinforcement Learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 478–492.
- [84] C. Tang, "Meta's Hyperscale Infrastructure: Overview and Insights," *Commun. ACM*, vol. 68, no. 2, pp. 52–63, 2025.
- [85] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. M. Ghiassi, L. Orosa, J. Gómez-Luna, and O. Mutlu, "FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, M. Annavaram, T. M. Pinkston, and B. Falsafi, Eds. IEEE Computer Society, 2018, pp. 397–410.
- [86] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-defined Storage Architecture," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 182–196.
- [87] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance Insulation for Shared Storage Servers," in *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, Eds. USENIX, 2007, pp. 61–76.
- [88] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to Schedule Long-running Applications in Shared Container Clusters at Scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cui, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 68.
- [89] M. Wang and Y. Hu, "An I/O Scheduler Based on Fine-grained Access Patterns to Improve SSD Performance and Lifespan," in *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong, Eds. ACM, 2014, pp. 1511–1516.
- [90] S. Wang, K. Zhou, Z. Guo, Q. Cao, J. Xu, and J. Yao, "SIndex: An SSD-based Large-scale Indexing with Deterministic Latency for Cloud Block Storage," in *Proceedings of the 53rd International Conference on Parallel Processing, ICPP 2024, Gotland, Sweden, August 12-15, 2024*. ACM, 2024, pp. 1237–1246.
- [91] C. Whitaker, S. Sundar, B. Harris, and N. Altıparmak, "Do We Still Need IO Schedulers for Low-latency Disks?" in *Proceedings of the 15th ACM/USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2023, Boston, MA, USA, 9 July 2023*, A. Anwar, N. Mi, V. Tarasov, and Y. Zhang, Eds. ACM, 2023, pp. 44–50.
- [92] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-Direct Fair Queueing for NVMe SSDs," in *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, M. K. Aguilera and G. Yadgar, Eds. USENIX Association, 2021, pp. 403–415.
- [93] M. Xie, C. Qian, and H. Litz, "En4S: Enabling SLOs in Serverless Storage Systems," in *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*. ACM, 2024, pp. 160–177.
- [94] J. Xu, Y. Chen, Y. Wang, W. Shi, G. Fang, Y. Chen, H. Liao, Y. Wang, H. Lin, Z. Jin, Q. Liu, and W. Chen, "Lightpool: A NVMe-oF-based High-performance and Lightweight Storage Pool Architecture for Cloud-native Distributed Database," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 2024, pp. 983–995.
- [95] S. Yadalani, C. Alverti, V. Karakostas, J. Gandhi, and M. Swift, "BypassD: Enabling fast userspace access to shared SSDs," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafir, Eds. ACM, 2024, pp. 35–51.
- [96] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswani, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level I/O Scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 474–489.
- [97] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: First Class Support for Interactivity in Commodity Operating Systems," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 73–86.
- [98] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October*

8-10, 2018, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 477–492.

- [99] Y. Zhang, Y. Yu, W. Wang, Q. Chen, J. Wu, Z. Zhang, J. Zhong, T. Ding, Q. Weng, L. Yang, C. Wang, J. He, G. Yang, and L. Zhang, “Workload Consolidation in Alibaba Clusters: the Good, the Bad, and the Ugly,” in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, A. Gavrilovska, D. Altinbükten, and C. Binnig, Eds. ACM, 2022, pp. 210–225.
- [100] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, “History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 755–770.
- [101] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, “Prioritymeister: Tail Latency QoS for Shared Networked Storage,” in *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*, E. Lazowska, D. Terry, R. H. Arpaci-Dusseau, and J. Gehrke, Eds. ACM, 2014, pp. 29:1–29:14.
- [102] A. Zuck, P. Gühring, T. Zhang, D. E. Porter, and D. Tsafir, “Why and How to Increase SSD Performance Transparency,” in *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 2019, pp. 192–200.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Other products and service names might be trademarks of IBM or other companies.